دليل JavaScript الشامل

الجزء الأول

تأليف

Ilya Kantor

ترجمة

مجموعة من المترجمين

دليل JavaScript الشامل

الجزء الأول

مرجع شامل متكامل إلى لغة JavaScript البرمجية

Book Title: The JavaScript Language - Part 1	**اسم الكتاب**: دليل JavaScript الشامل - الجزء الأول
Author: Ilya Kantor	**المؤلف**: إيليا كانتور
Translator: Safa Al Fulaij, Saja Al Haj, Ishtiaq Muhammad, Nour Agha, Mohamed Lahah	**المترجم**: صفا الفليج، سجى الحاج، اشتياق محمد، نور آغا، محمد لحلح
Editor: Jamil Bailony	**المحرر**: جميل بيلوني
Cover Design: Mustafa Zaytony	**تصميم الغلاف**: مصطفى زيتوني
Publication Year: 2023	**سنة النشر**: 2023
Edition: 1.0	**رقم الإصدار**: 1.0

بعض الحقوق محفوظة - أكاديمية حسوب.

أكاديمية حسوب أحد مشاريع شركة حسوب محدودة المسؤولية.

مسجلة في المملكة المتحدة برقم 07571594.

https://academy.hsoub.com

academy@hsoub.com

Copyright Notice

The author publishes this work under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0).

You are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material

This license is acceptable for Free Cultural Works.
The licensor cannot revoke these freedoms as long as you follow the license terms:

- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- NonCommercial — You may not use the material for commercial purposes.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
Read the text of the full license on the following link:

إشعار حقوق التأليف والنشر

ينشر المصنِّف هذا العمل وفقا لرخصة المشاع الإبداعي نَسب المُصنَّف - غير تجاري - الترخيص بالمثل 4.0 دولي (CC BY-NC-SA 4.0).

لك مطلق الحرية في:

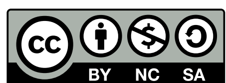
- المشاركة — نسخ وتوزيع ونقل العمل لأي وسط أو شكل.
- التعديل — المزج، التحويل، والإضافة على العمل.

هذه الرخصة متوافقة مع أعمال الثقافة الحرة.
لا يمكن للمرخِّص إلغاء هذه الصلاحيات طالما اتبعت شروط الرخصة:

- نَسب المُصنَّف — يجب عليك نَسب العمل لصاحبه بطريقة مناسبة، وتوفير رابط للترخيص، وبيان إذا ما قد أُجريت أي تعديلات على العمل. يمكنك القيام بهذا بأي طريقة مناسبة، ولكن على ألا يتم ذلك بطريقة توحي بأن المؤلف أو المرخِّص مؤيد لك أو لعملك.
- غير تجاري — لا يمكنك استخدام هذا العمل لأغراض تجارية.
- الترخيص بالمثل — إذا قمت بأي تعديل، تغيير، أو إضافة على هذا العمل، فيجب عليك توزيع العمل الناتج بنفس شروط ترخيص العمل الأصلي.

منع القيود الإضافية — يجب عليك ألا تطبق أي شروط قانونية أو تدابير تكنولوجية تقيد الآخرين من ممارسة الصلاحيات التي تسمح بها الرخصة.
اقرأ النص الكامل للرخصة عبر الرابط التالي:

https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode



The illustrations used in this book is created by the author and all are licensed with a license compatible with the previously stated license.

الصور المستخدمة في هذا الكتاب من إعداد المؤلف وهي كلها مرخصة برخصة متوافقة مع الرخصة السابقة.

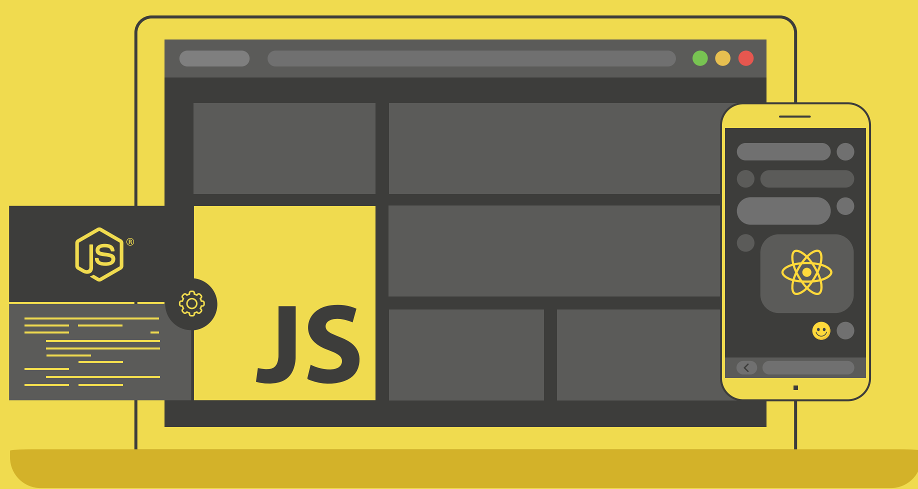
عن الناشر

أُنتج هذا الكتاب برعاية شركة حسوب وأكاديمية حسوب.

تهدف أكاديمية حسوب إلى تعليم البرمجة باللغة العربية وإثراء المحتوى البرمجي العربي عبر توفير دورات برمجة وكتب ودروس عالية الجودة من متخصصين في مجال البرمجة والمجالات التقنية الأخرى، بالإضافة إلى توفير قسم للأسئلة والأجوبة للإجابة على أي سؤال يواجه المتعلم خلال رحلته التعليمية لتكون معه وتؤهله حتى دخول سوق العمل.

حسوب شركة تقنية في مهمة لتطوير العالم العربي. تبني حسوب منتجات تركِّز على تحسين مستقبل العمل، والتعليم، والتواصل. تدير حسوب أكبر منصتي عمل حر في العالم العربي، مستقل وخمسات ويعمل في فيها فريق شاب وشغوف من مختلف الدول العربية.

دورة تطوير التطبيقات باستخدام لغة JavaScript



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

المحتويات باختصار

تمهيد	40
1. مقدمة	43
2. أساسيات جافاسكربت	57
3. الاعتناء بجودة الشيفرة	187
4. الكائنات: تأسيس المفاهيم	235
5. أنواع البيانات	310
6. التعامل المتقدم مع الدوال	487
7. ضبط خاصيات الكائنات	612
8. الوراثة النموذجية	627
9. الأصناف Classes	670
10. التعامل مع الأخطاء	733
11. الوعود واللاتزامن/الانتظار	763
12. المولدات والمكررات المتقدمة	825
13. الوحدات Modules	850
14. مواضيع متفرقة	876

جدول المحتويات

تمهيد **40**

حول الكتاب 40

استخدام الشيفرة 41

المساهمة 42

1. مقدمة **43**

1.1 مقدمة إلى لغة جافاسكربت 44

1.1.1 ما هي لغة جافاسكربت JavaScript؟ 44

1.1.2 ما يمكن أن تفعله جافاسكربت ضمن المتصفح؟ 45

1.1.3 المهام التي لا يمكن لجافاسكربت إنجازها ضمن المتصفح 46

1.1.4 ما المميز في لغة جافاسكربت؟ 47

1.1.5 اللغات المعتمدة على لغة جافاسكربت 48

1.1.6 الخلاصة 48

1.2 الأدلة والمواصفات القياسية المعتمدة 49

1.2.1 المواصفات القياسية Specification 49

1.2.2 الأدلة Manuals 49

1.2.3 جداول التوافقية مع المتصفحات 50

1.3 محررات الشيفرة البرمجية 51

1.3.1 بيئة التطوير المتكاملة 51

1.3.2 المحررات البسيطة 52

1.3.3 ما هو محرري المفضل الذي سأستخدمه؟ 52

1.4 أدوات المطور 54

1.4.1 أدوات المطور في متصفح Google Chrome 54

1.4.2 أدوات المطور في متصفحي FireFox و Safari وغيرهما 55

ا. أدوات المطور في متصفح Safari 55

ب. الإدخال متعدد الأسطر 56

1.4.3 الخلاصة 56

2. أساسيات جافاسكربت **57**

2.1 المثال الأول: مرحبًا يا عالم! 58
2.1.1 الوسم script 58
2.1.2 الاستعمال القديم للعنصر script 59
ا. الخاصية type 59
ب. الخاصية language 59
ج. التعليقات قبل وبعد السكربتات 59
2.1.3 الاستعمال الحديث للعنصر script 59
2.1.4 الخلاصة 61
2.1.5 تمارين 61
ا. إظهار تنبيه 61
ب. إظهار تنبيه باستخدام سكربت خارجي 61
2.2 بنية الشيفرة البرمجية 63
2.2.1 التعليمات البرمجية 63
2.2.2 الفاصلة المنقوطة 63
2.2.3 التعليقات 65
2.3 الوضع الصارم: النمط الحديث لكتابة الشيفرات 67
2.3.1 الموجه "use strict" 67
2.3.2 طرفية المتصفح 68
2.3.3 استخدم "use strict" دومًا 68
2.4 المتغيرات Variables 69
2.4.1 المتغير 69
2.4.2 مثال واقعي للمتغيرات 70
2.4.3 تسمية المتغيرات 72
2.4.4 الثوابت 74
ا. متى تستخدم الأحرف الكبيرة في تسمية الثوابت؟ 74
2.4.5 تسمية الأشياء تسمية صحيحة 75
2.4.6 الخلاصة 76
2.4.7 تمارين 76
ا. العمل مع المتغيرات 76

ب. اختيار الأسماء المناسبة 77
ج. الثوابت بالأحرف الكبيرة؟ 78
2.5 أنواع البيانات Data Types 79
2.5.1 النوع number: الأعداد 79
2.5.2 النوع BigInt: الأعداد الكبيرة 80
2.5.3 النوع string: السلاسل النصية (النصوص) 81
2.5.4 النوع boolean: قيمة ثنائية (بوليانية) 82
2.5.5 القيمة الخالية: null 82
2.5.6 القيمة الغير معرفة: undefined 82
2.5.7 النوع object والنوع symbol: الكائنات والرموز 83
2.5.8 العامل typeof 83
2.5.9 الخلاصة 84
2.5.10 تمارين 85
ا. إشارات اقتباس السلاسل النصية 85
2.6 الدوال التفاعلية: alert، prompt، confirm 86
2.6.1 الدالة alert 86
2.6.2 الدالة prompt 86
2.6.3 الدالة confirm 87
2.6.4 الخلاصة 87
2.6.5 تمارين 88
ا. صفحة بسيطة 88
2.7 التحويل بين الأنواع 89
2.7.1 التحويل إلى سلسلة نصية 89
2.7.2 التحويل إلى عدد 89
ا. جمع السلاسل النصية عبر المعامل + 90
2.7.3 التحويل إلى قيمة منطقية 91
2.7.4 الخلاصة 91
2.7.5 تمارين 92
ا. تحويلات الأنواع 92

2.8 عوامل Operators 94
2.8.1 مصطلحات جديدة: أحادي، ثنائي، عامل 94
2.8.2 وصل سلاسل نصية عبر العامل + 94
2.8.3 خاصية التحويل العددي 95
2.8.4 ترتيب أولوية العمليات 96
2.8.5 عامل الإسناد = 97
ا. عامل الإسناد = يُرجع قيمة 98
2.8.6 عامل باقي القسمة % 98
2.8.7 عامل القوة 99
2.8.8 عاملات الزيادة ++ والنقصان -- 99
ا. عاملات زيادة / نقصان داخل عاملات أخرى 101
2.8.9 العاملات الثنائية 101
2.8.10 عاملات التعديل المباشر على المتغير 102
2.8.11 عامل الفاصلة , 103
2.8.12 تمارين 104
ا. النموذج السابق والنموذج اللاحق 104
ب. نتيجة عملية الإسناد 104
2.9 عاملات الموازنة 105
2.9.1 النتيجة عبارة عن قيمة منطقية 105
2.9.2 موازنة السلاسل النصية 105
2.9.3 موازنة بين أنواع البيانات المختلفة 106
2.9.4 معامل المساواة الصارمة 107
2.9.5 الموازنة مع قيمة فارغة null وغير معرفة undefined 108
ا. الرياضيات وعاملات الموازنة الأخرى =< => < > 108
ب. نتيجة غريبة: null مقابل 0 108
2.9.6 تجنب المشاكل 109
2.9.7 الخلاصة 109
2.9.8 تمارين 110
ا. الموازنات 110

2.10 العاملات الشرطية 112
2.10.1 التعبير الشرطي if 112
2.10.2 التحويل المنطقي 112
2.10.3 الكتلة الشرطية else 113
2.10.4 الشروط المتعددة else if 113
2.10.5 العامل الشرطي ? 114
2.10.6 العامل الشرطي ؟ المتعدد 115
ا. الاستخدام غير التقليدي للعامل ؟ 116
2.10.7 تمارين 117
ا. التعبير الشرطي if (سلسلة نصية مع صفر) 117
ب. اسم JavaScript 117
ج. إظهار إشارة 118
د. تحويل التعبير الشرطي if إلى صيغة العامل ? 118
ه. تحويل التعبير الشرطي if..else إلى صيغة العامل ? 119
2.11 العاملات المنطقية 120
2.11.1 العامل || (OR) المنطقي 120
ا. العامل OR يرجع القيمة الصحيحة الأولى 121
2.11.2 العامل && (AND) المنطقي 123
ا. العامل AND يرجع القيمة الخطأ الأولى 123
ب. أولوية التنفيذ 124
ج. لا تستعمل || أو && مكان الشرط if 124
2.11.3 العامل المنطقي ! (NOT) 125
2.11.4 تمارين 126
ا. ما هي نتيجة العامل OR 126
ب. ما هي نتيجة سلسلة من العامل OR للدالة alert 126
ج. ما هي نتيجة العامل AND 127
د. ما هي نتيجة سلسلة من العامل AND للدالة alert 127
ه. الناتج من السلسلة (OR AND OR) 127
و. حصر قيمة متغير ضمن مجال 128

ز. حصر متغير خارج مجال 128

ح. سؤال باستخدام التعبير الشرطي if 128

ط. التحقق من تسجيل الدخول 129

2.12 عامل الاستبدال اللاغي "??" 132

2.12.1 الموازنة باستخدام العامل || 132

2.12.2 أولوية عامل ?? 133

2.12.3 الخلاصة 134

2.13 حلقتا التكرار while و for 135

2.13.1 حلقة التكرار while 135

2.13.2 حلقة التكرار do..while 136

2.13.3 حلقة التكرار for 136

ا. التصريح عن المتغيرات داخل نطاق الحلقة 138

ب. تجاهل بعض أجزاء التحكم بحلقة التكرار for 138

2.13.4 إيقاف حلقة التكرار 139

2.13.5 الاستمرار في التكرار التالي 139

ا. تقليل مستوى التداخل عبر التعليمة continue 140

ب. لا يُستخدم الموجهان break/continue في المعامل الشرطي الثلاثي ? 140

2.13.6 تسمية حلقات التكرار 141

2.13.7 الخلاصة 142

2.13.8 التمارين 143

ا. قيمة حلقة التكرار الأخيرة 143

ب. ما هي القيم التي ستظهرها حلقة التكرار while؟ 143

ج. ما القيم التي ستظهرها حلقة التكرار for؟ 145

د. إخراج الأعداد الزوجية باستخدام حلقة التكرار 145

ه. استبدال حلقة التكرار for بحلقة التكرار while 146

و. كرر حتى يكون الإدخال صحيحًا 146

ز. إظهار الأعداد الأولية 147

2.14 التعليمة switch 149

2.14.1 الصياغة 149

2.14.2 مثال تطبيقي 149
2.14.3 تجميع حالات case متعدِّدة 151
2.14.4 نوع القيم 152
2.14.5 تمارين 152
ا. أعد كتابة المبدِّل switch بصيغة الشرط if 152
ب. أعد كتابة الشرط if بصيغة المبدِّل switch 153
2.15 الدوال في JavaScript 155
2.15.1 تعريف الدوال 155
2.15.2 المتغيرات المحلية 156
2.15.3 المتغيرات العامة 156
2.15.4 المعاملات 157
2.15.5 القيم الافتراضية 158
ا. تقييم المعاملات الافتراضية 159
ب. الطراز القديم للمعاملات الافتراضية 159
2.15.6 إرجاع قيمة 159
ا. إرجاع قيمة غير مُعرَّفة 160
ب. إضافة سطر جديد بين الموجه return والقيمة المعادة 161
2.15.7 تسمية الدوال 161
ا. دالة واحدة مقابل إجراء واحد 162
ب. أسماء دوال قصيرة جدًا 163
2.15.8 الدوال == التعليقات 163
2.15.9 الخلاصة 164
2.15.10 التمارين 165
ا. هل وجود التعبير البرمجي else مهم في الشيفرة؟ 165
ب. أعد كتابة الدالة باستخدام المعامل ? أو المعامل || 166
ج. الدالة min(a,b) 167
د. الدالة pow(x,n) 167
2.16 تعابير الدوال 169
ا. وجود الفاصلة المنقوطة في نهاية تعبير الدالة 170

2.16.2 دوال رد النداء 171
2.16.3 تعبير الدوال مقابل التصريح عن الدوال 172
2.16.4 الخلاصة 176
2.17 أساسيات الدوال السهمية 177
2.17.1 الدوال السهمية متعددة الأسطر 178
2.17.2 الخلاصة 179
2.17.3 تمارين 179
ا. اكتب الشيفرة التالية مجدَّدًا باستعمال الدوال السهمية 179
2.18 مراجعة لما سبق 180
2.18.1 صياغة الشيفرة 180
2.18.2 الوضع الصارم 181
2.18.3 المتغيرات 181
2.18.4 الدوال التفاعلية 182
2.18.5 المعاملات 183
ا. المعاملات الحسابية 183
ب. معاملات الإسناد 183
ج. المعاملات الثنائية 183
د. المعاملات الشرطية 183
ه. المعاملات المنطقية 183
و. عامل الاستبدال اللاغي 183
ز. معاملات الموازنة 184
ح. معاملات أخرى 184
2.18.6 حلقات التكرار 184
2.18.7 التعبير switch 185
2.18.8 الدوال 185
2.18.9 المزيد قادم 186
3. الاعتناء بجودة الشيفرة 187
3.1 تنقيح الأخطاء في Chrome 188
3.1.1 جزء الموارد Sources 188

3.1.2 طرفية التحكم (Console) 189
3.1.3 نقاط التوقف (Breakpoints) 190
3.1.4 تعليمة Debugger 191
3.1.5 توقف وتحقق 191
3.1.6 تتبع التنفيذ 192
3.1.7 التسجيل Logging 194
3.1.8 الخلاصة 194
3.2 نمط كتابة الشيفرة 196
3.2.1 الصياغة 196
ا. الأقواس المعقوصة {} 196
ب. طول السطر 197
ج. المسافة البادئة 198
د. الفواصل المنقوطة ";" 199
ه. مستويات التداخل 199
3.2.2 موضع الدوال 201
3.2.3 دليل نمط كتابة الشيفرة 202
3.2.4 منقحات الصياغة التلقائية Automated Linters 203
3.2.5 الخلاصة 204
3.2.6 تمارين 204
ا. نمط تكويد سيئ 204
3.3 التعليقات 207
3.3.1 التعليقات السيئة 207
ا. طريقة: أخرج الدوال 207
ب. طريقة: أنشئ دوال 208
3.3.2 التعليقات الجيدة 210
3.3.3 الخلاصة 211
3.4 شيفرة النينجا البرمجية 212
3.4.1 البلاغة في الإيجاز 212
3.4.2 المتغيرات ذات الحرف الواحد 212

3.4.3 استخدم الاختصارات 213
3.4.4 حلق عاليًا واستخدم التجريد 213
3.4.5 اختبار الملاحظة 214
3.4.6 مرادفات ذكية 214
3.4.7 أعد استخدام الأسماء 214
3.4.8 الشرطة السفلية للمتعة 215
3.4.9 أظهر حبك 215
3.4.10 داخِل المتغيرات الخارجية 216
3.4.11 آثار جانبية في كل مكان 216
3.4.12 دوال قوية! 217
3.4.13 الخلاصة 217
3.5 الاختبار الآلي باستخدام mocha 218
3.5.1 لم نحتاج الاختبارات؟ 218
3.5.2 التطوير المستند إلى السلوك 218
3.5.3 تطوير الدالة "pow": الوصف 218
3.5.4 تدفق التطوير 219
3.5.5 المواصفات أثناء التنفيذ 220
3.5.6 التنفيذ الأولي 222
3.5.7 تطوير الوصف 223
3.5.8 تطوير التنفيذ 224
3.5.9 دالة describe متداخلة 225
3.5.10 توسيع الوصف 227
3.5.11 الخلاصة 230
3.5.12 تمارين 231
ا. ما الخطأ في الاختبار التالي؟ 231
3.6 تعويض نقص دعم المتصفحات 233
3.6.1 Babel 233
3.6.2 أمثلة من الشرح 234
4. الكائنات: تأسيس المفاهيم 235

4.1 الكائنات Objects 236
4.1.1 القيم المُجرَّدة والخاصيات 237
4.1.2 الأقواس المعقوفة 239
ا. الخاصيات المحسوبة 240
4. يمكن استخدام الأسماء المحجوزة مع أسماء الخاصيات 241
4.1.3 اختزال قيم الخاصيات 242
4.1.4 فحص الكينونة 243
ا. استخدام "in" مع الخاصيات التي تُخزن القيمة undefined 244
4.1.5 الحلقة for...in 244
ا. الترتيب في الكائنات 245
4.1.6 الخلاصة 247
4.1.7 تمارين 248
ا. مرحبًا، بالكائن 248
ب. التحقق من الفراغ 248
ج. جمع خاصيات الكائن 249
د. ضرب الخاصيات العددية بالقيمة 2 250
4.2 نسخ كائن: الفرق بين القيمة والمرجع 251
4.2.1 الموازنة بحسب المرجع 253
ا. الكائنات الثابتة 253
4.2.2 الاستنساخ والدمج 254
4.2.3 الاستنساخ المتداخل 256
4.2.4 الخلاصة 257
4.2.5 تمارين 257
ا. كائنات ثابتة؟ 257
4.3 كنس البيانات المهملة 259
4.3.1 قابلية الوصول 259
4.3.2 مثال بسيط 259
4.3.3 مرجعان لكائن 260
4.3.4 الكائنات المتداخلة 261

4.3.5 جزيرة غير قابلة للوصول 263
4.3.6 الخوارزميات الداخلية 264
4.3.7 الخلاصة 267
4.4 الدوال في الكائنات واستعمالها this 268
4.4.1 أمثلة على الدوال 268
ا. اختصار الدالة 269
4.4.2 الكلمة المفتاحية "this" في الدوال 270
4.4.3 "this" غير محدودة النطاق 271
4.4.4 ميزة داخلية: النوع المرجعي 273
4.4.5 الدوال السهمية لا تحوي this 275
4.4.6 الخلاصة 275
4.4.7 تمارين 276
ا. فحص الصياغة 276
ب. شرح قيمة this 277
ج. استخدام this في الكائن معرَّف باختصار عبر الأقواس 278
د. إنشاء آلة حاسِبة 280
ه. التسلسل 281
4.5 الباني والعامل new 283
4.5.1 الدالة البانية 283
ا. new function() { … } 284
4.5.2 وضع اختبار الباني: new.target 285
4.5.3 ما تُرجِعه الدوال البانية 286
4.5.4 الدوال في الباني 287
4.5.5 الخلاصة 288
4.5.6 تمارين 288
ا. دالتين - كائن واحد 288
4.5.7 إنشاء حاسبة جديدة 289
4.5.8 إنشاء مجمِّع 290
4.6 التسلسل الاختياري "?." 292

4.6.1 المشكلة 292
4.6.2 تسلسل اختياري Optional chaining 292
4.6.3 سلوك الطريق المختصر Short-circuiting 294
4.6.4 حالات أخرى ().? و [].? 294
4.6.5 الخلاصة 295
4.7 النوع الرمزي Symbol 296
4.7.1 الرموز 296
4.7.2 خاصيات "خفية" 297
ا. الرموز في التعريف المختصر لكائن 298
4.7.3 تتخطى for…in الرموز 298
4.7.4 الرموز العامة 300
ا. Symbol.keyFor 300
4.7.5 رموز النظام 301
4.7.6 الخلاصة 301
4.8 التحويل من نوع كائن إلى نوع أولي 303
4.8.1 التحويل إلى أنواع أساسية عبر toPrimitive 303
ا. إلى سلسلة نصية string 303
ب. إلى عدد number 303
ج. إلى نوع افتراضي default 304
د. تابع التحويل Symbol.toPrimitive 305
ه. التحويل إلى نص عبر toString أو valueOf 305
4.8.2 الأنواع المعادة 307
4.8.3 عمليات تحويل إضافية 307
4.8.4 الخلاصة 308
5. أنواع البيانات 310
5.1 توابع الأنواع الأولية 311
5.1.1 نوع أولي مثل كائن 312
ا. البانيات String أو Number أو Boolean هي للاستخدام الداخلي فقط 313
ب. ليس لدى القيمتان الأوليتان null/undefined توابع 313

5.1.2 الخلاصة 314
5.1.3 المهام 314
ا. هل من الممكن إضافة خاصية نصية؟ 314
5.2 النوع number: الأعداد 316
5.2.1 طرائق أخرى لكتابة عدد 316
ا. الأعداد الست عشرية، والثنائية والثمانية 317
5.2.2 toString(base) 317
5.2.3 التقريب Rounding 318
5.2.4 حسابات غير دقيقة 319
5.2.5 الفحص: isFinite و isNaN 322
5.2.6 parseInt و parseFloat 323
5.2.7 دوال رياضية أخرى 324
5.2.8 الخلاصة 324
5.2.9 المهام 325
ا. جمع الأعداد من الزائر 325
ب. لماذا 6.3 == (1)6.35.toFixed؟ 325
ج. كرر حتى يصبح المُدخَل عددًا 326
د. حلقة غير منتهية أحيانًا 327
ه. رقم عشوائي من العدد الأدنى إلى الأقصى 328
و. قيمة صحيحة عشوائية من min إلى max 329
5.3 النوع string: السلاسل النصية 331
5.3.1 علامات الاقتباس "" 331
5.3.2 الرموز الخاصة 332
5.3.3 طول النص 334
5.3.4 الوصول إلى محارف سلسلة 334
5.3.5 النصوص ثابتة 335
5.3.6 تغيير حالة الأحرف الأجنبية 335
5.3.7 البحث عن جزء من النص 336
ا. str.indexOf 336

5. خدعة NOT على مستوى البِت 338
ب. includes, startsWith, endsWith 339
5.3.8 جلب جزء من نص 339
ا. str.slice(start [, end]) 339
ب. str.substring(start [, end]) 340
ج. str.substr(start [, length]) 340
5.3.9 موازنة النصوص 341
ا. str.codePointAt(pos) 342
ب. String.fromCodePoint(code) 342
ج. موازنات صحيحة 343
5.3.10 ما خلف الستار، يونيكود 343
ا. أزواج بديلة Surrogate pairs 343
ب. علامات التشكيل وتوحيد الترميز 345
5.3.11 الخلاصة 346
5.3.12 تمارين 347
ا. حول الحرف الأول إلى حرف كبير 347
ب. فحص وجود شيء مزعج 348
ج. قص النص 348
د. استخراج المال 349
5.4 المصفوفات Arrays 350
5.4.1 التصريح 350
5.4.2 توابِع الدفع والجلب 352
5.4.3 داخليًا وخلف الكواليس 354
5.4.4 الأداء 355
5.4.5 الحلقات 356
5.4.6 كلمتان حول "الطول" 358
5.4.7 new Array() 358
5.4.8 المصفوفات متعدّدة الأبعاد 359
5.4.9 تحويل المصفوفات إلى سلاسل نصية 359

5.4.10 الخلاصة 360
5.4.11 تمارين 361
ا. هل تُنسخ المصفوفات؟ 361
ب. العمليات على المصفوفات 361
ج. النداء داخل سياق المصفوفة 362
د. جمع الأعداد المُدخَلة 363
ه. أكبر مصفوفة فرعية 364
5.5 توابِع المصفوفات 368
5.5.1 إضافة العناصر وإزالتها 368
ا. الوصل splice 368
ب. القطع slice 370
ج. الربط concat 371
5.5.2 التكرار: لكلّ forEach 372
5.5.3 البحث في المصفوفات 372
ا. التوابع indexOf و lastIndexOf و includes 373
ب. البحث عبر find و findIndex 373
ج. الترشيح filter 374
5.5.4 التعديل على عناصر المصفوفات 375
ا. التابع map 375
ب. التابع sort(fn) 376
ج. العكس reverse 378
د. التقسيم split والدمج join 378
ه. التابعان reduce و reduceRight 379
5.5.5 التابع Array.isArray 381
5.5.6 تدعم أغلب التوابع thisArg 382
5.5.7 الخلاصة 383
5.5.8 تمارين 384
ا. حول border-left-width إلى borderLeftWidth 384
ب. نطاق ترشيح 385

ج. نطاق ترشيح "كما هو" 386
د. الفرز بالترتيب التنازلي 387
ه. نسخ المصفوفة وفرزها 388
و. خارطة بالأسماء 388
ز. أنشِئ آلة حاسبة يمكن توسعتها لاحقًا 389
ح. المرور على خاصيات كائن 391
ط. تحويل مصفوفة إلى شكل آخر 392
ي. فرز المستخدمين حسب أعمارهم 394
ك. خلط المصفوفات 395
ل. ما متوسّط الأعمار؟ 398
م. ترشيح العناصر الفريدة في المصفوفة 399
ن. إنشاء كائن من مصفوفة 401
5.6 المُكرَّرات Iterables 402
5.6.1 Symbol.iterator 402
5.6.2 السلاسل النصية مُكرَّرة 405
5.6.3 نداء المُكرِّر جهارةً 405
5.6.4 المُكرَّرات والشبيهات بالمصفوفات 406
5.6.5 التابع Array.from 406
5.6.6 الخلاصة 408
5.7 النوع Map (الخرائط) والنوع Set (الأطقم) 410
5.7.1 الخارطة Map 410
5.7.2 المرور على خارطة 412
5.7.3 Object.entries: صنع خارطة من كائن 413
5.7.4 Object.fromEntries: صنع كائن من خارطة 414
5.7.5 الطقم Set 415
5.7.6 المرور على طقم 416
5.7.7 ملخص 417
5.7.8 تمارين 418
ا. ترشيح العناصر الفريدة في مصفوفة 418

ب. ترشيح الألفاظ المقلوبة 419
ج. مفاتيح مُكرَّرة 421
5.8 النوع WeakMap والنوع WeakSet: الخرائط والأطقم ضعيفة الإشارة 423
5.8.1 WeakMap 424
5.8.2 استعمالاتها: بيانات إضافية 425
5.8.3 استعمالاتها: الخبيئة 426
5.8.4 WeakSet 428
5.8.5 الخلاصة 429
5.8.6 تمارين 430
ا. تخزين رايات "غير مقروءة" 430
ب. تخزين تواريخ القراءة 432
5.9 مفاتيح الكائنات وقيمها ومدخلاتها 433
5.9.1 التوابع keys وvalues وentries 433
5.9.2 تعديل محتوى الكائنات 434
5.9.3 تمارين 435
ا. مجموع الخاصيات 435
ب. عدد الخاصيات 436
5.10 الإسناد بالتفكيك 438
5.10.1 تفكيك المصفوفات 438
ا. عامل البقية "..." 440
ب. القيم المبدئية 440
5.10.2 تكفيك الكائنات 441
ا. نمط البقية "..." 444
5.10.3 تفكيك المتغيرات المتداخلة 445
5.10.4 مُعاملات الدوال الذكية 446
5.10.5 الخلاصة 449
5.10.6 تمارين 449
ا. الإسناد بالتفكيك 449
ب. أكبر راتب 450

5.11 النوع Date: التاريخ والوقت 452
5.11.1 الباني 452
ا. new Date() 452
ب. new Date(milliseconds) 452
ج. new Date(datestring) 453
د. new Date(year, month, date, hours, minutes, seconds, ms) 453
5.11.2 الوصول إلى مكوّنات التاريخ 454
ا. التابع getDay() 454
ب. التابع getTime() 455
ج. التابع GetTimezoneOffset() 455
5.11.3 ضبط مكونات التاريخ 455
5.11.4 التصحيح التلقائي 456
5.11.5 تحويل التاريخ إلى عدد، والفرق بين تاريخين 457
5.11.6 التاريخ الآن 457
5.11.7 قياس الأداء 458
5.11.8 تحليل سلسلة نصية باستعمال Date.parse 461
5.11.9 الخلاصة 462
5.11.10 تمارين 462
ا. إنشاء تاريخ 462
ب. اعرض اسم اليوم من الأسبوع 463
ج. اليوم من الأسبوع في أوروبا 464
د. ما هو التاريخ الذي كان قبل كذا يوم؟ 464
ه. آخر يوم من الشهر كذا؟ 466
و. كم من ثانية مضت اليوم؟ 466
5.11.11 كم من ثانية بقت حتّى الغد؟ 467
ا. تنسيق التاريخ نسبيًا 468
5.12 صيغة JSON وتوابعها 472
5.12.1 JSON.stringify 472
5.12.2 الاستثناءات وتعديل الكائنات: آلة الاستبدال 476

5.12.3 التنسيق: المسافات 479
5.12.4 تابِع "toJSON" مخصّص 480
5.12.5 التابع JSON.parse 481
5.12.6 استعمال آلة الإحياء 483
5.12.7 الخلاصة 484
5.12.8 تمارين 484
ا. تحويل كائن إلى JSON وإعادته كما كان 484
ب. استثناء الإشارات السابقة 485
6. التعامل المتقدم مع الدوال 487
6.1 التعاود Rescursion والمكدس Stack 488
6.1.1 نهجان في التطوير 488
6.1.2 سياق التنفيذ والمكدس 491
ا. pow(2, 3) 491
ب. pow(2, 2) 492
ج. pow(2, 1) 492
د. المخرج 493
6.1.3 مسح الأشجار تعاوديًا 494
6.1.4 بنى التعاود 497
ا. القوائم المترابطة 498
6.1.5 الخلاصة 501
6.1.6 تمارين 502
ا. اجمع كل الأعداد إلى أن تصل للممرّر 502
ب. احسب المضروب 504
ج. أعداد فيبوناتشي 505
د. طباعة قائمة مترابطة 508
ه. طباعة قائمة مترابطة بترتيب معكوس 511
6.2 المعاملات "البقية" ومعامل التوزيع 513
6.2.1 المُعاملات "البقية" ... 513
6.2.2 متغير الوسطاء arguments 514

6.2.3 معامل التوزيع 515
6.2.4 الخلاصة 517
6.3 المنغلقات ومجال المتغيرات 518
6.3.1 أسئلة تحتاج أجوبة 518
6.3.2 البيئات المعجمية 519
ا. تصريحات الدوال Function Declarations 520
ب. البيئات المُعجمية الداخلية والخارجية 521
6.3.3 الدوال المتداخلة 523
6.3.4 البيئات بالتفصيل الممل 526
6.3.5 كُتل الشفرات والحلقات، تعابير الدوال الآنية 530
ا. الجملة الشرطية If 530
ب. حلقة "كرّر طالما" 530
ج. كُتل الشفرات 531
د. تعابير الدوال الآنية IIFE 531
6.3.6 كنس المهملات 533
ا. التحسينات على أرض الواقع 535
6.3.7 تمارين 536
ا. هل العدّادات مستقلة عن بعضها البعض؟ 536
ب. كائن عد 537
ج. دالة في شرط if 538
د. المجموع باستعمال المُنغلِقات 539
ه. الترشيح عبر دالة 539
و. الترشيح حسب حقل الاستمارة 541
ز. جيش عرمرم من الدوال 542
6.4 إفادة "var" القديمة 547
6.4.1 ليس لإفادة "var" نطاقًا كُتليًا 547
6.4.2 تعالج التصريحات باستعمال var عند بدء الدالة 549
6.4.3 الخلاصة 551
6.5 الكائن العمومي Global object 552

6.5.1 استعمالها للترقيع تعدديًا 553
6.5.2 الخلاصة 553
6.6 كائنات الدوال Function object وتعابير الدوال المسمّاة NFE 555
6.6.1 خاصية الاسم name 555
6.6.2 خاصية الطول ""length 556
6.6.3 خاصيات مخصّصة 558
6.6.4 تعابير الدوال المسماة 559
6.6.5 الخلاصة 562
6.6.6 تمارين 563
ا. ضبط قيمة العدّاد وإنقاصها 563
ب. مجموع ما في الأقواس أيًا كان عدد الأقواس 564
6.7 صياغة "الدالة الجديدة" new Function 566
6.7.1 الصياغة 566
6.7.2 المُنغلِقات closures 567
6.7.3 الخلاصة 568
6.8 الجدولة: المُهلة setTimeout والفترة setInterval 569
6.8.1 تابِع تحديد المهلة setTimeout 569
ا. الإلغاء باستعمال clearTimeout 570
6.8.2 setInterval 571
6.8.3 تداخل setTimeout 571
6.8.4 جدولة setTimeout بتأخير "صفر" 574
6.8.5 الخلاصة 575
6.8.6 تمارين 576
ا. اكتب الناتج كلّ ثانية 576
ب. ماذا سيعرض setTimeout؟ 577
6.9 المزخرفات والتمرير: التابعان call و apply 579
6.9.1 خبيئة من خلف الستار 579
6.9.2 استعمال func.call لأخذ السياق 581
6.9.3 استعمال أكثر من وسيط داخل func.apply 584

6.9.4 استعارة التوابع borrowing a method 587
6.9.5 المُزخرِفات decorators وخاصيات الدوال function properties 588
6.9.6 الخلاصة 589
6.9.7 تمارين 589
ا. مُزخرِف تجسّس 589
ب. مُزخرِف تأخير 590
ج. مُزخرف إزالة ارتداد 592
د. مُزخرِف خنق 593
6.10 ربط الدوال Function binding 596
6.10.1 ضياع الأنا (الكلمة المفتاحية this) 596
6.10.2 الحل رقم واحد: نستعمل دالة مغلفة 597
6.10.3 الحل رقم اثنين: ربطة 598
6.10.4 الدوال الجزئية partial functions 600
6.10.5 الدوال الجزئية، بدون السياق 601
6.10.6 الخلاصة 602
6.10.7 تمارين 603
ا. دالة ربط على أنّها تابِع 603
ب. ربطة ثانية 603
ج. خاصية الدالة بعد الربط 604
د. أصلِح هذه الدالة التي يضيع "this" منها 604
ه. استعمال الدوال الجزئية لولوج المستخدم 606
6.11 الحديث عن الدوال السهمية Arrow functions مرة أخرى 608
6.11.1 ليس للدوال السهمية مفهوم this 608
6.11.2 ليس للدوال السهمية "مُعاملات" 610
6.11.3 الخلاصة 610
7. ضبط خاصيات الكائنات 612
7.1 رايات الخاصيات وواصِفاتها 613
7.1.1 رايات الخاصيات 613
7.1.2 منع قابلية التعديل 615

7.1.3 منع قابلية الإحصاء 616
7.1.4 منع قابلية إعادة الضبط 617
7.1.5 التابع Object.defineProperties 618
7.1.6 التابع Object.getOwnPropertyDescriptors 619
7.1.7 إغلاق الكائنات على المستوى العام 619
7.2 جالبات الخاصيات Getters وضابطاتها Setters 621
7.2.1 الجالبات والضابطات 621
7.2.2 واصِفات الوصول Accessor Descriptor 623
7.2.3 الجوالب والضوابط الذكيّة 624
7.2.4 استعمالها لغرض التوافقية 625
8. الوراثة النموذجية 627
8.1 الوراثة النموذجية Prototypal inheritance 628
8.1.1 الخاصية [[Prototype]] 628
8.1.2 كائن النموذج الأولي للقراءة فقط 632
8.1.3 ماذا عن "this"؟ 633
8.1.4 حلقة for..in 635
8.1.5 الخلاصة 637
8.1.6 تمارين 638
ا. العمل مع prototype 638
ب. خوارزمية بحث 638
ج. أين سيحدث التعديل؟ 640
د. لماذا أصابت التخمة كِلا الهامسترين؟ 641
8.2 الوراثة النموذجية بتعمق: F.prototype 644
8.2.1 القيمة الافتراضية للخاصية prototype في الباني 645
8.2.2 الخلاصة 648
8.2.3 تمارين 648
ا. تغيير الخاصية prototype 648
ب. إنشاء كائن جديد من خلال نفس باني لكائنٍ آخر 650
8.3 النماذج الأولية الأصيلة Native prototypes 653

8.3.1 Object.prototype 653
8.3.2 كائنات النماذج الأولية الأخرى المضمّنة في اللغة 654
8.3.3 الأنواع الأولية 656
8.3.4 تغيير كائنات النماذج الأولية الأصيلة 657
8.3.5 الاستعارة من كائنات النماذج الأولية 658
8.3.6 الخلاصة 658
8.3.7 تمارين 659
ا. إضافة التابع "f.defer(ms)" إلى الدوالّ 659
ب. إضافة المُزخرِف "defer()" إلى الدوال 659
8.4 توابِع النماذج الأولية والكائنات بلا proto 661
8.4.1 لمحة تاريخية 662
8.4.2 الكائنات "البسيطة جدًا" 663
8.4.3 الخلاصة 665
8.4.4 تمارين 667
ا. إضافة toString إلى dictionary 667
ب. الفرق بين الاستدعاءات 668
9. الأصناف Classes 670
9.1 صياغة الأصناف الأساسية 671
9.1.1 صياغة الأصناف class 671
9.1.2 ما الصنف أصلًا؟ 672
9.1.3 ليست مجرد تجميل لغوي 673
9.1.4 تعابير الأصناف 675
9.1.5 الجوالِب والضوابِط والاختصارات الأخرى 676
9.1.6 خاصيات الأصناف 678
ا. إنشاء توابع مرتبطة بخاصيات الصنف 678
9.1.7 الخلاصة 680
9.1.8 تمارين 681
ا. أعِد كتابتها لتكون صنفًا 681
9.2 وراثة الأصناف Class inheritance 683

9.2.1 عبارة التوسعة extends 683
9.2.2 إعادة تعريف دالة 685
9.2.3 إعادة تعريف الباني 687
9.2.4 عبارة super: أمور داخلية و [[HomeObject]] 689
ا. [[HomeObject]] 692
ب. التوابع ليست "حرّة" 694
ج. توابِع لا صفات دالية 695
9.2.5 الخلاصة 696
9.2.6 تمارين 696
ا. خطأ في اشتقاق صنف 696
ب. توسعة ساعة 698
ج. الأصناف تُوسّع الكائنات؟ 699
9.3 الخاصيات والتوابع الثابتة 703
9.3.1 الخاصيات الثابتة 705
9.3.2 وراثة الخاصيات والتوابِع الثابتة 705
9.3.3 الخلاصة 708
9.4 الخاصيات والتوابِع الخاصّة والمحمية 709
9.4.1 مثال من الحياة العملية 709
9.4.2 الواجهتان الداخلية والخارجية 710
9.4.3 حماية "مقدار الماء" 711
9.4.4 "القوّة" للقراءة فقط 713
9.4.5 "#حد الماء" خاصة 714
9.4.6 الخلاصة 716
9.5 توسعة الأصناف المضمّنة في اللغة 718
9.5.1 الأنواع المضمّنة لا ترث الثوابت 719
9.6 فحص الأصناف عبر instanceof 721
9.6.1 معامل instanceof 721
9.6.2 التابع Object.prototype.toString للأنواع 724
ا. Symbol.toStringTag 725

9.6.3 الخلاصة 726
9.6.4 تمارين 726
9.6.5 instnaceof غريب عجيب 726
9.7 المخاليط Mixins 727
9.7.1 مثال عن المخاليط 727
9.7.2 مخلوط الأحداث EventMixin 730
9.7.3 الخلاصة 732
10. التعامل مع الأخطاء 733
10.1 التعامل مع الأخطاء: "جرب... التقط" try..catch 734
10.1.1 صياغة try..catch 734
10.1.2 كائن الخطأ Error 737
10.1.3 إسناد "catch" الاختياري 738
10.1.4 استعمال try..catch 738
10.1.5 رمي أخطائنا نحن 739
ا. مُعامل "الرمي" throw 740
10.1.6 إعادة الرمي 742
10.1.7 try..catch..finally 744
ا. المتغيرات في صياغة try..catch..finally محلية 746
ب. بخصوص finally و return 746
ج. بخصوص باني try..finally 747
10.1.8 الالتقاط العمومي 747
10.1.9 الخلاصة 748
ا. تمارين 749
10.1.10 أخيرًا أم الشيفرة؟ 749
10.2 الأخطاء المخصصة وتوسعة صنف Error 752
10.2.1 توسعة Error 752
10.2.2 تعميق الوراثة 755
10.2.3 تغليف الاستثناءات 757
10.2.4 الخلاصة 761

10.2.5 تمارين 761
ا. الوراثة من SyntaxError 761
11. الوعود واللاتزامن/الانتظار 763
11.1 مقدمة إلى ردود النداء callbacks في جافاسكربت 764
11.1.1 رد النداء داخل رد النداء 766
11.1.2 التعامل مع الأخطاء 767
11.1.3 هرم العذابات Pyramid of Doom 768
11.1.4 تمارين 771
ا. دائرة تتحرّك ولها ردّ نداء 771
11.2 الوعود Promise 772
11.2.1 الاستهلاك: عبارات then وcatch وfinally 776
ا. then 776
ب. catch 777
ج. finally 777
11.2.2 تحميل السكربتات: الدالة loadScript 779
11.2.3 تمارين 781
ا. إعادة ... الوعد؟ 781
ب. التأخير باستعمال الوعود 781
ج. صورة دائرة متحركة مع وعد 782
11.3 سَلسلة الوعود Promises chaining 783
11.3.1 إعادة الوعود 785
11.3.2 مثال: loadScript 786
ا. كائنات قابلة للسلسلة Thenables 788
11.3.3 مثال أضخم: fetch 789
11.3.4 الخلاصة 792
11.3.5 تمارين 793
ا. الوعد: then وcatch 793
11.4 التعامل مع الأخطاء then/catch 794
11.4.1 صياغة try..catch الضمنية 795

11.4.2 إعادة الرمي 796
11.4.3 حالات رفض 797
11.4.4 الخلاصة 798
11.4.5 تمارين 799
ا. خطأ في تابِع setTimeout 799
11.5 واجهة الوعود البرمجية 800
11.5.1 Promise.all 800
11.5.2 Promise.allSettled 802
ا. تعويض نقص الدعم 804
11.5.3 Promise.race 804
11.5.4 Promise.resolve/reject 805
ا. Promise.reject 806
11.5.5 الخلاصة 806
11.6 الدوال الواعدة: تحويل الدوال إلى وعود Promisification 807
11.7 المهام السريعة Microtasks مقابل الوعد في تنفيذ المهام لاحقًا 810
11.7.1 طابور المهام السريعة 810
11.7.2 الرفض غير المعالج 811
11.7.3 الخلاصة 812
11.8 اللاتزامن والانتظار async/await 813
11.8.1 الدوال غير المتزامنة 813
11.8.2 Await 814
ا. لن تعمل await في الشيفرة ذات المستوى الأعلى 815
ب. إن await تقبل then 816
ج. توابع صنف غير متزامنة 817
11.8.3 التعامل مع الأخطاء 817
ا. async/await و promise.then/catch 819
ب. تعمل async/await مع Promise.all 819
11.8.4 الخلاصة 819
11.8.5 تمارين 820

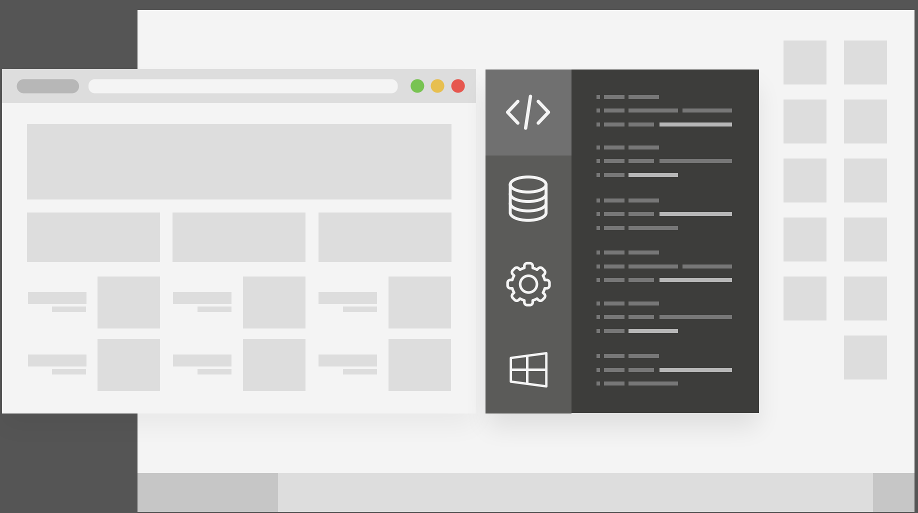
ا. إعادة الكتابة باستعمال async/await 820
ب. أعِد كتابة rethrow باستعمال async/await 821
ج. استدعاء async من دالة غير متزامنة 824
12. المولدات والمكررات المتقدمة 825
12.1 المولدات Generators 826
12.1.1 الدوال المولدة 826
12.1.2 المولدات قابلة للتكرار 828
12.1.3 استعمال المولّدات على أنّها مكرّرات 830
12.1.4 تراكب المولّدات 831
12.1.5 عبارة "yield" باتجاهين اثنين 834
12.1.6 generator.throw 836
12.1.7 الخلاصة 838
12.1.8 تمارين 838
ا. مولّد أرقام شبه عشوائيّة 838
12.2 المكررات والمولدات غير المتزامنة 841
12.2.1 المكررات غير المتزامنة 841
12.2.2 المولدات غير المتزامنة 844
12.2.3 المكررات غير المتزامنة 845
12.2.4 مثال من الحياة العملية 847
12.2.5 الخلاصة 849
13. الوحدات Modules 850
13.1 مقدمة إلى الوحدات Modules في جافاسكربت 851
13.1.1 ما الوحدة؟ 851
13.1.2 ميزات الوحدات الأساسية 852
ا. الوضع الصارم الافتراضي 853
ب. النطاق على مستوى الوحدات 853
ج. تقييم شيفرة الوِحدة لمرة واحدة فقط 854
د. import.meta 856
ه. this في الوِحدات ليست معرّفة 856

13.1.3 الميزات الخاصة بالمتصفّحات 857
ا. سكربتات الوِحدات مؤجلة 857
ب. خاصية Async على السكربتات المضمّنة 858
ج. السكربتات الخارجية 858
د. لا يُسمح بالوحدات المجردة 859
ه. التوافقية باستخدام ""nomodule 859
13.1.4 أدوات البناء 860
13.1.5 الخلاصة 861
13.2 تصدير الوحدات واستيرادها 862
13.2.1 التصدير قبل التصريح 862
13.2.2 التصدير بعيدًا عن التصريح 863
13.2.3 عبارة استيراد كلّ شيء 863
13.2.4 استيراد كذا بالاسم كذا as 864
13.2.5 تصدير كذا بالاسم كذا as 864
13.2.6 التصدير المبدئي 865
ا. الاسم المبدئي 866
13.2.7 إعادة التصدير 868
ا. إعادة تصدير التصديرات المبدئية 870
13.2.8 الخلاصة 871
13.3 استيراد الوحدات ديناميكيًا 873
13.3.1 تعبير الاستيراد 873
14. مواضيع متفرقة 876
14.1 الوسيط Proxy والمنعكس Reflect 877
14.1.1 الوسيط Proxy 877
14.1.2 إضافة اعتراض للقيم المبدئية 879
14.1.3 تدقيق المدخلات باعتراض الضابِط set 881
14.1.4 التكرار باستخدام ownKeys و getOwnPropertyDescriptor 883
14.1.5 الخاصيات المحمية والاعتراض "deleteProperty" وغيره 885
14.1.6 استخدام الاعتراض In range مع has 888

14.1.7 تغليف التوابع باستخدام apply 889
14.1.8 الانعكاس 891
ا. استخدام الوسيط مع الجالِب 893
14.1.9 قيود الوسيط 896
ا. كائنات مضمّنة: فتحات داخلية 896
ب. الخاصيات الخاصة 897
ج. Proxy! = target 898
14.1.10 الوسيط القابل للتعطيل 899
14.1.11 المصادر 901
14.1.12 الخلاصة 901
14.1.13 تمارين 902
ا. خطأ في قراءة الخاصيات غير موجودة في الكائن الأصلي 902
ب. الوصول إلى الدليل [1-] في فهرس المصفوفة 903
ج. المراقب 904
14.2 الدالة "Eval" لتنفيذ الشيفرة البرمجية 907
14.2.1 استخدامات الدالة Eval 908
14.2.2 الخلاصة 909
14.2.3 التمارين 909
ا. آلة حاسبة باستخدام الدالة Eval 909
14.3 تقنية Currying في جافاسكربت 911
14.3.1 لماذا نحتاج لتقنية currying؟ 912
14.3.2 الاستخدام المتقدم لتقنية currying 913
14.3.3 الخلاصة 915
14.4 النوع المرجعي eference 916
14.4.1 شرح النوع المرجعي 917
14.4.2 الخلاصة 918
14.4.3 المهام 918
ا. التحقق من الصياغة 918
ب. اشرح قيمة this 920

14.5 النوع BigInt: الأعداد الكبيرة 921
14.5.1 المعاملات الرياضية 921
14.5.2 عمليات الموازنة 922
14.5.3 العمليات المنطقية 922
14.5.4 ترقيع مشاكل نقص الدعم 923
14.5.5 المصادر 923

دورة علوم الحاسوب



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

تمهيد

لغة جافاسكربت JavaScript هي لغة برمجة شـهيرة موجـودة حيث وجـدت متصـفحات الـويب فهي أحـد الركائز الثلاثية لتطوير الويب مع لغة HTML ولغة CSS فبها مجتمعة تُبنى واجهـات الـويب وبـذلك لا غـنى للغـة جافاسكربت في مجال الويب إذ تساهم في إضفاء التفاعليـة على صـفحات الـويب وبـدونها تصـبح الصـفحات جامـدة، كمـا أنهـا تُنجـز أي عمليـات برمجيـة ومنطقيـة في الصـفحات مثـل التحقـق من مـدخلات المسـتخدم وسلامة البيانات.

اسـتخدام جافاسـكربت لا ينحصـر في المتصـفح بـل يمتـد إلى تطبيقـات أوسـع إذ تُشـغل شـيفرات لغـة جافاسـكربت خـارج المتصـفح في بيئـة Node.js وبـذلك يمكن اسـتخدام لغـة جافاسـكربت في بنـاء مختلـف التطبيقات أشهرها حاليًا بناء خوادم الويب وهنا أصبح بالإمكان استعمال لغـة جافاسـربت في تطـوير الواجهـات الأمامية والواجهات الخلفية لصفحات الويب بمختلف أنواعها والحديث عن اللغة وميزاتها يطـول لـذا لن أتوسـع في التمهيد بالتحدث عن اللغة لأن الكتاب كله عنها وخصوصًا باب المقدمة.

حول الكتاب

هذا الكتاب مـترجم عن الكتـاب The JavaScript Language: Part 1 الجـزء الأول لكاتبـه إيليـا كـانتور Ilya Kantor وفيه يشرح لغة جافاسكربت شرحًا كاملًا لكل ميزاتها وتفاصيلها ويعد أفضل وأول مرجـع تعليمي أجنبي لتعلم لغة جافاسكربت.

ويوجد جزء ثان للكتاب يتحدث عن استعمال جافاسكربت في المتصفحات تحديدًا وجزء ثالث يتحـدث عن مواضيع متفرقة ومتقدمة عن جافاسكربت يكمل فيها الجزأين السـابقين ويتمم الحـديث عن لغـة جافاسـكربت ليكون الكتاب بأجزائه الثلاثة مرجعًا كاملًا عن لغة جافاسكربت.

يُقسم الكتاب إلى 14 فصلًا أو بابًا كل منها مقسوم إلى فصول فرعية أصغر تتحدث عن موضوع محدد من اللغة ولكل قسم مقدمة وخاتمة وتمارين منفصلة، والفصول مرتبة ترتيبًا متدرجًا بدءًا من المواضيع الأساسية وحتى المواضيع المتقدمة بما يناسب المتعلم الجديد الذي يريد تعلم لغة جافاسكربت من الصفر دون خبرة مسبقة، وأما إن كنت تملك خبرة مسبقة بلغة جافاسكربت فيمكنك قراءة الفصول وفق الترتيب الذي تريد والرجوع إليها وفقًا لأبوابه وفصولها الفرعية.

إن قرأت الكتاب كاملًا من البداية وحتى النهاية وطبقت كل التمارين فيه ووعيته بالكامل، فيمكن أن تطلق على نفسك لقب مبرمج جافاسكربت وبجدارة فهذا هو الهدف من الكتاب بأجزائه الثلاثة.

ننصحك للاستفادة القصوى من الكتاب أن تضع توثيق لغة جافاسكربت العربي من موسوعة حسوب في جوارك وأنت تقرأ الكتاب الأمر الذي يساعدك على فهم أجزاء محددة من اللغة والتعرف عليها بعمق.

أضفنا المصطلحات الأجنبية بجانب المصطلحات العربية لسببين، أولهما التعرف على المصطلحات العربية المقابلة للمصطلحات الأجنبية البرمجية الأكثر شيوعًا وعدم الخلط بين أي منها، وثانيًا تأهيلك للاطلاع على المراجع فتصبح محيطًا بعد قراءة الكتاب بالمصطلحات الأجنبية التي تخص لغة جافاسكربت خصوصًا والبرمجة عمومًا وبذلك يمكنك قراءتها وفهمها وربطها بسهولة مع المصطلحات العربية المقابلة والبحث عنها والتوسع فيها إن شئت وأيضًا يسهل عليك قراءة الشيفرات وفهمها. عمومًا، نذكر المصطلح الأجنبي بجانب العربي في أول ذكر له ثم نكمل بالمصطلح العربي، فإذا انتقلت إلى قراءة فصول محددة من الكتاب دون تسلسل، فتذكر إن مررت على أي مصطلح عربي أننا ذكرنا المصطلح الأجنبي المقابل له في موضع سابق.

استخدام الشيفرة

الكتاب مدعوم بشيفرات عملية كثيرة وهي جاهزة للتجريب في المتصفح لذا تجدها تعتمد على دالة المتصفح alert التي تعرض رسالة على شكل مربع منبثق في صفحة المتصفح وبالتالي يمكنك تجريب الشيفرات بنسخها ولصقها في طرفية المتصفح console وسيدلك فصل أدوات المطور على كيفية استخدام هذه الأداة إن لم تكن تعرفها من قبل.

إن كنت تجد صعوبة في نسخ الشيفرات أو تجريبها، يمكنك للسهولة قراءة الكتاب على شكل فصول منشورة على موقع أكاديمية حسوب مباشرةً وتجد روابط الفصول مسردة في صفحة تنزيل الكتاب.

تجد في نهاية كل قسم تمارين ننصح بالتدرّب عليها بعد قراءة القسم قبل الاطلاع على الحل ثم الرجوع إلى الحل وموازنته مع حلك الناتج، وهذا يحقق أقصى استفادة أثناء تعلمك لغة جافاسكربت، وضع في بالك أن حل هذه التمارين يساعدك على تعلم لغة جافاسكربت من جهة وعلى حل الأسئلة البرمجية التي تُسأل في مقابلات العمل لذا لا تهملها.

المساهمة

استغرق العمل على ترجمة هذه الأجزاء ونقلها إلى العربية ما يزيد عن السنتين ونرجو أن نكون قد وفقنا لتوفير مرجع كامل للمبرمج العربي عن أهم لغة برمجية وأشهرها وهي لغة جافاسكربت.

ويرجى إرسال بريد إلكتروني إلى academy@hsoub.com إذا كان لديك اقتراح أو تصحيح على النسخة العربية من الكتاب أو أي ملاحظة حول أي مصطلح من المصطلحات المستعملة. إذا ضمَّنتَ جزءًا من الجملة التي يظهر الخطأ فيها على الأقل، فهذا يسهِّل علينا البحث، ,ويفضل أيضًا إضافة أرقام الصفحات والأقسام.

جميل بيلوني

1. مقدمة

يتضمن هذا الفصل الأقسام التالية:

1. مقدمة إلى لغة JavaScript
2. الأدلة والمواصفات القياسية المعتمدة
3. محررات الشيفرة البرمجية
4. أدوات المطور DevTools

1.1 مقدمة إلى لغة جافاسكربت

تُستخدَم لغة JavaScript (تنطق جَافاسِكْرِبت) لإنشاء صفحات ويب تفاعلية، وهي منتشرة انتشارًا كبيرًا وتُستعمَل في أغلب المواقع، وتدعمها جميع المتصفحات تقريبًا دون الحاجة إلى إضافات خارجية.

يشرح هذا الفصل مزايا لغة JavaScript، وما يمكنك تحقيقه باستخدامها، وأفضل التقنيات التي تعمل معها بشكل جيد.

1.1.1 ما هي لغة جافاسكربت JavaScript؟

ظهرت لغة JavaScript لإنشاء صفحات ويب حيوية وتفاعلية.

تسمى البرامج المكتوبة بلغة JavaScript بالسكربتات (scripts)، ويمكن كتابتها بشكل مباشر ضمن شيفرة HTML لصفحة الويب ليتم تنفيذها تلقائيًا عند تحميل الصفحة. ولا تحتاج هذه السكربتات إلى تحضير خاص أو تصريف مسبق وإنما تتم كتابتها ثم تنفيذها كنص عادي. هذا الأمر يميز لغة JavaScript عن لغة برمجية أخرى تشبهها بالاسم تدعى Java.

لماذا سميت JavaScript؟

عندما ظهرت لغة JavaScript، كان لها اسم مختلف وهو LiveScript (لايف سكربت) ولكن الشعبية الكبيرة للغة جافا Java في تلك الفترة دفعت إلى تغيير اسم اللغة إلى JavaScript بهدف إظهارها بصورة "الأخ الصغير" للغة Java واكتساب بعض الشهرة منها. ومع تطور لغة JavaScript، أصبحت لغةً مستقلة استقلالاً كاملًا ولها مواصفات ومعايير خاصة بها تُسمى ECMAScript، ولم يعد لها أي علاقة بلغة Java.

في الوقت الحالي لا يقتصر تنفيذ لغة JavaScript على المتصفح وإنّما من الممكن تنفيذها ضمن الخادم أيضًا، أو أي جهاز يحتوي على برنامج خاص يسمى محرك JavaScript.

يحتوي المتصفح على مُحرِّك مدمج ضمنه عادةً ما يسمى بآلة JavaScript الافتراضية وله أسماء تختلف باختلاف المتصفحات. فمثلًا يسمى المحرك باسم:

- V8 في متصفحي Chrome و Opera
- Quantum في المتصفح FireFox
- ChakraCore في MicroSoftEdge
- Nitro و SquirrelFish في متصفح Safari

احفظ هذه الأسماء في ذهنك جيدًا لأنَّها مستخدمة بكثرة بين المطورين عبر الإنترنت وسنأتي على ذكرها لاحقًا بين الفصول. إذا قرأت مثلًا الجملة التالية: "الخاصية X مدعومة من قبل V8"، فيجب أن تعرف أنَّ هذه الخاصية تعمل على الأغلب ضمن المتصفحين Opera و Chrome.

كيف تعمل محركات JavaScript؟

طريقة عمل المحركات معقَّدَة وتتألف من ثلاث مراحل أساسية:

- بدايةً، يقرأ المحرك (إذا كان التنفيذ يتم عن طريق المتصفح فالمحرك مدمج ضمنه) أو يفسر (بمعنى أدق السكربت، ثم
- يحول السكربت إلى لغة الآلة (عملية التصريف)، ثم
- تُنفَّذ شيفرات الآلة بشكل سريع.

يحسِّن المحرك كل مرحلة من هذه العملية عن طريق مراقبة السكربت المترجم أثناء تنفيذه، وتحليل تدفق البيانات وفق السكربت ثم يقوم بتحسين شيفرة الآلة الناتجة وفقًا للمعرفة التي اكتسبها مما يساعد على زيادة سرعة تنفيذ السكربت.

1.1.2 ما يمكن أن تفعله جافاسكربت ضمن المتصفح؟

لغة JavaScript بإصدارها الحالي هي لغة برمجية "آمنة". ويعود ذلك إلى توجيه هذه اللغة للعمل ضمن المتصفحات لذلك فهي لا تحتوي على تعليمات للوصول إلى طبقة أدنى من العتاد مثل المعالج أو الذاكرة لأن عملها لا يتطلب ذلك. وتعتمد إمكانيات لغة JavaScript اعتمادًا كبيرًا على البيئة التي تعمل ضمنها؛ فمثلًا تدعم بيئة Node.Js (التي تنفِّذ شيفرة JavaScript خارج بيئة المتصفح) التوابع التي تسمح بالقراءة من أو الكتابة على الملفات، وتنفِّذ طلبات الشبكة، وغيرها من المهام التي لا يمكن تنفيذها في بيئة المتصفح. أمَّا ضمن المتصفح، فيمكن للغة JavaScript القيام بجميع الأعمال المتعلقة بمعالجة صفحات الويب، والتحكم بالتفاعل بين المستخدم وخادم الويب.

يمكن للغة JavaScript ضمن المتصفح تنفيذ العديد من المهام نذكر منها:

- إضافة عناصر HTML جديدة إلى الصفحة، وتغيير المحتوى الحالي لها، وتعديل التنسيقات.
- التفاعل مع المستخدم وتنفيذ أعمال معينة عند النقر على الفأرة وتحريك المؤشر والضغط على أزرار لوحة المفاتيح.
- إرسال الطلبات عبر الإنترنت للخوادم البعيدة، بالإضافة إلى تحميل وتنزيل الملفات (كما في تقنية AJAX و COMET).
- الحصول على معلومات من "ملفات تعريف الارتباط" (Cookies) والتعديل عليها، وطرح الأسئلة على زوار الموقع وإظهار الرسائل والإشعارات.
- يمكن استخدامها لتخزين البيانات من جهة المستخدم أي ذاكرة محلية.

1.1.3 المهام التي لا يمكن لجافاسكربت إنجازها ضمن المتصفح

الحد من إمكانيات JavaScript ضمن المتصفح أمرٌ بالغ الضرورة وذلك لضمان أمان المستخدم. أي أن الهدف من ذلك هو منع صفحات الويب الخطيرة من الوصول إلى معلومات سرية أو التلاعب ببيانات المستخدم ...إلخ.

يوجد العديد من القيود على عمل JavaScript ضمن المتصفح نذكر منها:

أولًا، لا يمكن للغة JavaScript ضمن المتصفح قراءة الملفات الموجودة على القرص الصلب للمستخدم، أو نسخها، أو التعديل عليها. كما لا يمكنها تشغيل البرامج على جهاز المستخدم أو الوصول إلى ملفات نظام التشغيل.

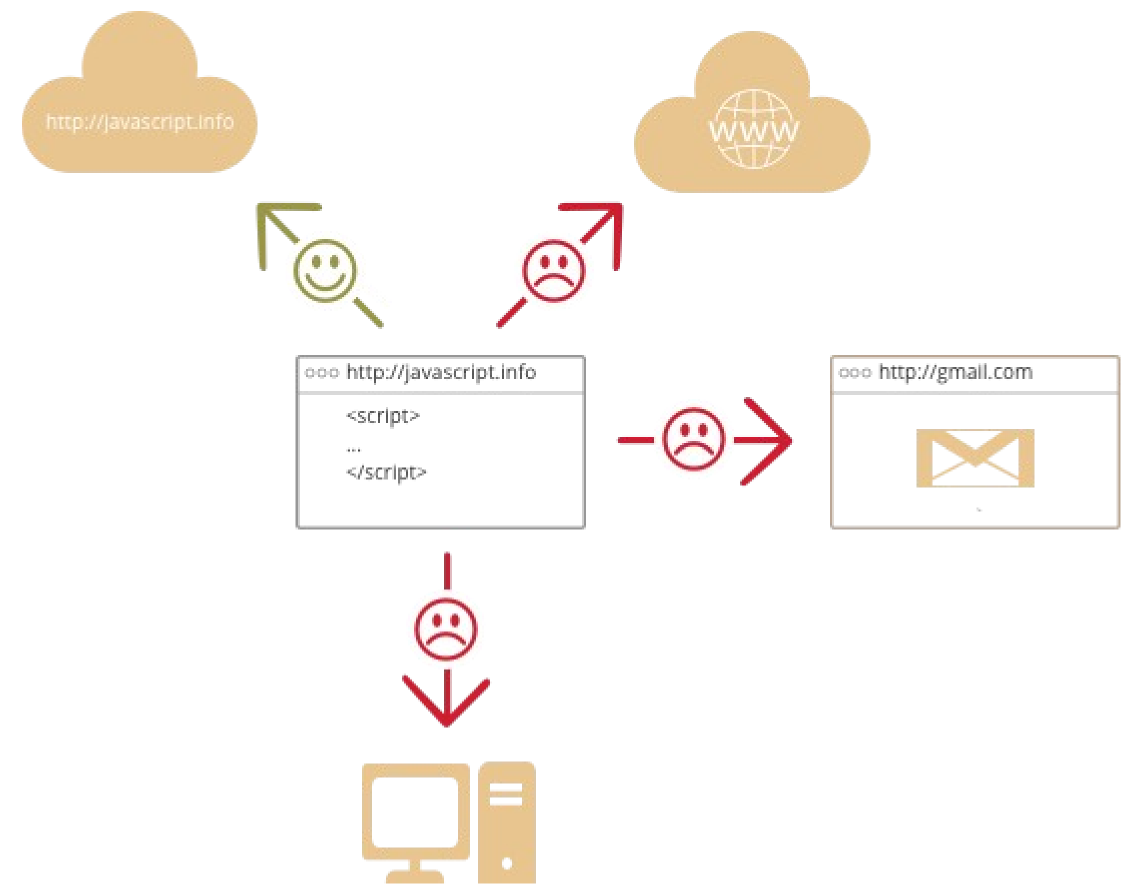
تسمح المتصفحات في الوقت الحالي للغة JavaScript بالتعامل مع الملفات في حالة سماح المستخدم بذلك. مثال على ذلك هو تحميل المستخدم ملفًا ما إلى موقع ويب طلب ذلك إما عبر السحب والإفلات في مربع محدَّد أو اختيار ملف معين في الوسم.

وتوجد طرائق تتيح التعامل مع الأجهزة الملحقة مثل الكاميرا أو مسجل الصوت ولكنها تتطلب تقديم صلاحيات صريحة من قبل المستخدم وأخذ موافقته على أداء مهمة معينة فقط. بمعنى أن الصفحة التي جرى تفعيل JavaScript فيها لا يعني بالضرورة تفعيل كاميرات الويب والوصول إليها الأمر الذي يتيح مراقبة المستخدم ومحيطه وإرسال المعلومات عبر الإنترنت إلى وكالات الأمن القومي كما يظن البعض.

ثانيًا، لا يمكن للألسن والنوافذ المختلفة في المتصفح تبادل البيانات فيما بينها أو معرفة أية تفاصيل عن بعضها بعضًا وهذا في الحالة العامة إلا عندما تستخدم صفحة JavaScript لفتح صفحة أخرى؛ وحتى في هذه الحالة، لا يمكن لصفحة تبادل البيانات مع صفحة فتحتها إذا كانت هاتين الصفحتين من موقعين مختلفين (نطاق مختلف، بروتوكول أو منفذ). يسمى ذلك "سياسة المصدر الواحد" ويمكن تجاوزها عند الحاجة ولكن يجب أن تحتوي كلا الصفحتين على سكربت JavaScript خاص يتحكم بتبادل البيانات بينهما.

هذه المحدودية - مرةً أخرى - ضرورية لضمان أمان المستخدم. فلا يجب على أي صفحة من الموقع http://anysite.com مثلًا والتي فتحها المستخدم أن يكون باستطاعتها الوصول وسرقة بيانات صفحة أخرى من الموقع http://gmail.com مثلًا مفتوحة في لسان آخر من نفس المتصفح.

ثالثًا، بإمكان لغة JavaScript تبادل البيانات عبر الإنترنت بسهولة بين صفحة الويب والخادم الخاص بها ولكن إمكانيتها على استقبال البيانات من مواقع ونطاقات أخرى محدودة إلا في حالة السماح بذلك بتصريح صريح (يذكَر ضمن ترويسة HTTP) للطرف الآخر البعيد. هذه المحدودية مرةً أخرى ضرورية لأمان المستخدم.



يجب التنويه أنَ المحدوديات السابقة توجد فقط عند استخدام لغة JavaScript ضمن المتصفح ولا توجد عند استخدامها ضمن بيئة مختلفة مثل الخادم. كما أنَّ المتصفحات في الوقت الحالي توفر ملحقات/إضافات تسأل المستخدم عن الرغبة بإعطاء صلاحيات أكبر للغة JavaScript المنفذة في المتصفح.

1.1.4 ما المميز في لغة جافاسكربت؟

هنالك على الأقل ثلاثة أمور تميّز لغة JavaScript وهي:

- الاندماج الكامل مع HTML و CSS.
- سهولة تنفيذ الأمور البسيطة.
- مدعومة من قبل أغلبية المتصفحات ومفعلة تلقائيًا.

لغة JavaScript هي اللغة الوحيدة التي تتوفر فيها هذه المزايا الثلاث. وهذا ما يجعلها الأداة الأكثر انتشارًا لبناء صفحات الويب.

قبل البدء بتعلم لغة JavaScript، من المفيد الاطلاع على الآفاق المستقبلية لها والتقنيات الحديثة التي تنافسها من لغات برمجية جديدة وتحديثات في عمل المتصفحات.

1.1.5 اللغات المعتمدة على لغة جافاسكربت

لا تناسب قواعد كتابة لغة JavaScript الجميع. ويختلف المطـورون في حاجـاتهم إلى مزايـا مختلفـة، وهـذا بالطبع أمر متوقع لاختلاف مشاريعهم ومتطلباتهم. لذلك، ظهرت مؤخرًا مجموعة من اللغـات البرمجيـة الجديـدة والتي تُحوَّل transpiled إلى لغة JavaScript قبل تنفيذها في المتصفح.

ومع تطور هذه الأدوات، أصبحت عملية التحويـل (transpilation) سـريعة. الأمـر الـذي سـمح للمطـورين بكتابة الشيفرات بلغة برمجية أخرى ليتم تحويلها تلقائيًا دون أي اختلاف إلى لغة JavaScript وكأنهـا مكتوبـة من الأصل بهذه اللغة.

بعض الأمثلة عن مثل هذه اللغات هي:

- CoffeeScript: تعدُّ لغةً ذات صياغة تجميلية للغة JavaScript. توفر صياغة أقصر ممـا يسـمح بكتابـة شيفرة برمجية أوضح وأكثر دقةً. لها شعبية بين مطوري لغة Ruby.
- TypeScript: تركز على إضافة تعريف لأنواع البيانـات لتسـهيل دعم وتطـوير الأنظمـة المُعقَّدة. طُوِّرت هذه اللغة من قبل شركة Microsoft.
- Dart: هي لغة مستقلة ولها محرك خاص وتعمل على بيئـات مختلفـة غـير المتصـفح (مثـل تطبيقـات الجوال). قدمتها شركة Google بديلًا عن لغة JavaScript، ولكن تعمل المتصفحات في الوقت الحالي على تحويلها إلى لغة JavaScript كما هو حال اللغات السابقة.

هنالك المزيد من اللغات ولكن تعلمك واسـتخدامك لإحـداها لا يلغي ضـرورة تعلمـك للغـة JavaScript، إذ يجب تعلم وفهم هذه اللغة لمعرفة كيفية تنفيذ العمليات والوظائف المختلفة بشـكل فعلي. على أي حـال، تبقى لغة JavaScript هي الأشهر والأقوى والأسرع تطورًا بين اللغات الشبيهة والمنافسة لها.

1.1.6 الخلاصة

أُوجدت لغـة JavaScript في البدايـة كلغـة يتم تنفيـذها فقـط من قبـل المتصـفح ثم تطـورت وأصـبح من الممكن تنفيذها ضمن بيئات مختلفـة عـبر Node.js. أمـا اليـوم، تملـك لغـة JavaScript مكانـة كبـيرة وخاصـة بسبب دعمها من قبل معظم المتصفحات واندماجها الكامل مع HTML و CSS.

هنالك العديد من اللغات التي تُحول إلى JavaScript وتوفر مزايا خاصـة. لـذا ننصـحك بـالاطلاع عليهـا ولـو سريعًا بعد تمكنك من لغة JavaScript.

1.2 الأدلة والمواصفات القياسية المعتمدة

هذا الكتاب هو دورس تعليمية مجزأة تهدف لمساعدتك على تعلم اللغة تدريجيًا. ولكن بمجـرد التعـرف على الأساسيات، ستحتاج إلى مصادر أخرى للتعلم وهو ما سيزودك به هذا الفصل.

1.2.1 المواصفات القياسية Specification

تحتوي مواصفات ECMA-262 على معلومات أكثر تعمقًا وتفصيلًا ورسميةً حـول لغـة جافاسـكربت. والـتي تُحدِّد شكل اللغة.

قد يكون من الصعوبة بمكان قراءة هذه المواصفات الرسـمية في البدايـة. ولكن ولكن وبكـل تأكيـد إذا كنت بحاجة لأفضل مصدر موثوق للمعلومات الـتي تخص تفاصـيل اللغـة، فـإن المواصـفات هي المكـان المناسـب لذلك. ولكن ليست مناسبة للاستخدام اليومي.

تصدر نسخة مواصفات جديدة كل عام. من بين هذه الإصدارات مسودة المواصفات الأخيرة.

لقراءة المزيد عن الميزات الحديثة والجديدة، بما في ذلك الميزات "القياسـية تقريبًـا" (والـتي تسـمى أيضًـا "المرحلة 3")، يمكنك الإطلاع على الاقتراحات على الموجودة هنا.

وإذا كنت بصـدد التطـوير للمتصـفح، فهنـاك أيضًـا مواصـفات أخـرى سنشـرحها في الجـزء الثـاني من هذا الكتاب.

1.2.2 الأدلة Manuals

- مرجع موزيلا للغة جافاسكربت MDN: هو دليل يحتوي على أمثلة ومعلومات أخرى. من الرائع الحصـول على معلومات متعمقة حول الدوال functions والتوابع methods وما إلى ذلك. يمكنك زيـارة الموقـع الرسمي للاطلاع عليها. لمَّا كان من السهل غالبًا أن تبحث على الإنترنت بدلًا من ذلك، فما عليك سـوى استخدام مصطلح "MDN" في خانـة البحث، على سـبيل المثـال للبحث عن الدالـة `parseInt` يكـون الاستعلام هكذا "MDN parseInt".

- توثيـق JavaScript في موسـوعة حسـوب: هـو توثيـق مـترجم عن مرجـع مـوزيلا MDN ومواصـفات جافاسكربت الرسمية (معيار ECMAScript) والمميز فيه أنَّ المصطلحات المسـتعملة هنالـك تتوافـق مع المصطلحات المستعملة في هذا الكتاب وستجد فيه أغلب ما تحتاج إليه من توثيـق جافاسـكربت. يمكنك زيادة موقع موسوعة حسوب واختيار توثيق JavaScript من الصفحة الرئيسية. في حال لم تعثر على شيء ما ضمنه، فيمكنك الرجوع إلى شبكة MDN. أفضل طريقة للبحث ضـمن موسـوعة حسـوب هي عـبر إضـافة الموقـع بعـد الكلمـة المفتاحيـة site أي بالشـكل: site:https://wiki.hsoub.com map javascript

1.2.3 جداول التوافقية مع المتصفحات

تعدُّ لغة جافاسكربت لغةً ناميةً، ودائمًا ما يضاف لها ميزات جديدة بانتظام، ولمعرفة دعمها بين محركات المتصفحات، يمكنك الاطلاع على:

- موقع caniuse : يوفر هذا الموقع جداول الدعم لكل ميزة، فعلى سبيل المثال لمعرفة المحركات التي تدعم وظائف التشفير الحديثة يمكنك الاطلاع على هذه الصفحة.
- المرجع kangax: يوفر هذا المرجع جدول بالميِّزات وجميع المحركات الّتي تدعمها أم لا.

تعد هذه الموارد مفيدة في تطوير المشاريع الحقيقية، لأنَّها تحتوي على معلومات قيمة حول تفاصيل اللغة ودعمها وما إلى ذلك. لذلك حاول تذكر هذه المراجع (أو هذا الفصل) عند مصادفتك للحالات التي تحتاج فيها إلى معلومات متعمِّقة حول ميزة معيَّنة.

1.3 محررات الشيفرة البرمجية

يقضي المبرمجون معظم وقتهم في العمل على أحد محررات الشيفرة البرمجية لكتابة وتطوير برامجهم. سنساعدك في هذا الفصل على اختيار المحرر الأنسب لك.

بدايةً، يوجد نوعان رئيسيان لمحررات الشيفرة البرمجية: "بيئات التطوير المتكاملة" (Integrated Development Environments وتدعى اختصارًا IDE) و "المحررات البسيطة" (lightweight editors). يستخدم العديد من المبرمجين محرِّرًا واحدًا من كل نوع. في بداية تعلمك للبرمجة، يمكنك استعمال المحررات البسيطة مثل المفكرة notepad في ويندوز والمحرر vim أو kate في لينكس ولكن ستحتاج مع تقدمك وتطور وكبر شيفرتك إلى بيئة تطوير متكاملة. فما هي بيئة التطوير المتكاملة؟

1.3.1 بيئة التطوير المتكاملة

بيئة التطوير المتكاملة IDE عبارة عن محرر شيفرة برمجية قوي، ومزود بمزايا كثيرة ومتنوعة تكفي للعمل عادةً على كامل المشروع. وكما هو واضح من اسمها، فهي ليست كأي محرر عادي، ولكن "بيئة تطوير" شاملة.

يمكنك تحميل المشروع بجميع ملفاته إلى بيئة التطوير IDE، مما يسمح بالتنقل بينها بسهولة، كما توفر هذه البيئة IDE ميزة الإكمال التلقائي أثناء كتابتك للشيفرة البرمجية كالتعليمات وأسماء المتغيرات بالنسبة لكامل المشروع (وليس ضمن الملف المفتوح فقط). وتتيح الاندماج مع نظام التحكم في النسخ مثل git، وبيئة الاختبار، وغيرها من الأمور التي تُنفذ على مستوى المشروع.

بإمكانك الاطلاع على خيارات بيئات التطوير IDE التالية، لمساعدتك على اختيار واحدة منها للعمل عليها إن لم تحدد واحدة بعد:

- WebStorm: تستخدم لتطوير الواجهات الأمامية Front-end development. توفر الشركة محررات أخرى للعديد من اللغات البرمجية ولكنها مدفوعة.
- NetBeans بيئة متكاملة ومجانية وتعمل على مختلف أنظمة التشغيل.
- Visual Studio: في نظام التشغيل "ويندوز" (ميِّز بينه وبين Visual Studio Code). وهو محرر قوي ومدفوع وخاص بنظام "ويندوز"، ومناسب جدًا للعمل على منصة ".NET"، ويوجد نسخة مجانية منه تُسمى Visual Studio Community.

أغلب بيئات التطوير المتكاملة متعددة المنصات، وتعمل على أنظمة تشغيل مختلفة ولكنَّها مدفوعة (لها فترة تجريبية مجانية)، وكلفتها متواضعة بالنسبة لدخل المطور المؤهل، لذلك ننصحك باختيار الأفضل والأنسب لك، دون القلق من كلفتها.

1.3.2 المحررات البسيطة

المحررات البسيطة (Lightweight editors) ليست قوية مثل بيئات التطوير المتكاملة، لكنها سريعة، وأنيقة، وبسيطة الاستخدام. تُستخدَم بشكل أساسي لفتح وتعديل الملف بطريقة سريعة وآنية.

الاختلاف الرئيسي بين المحرر البسيط وبيئة التطوير المتكاملة، هو أنَّ بيئة التطوير المتكاملة تعمل على "مستوى المشروع"، لذلك تُحمِّل الكثير من البيانات في البداية، وتكون قادرة على تحليل بنية المشروع عند الحاجة وغيرها من المهام. أمَّا المُحرِّر البسيط، يكون أسرع بشكل كبير عند الحاجة للعمل على ملف واحد فقط من المشروع.

عمليًا، يمكن أن تملك المحررات البسيطة العديد من الإضافات التي تتضمن محللات ومدققات الصياغة على مستوى المشروع وميزة الإكمال التلقائي، وبالتالي لا توجد هناك حدود واضحة بين المحرر الخفيف وبيئة التطوير المتكاملة.

لذلك وقبل أن تحسم رأيك باستخدام بيئة تطوير متكاملة، ننصحك بالاطلاع على الخيارات التالية للمحررات البسيطة:

- Visual Studio Code: مجاني ومتعدد المنصات ويملك العديد من المزايا المشابهة لمزايا بيئة التطوير المتكاملة.
- Atom مجاني الاستخدام ومتعدد المنصات.
- Sublime Text: برنامج تجريبي ومتعدد المنصات ويدعم العديد من اللغات البرمجية (programming languages) واللغات التوصيفية (markup languages).
- Brackets: محرِّر مجاني ومفتوح المصدر وعابر للمنصات ومخصَّص بشكل عام لمطوري الويب وبشكل خاص لمصممي الويب ومطوري الواجهات الأمامية.
- Notepad++: محرِّر مجاني وخاص بنظام التشغيل ويندوز.
- Vim و Emacs: محرِّران بسيطان، ويعدَّان خيارًا جيدًا إذا كنت تعرف كيفية استخدامهما.

1.3.3 ما هو محرري المفضل الذي سأستخدمه؟

نصيحة مني لك هو تجريب الخيارين السابقين كلاهما؛ استخدم بيئة التطوير المتكاملة عند العمل على كامل المشروع، وأحد المُحرِّرات الخفيفة من أجل التعديلات السريعة والطفيفة في ملفات المشروع. يمكن في البداية استعمال المحررات الخفيفة (مثل VS code) التي تفي بالغرض في أغلب المشاريع ثمَّ الانتقال إلى البيئات المتكاملة (سواءً المجانية أو المدفوعة) في المستويات المتقدمة عندما تتولد الحاجة إليها. أسمعك تسألني ما الذي استخدمه؟ حسنًا، أنا استخدم:

- استعمل WebStorm - بيئة التطوير المتكاملة - عند العمل بلغة JavaScript، واستعمل خيـارًا آخـر من JetBrains مثل IntelliJ IDEA عند العمل بلغات برمجية مختلفة.
- واستخدم Sublime أو Atom كمحرر خفيف.

قبل أن تختلف معي بشأن القائمة السابقة، فإن هـذه المحـررات هي إمـا محـررات اسـتخدمتها بنفسـي أو استخدمها أحد أصدقائي من المطورين الجيدين لوقت طويل وكانوا سعداء باستخدامهم لها.

هناك العديد من المحررات الجيدة أيضًا في عالمنا الكبـير. لـذلك، ننصـحك بالعمـل على المحـرر المفضـل لديك. فاختيار المحـرر، مثـل أي أداة أخـرى، هـو قـرار خـاص، ومعتمـد على مشـروعك، وعاداتـك، وتفضـيلاتك الشخصية. ولكي تعرف ما هو المحرر المفضل لديك، لا بد أن تجرب جميع المحرِّرات.

1.4 أدوات المطور

أبقِ في بالك أن الشيفرة البرمجية عرضةً لاحتواء الكثير من الأخطاء، إذ احتمال ارتكابك الأخطاء كبير أثناء كتابة الشيفرة البرمجية، لا بل وقوعك فيها هو أمر حتمي طالما أنك إنسان ولست رجلًا آليًا. هل تصدق أنَّه حتى المبرمجين المتمرسين وذوي الخبرة الطويلة يرتكبون الكثير من الأخطاء في الشيفرة التي يكتبونها! لا تقلق فهذا أمر طبيعي.

على أي حال، لا تُظهِر المتصفِّحات الأخطاء البرمجية تلقائيًا للمستخدمين. وعند وجود مشكلة ما في السكربت، ولا يمكنك آنذاك تحديد مكانها، وبالتالي لا يمكن إصلاحها. لذلك، أُضيفت "أدوات المطوّر" (developer tools) إلى المتصفِّحات لاستكشاف الأخطاء وتوفير معلومات مفيدة عن السكربت لتحسينه.

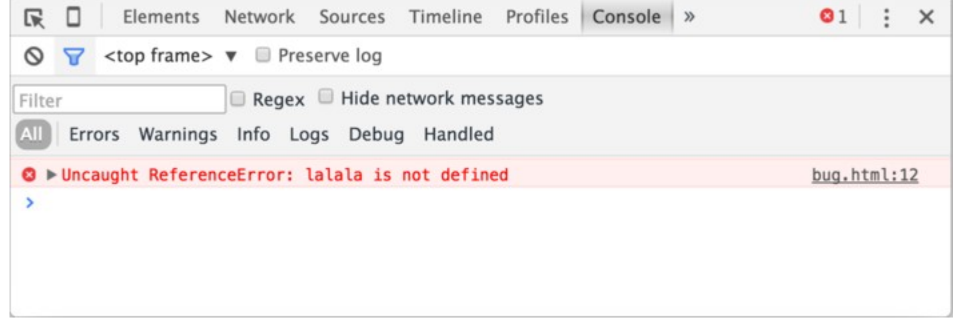
يفضل معظم المطورون العمل على متصفِّحي Chrome أو FireFox لاحتوائهما على أفضل أدوات المطوِّر. توفر المتصفِّحات الأخرى أيضًا مجموعة أدوات للمطوِّر والتي من الممكن أن تحتوي على مزايا خاصة. لكن عادةً ما تحاول اللحاق بمتصفِّحي Chrome و FireFox الأفضل من هذه الناحية. يفضل المطوِّرون بشكل عام العمل على متصفِّح واحد وينتقلون إلى متصفِّح آخر عندما تكون المشكلة التي يعملون عليها محدَّدة بهذا المتصفِّح.

بناءً على ذلك، نجد أنَّ أدوات المطور مهمة للغاية لما تمتلكه من مزايا تساعدك أثناء مسيرتك في تطوير الويب عبر JavaScript. سنتعلم في البداية طريقة فتحها، واستخدامها لاستكشاف الأخطاء، وتشغيل تعليمات JavaScript ضمنها.

1.4.1 أدوات المطور في متصفح Google Chrome

قم بفتح الصفحة bug.html. يوجد خطأ في الشيفرة المكتوبة بلغة JavaScript غير ظاهر للزائر العادي، لذا سنستخدم أدوات المطوّر لاكتشافه.

اضغط على F12 (أو الاختصار Cmd+Opt+J إذا كنت تستخدم نظام التشغيل "ماك") وسيفتح ذلك تلقائيًا أدوات المطوِّر على لسان "الطرفية" (Console). وتظهر أدوات المطوِّر تقريبًا بهذا الشكل:



يعتمد شكل أدوات المطوِّر على إصدار متصفِّح Chrome الذي تستخدمه، إذ يختلف بشكل بسيط من إصدار إلى آخر.

- بإمكانك رؤية رسالة الخطأ باللون الأحمر. معنى الرسالة أن السكربت يحتوي على أمر غير معروف هو "lalala".
- لاحظ في أقصى اليمين وجود رابط إلى المصدر bug.html:12 مع رقم سطر الخطأ في الشيفرة.

يظهر تحت رسالة الخطأ الرمز > باللون الأزرق، ويحدد "سطر الأوامر" (command line) الذي سنكتب عنده أوامر وتعليمات JavaScript. اضغط زر الإدخال Enter لتنفيذ الأمر بعد كتابته (أو Shift+Enter للانتقال إلى السطر التالي عند إدخال أمر متعدد الأسطر).

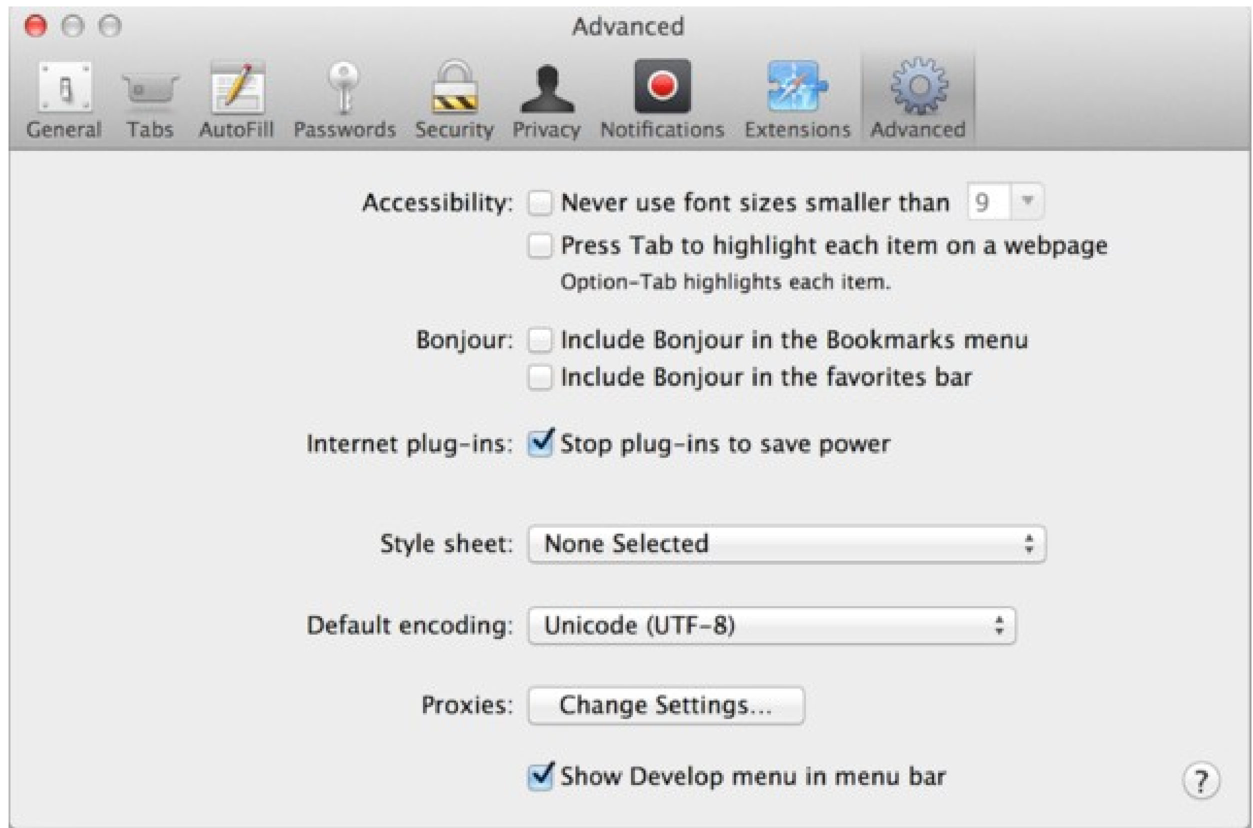
الآن أصبح بإمكانك استكشاف الأخطاء، وهذا يكفي كبداية وسنعود لاحقًا إلى أدوات المطوَر لندرس استكشاف الأخطاء وإصلاحها أو ما يعرف debugging في فصل تنقيح الأخطاء في المتصفِّح Chrome.

1.4.2 أدوات المطور في متصفحي FireFox و Safari وغيرهما

تَستخدِم معظم المتصفِّحات الاختصار F12 لفتح أدوات المطوِّر. تتشابه عمومًا هذه الأدوات في الشكل والمضمون، وبمجرد تعلمك العمل على إحدى هذه الأدوات (بإمكانك البدء مع Chrome)، يمكنك بسهولة الانتقال للعمل على الأدوات الأخرى.

ا. أدوات المطور في متصفح Safari

يختلف المتصفِّح Safari (متصفِّح خاص بنظام التشغيل "ماك" وغير مدعوم من قبل "ويندوز" أو "لينكس") في طريقة فتح أدوات المطور. نحتاج في البداية لتفعيل "قائمة المطوّر" Develop menu. لفعل ذلك، افتح "التفضيلات" Preferences ثم اختر قائمة "متقدم" Advanced. يوجد في الأسفل مربع اختيار لإظهار قائمة المطوِّر في شريط القائمة. حدده لتفعيل القائمة:



بإمكانك استعمال الاختصار Cmd+opt+C لإظهار وإخفاء الطرفية من أدوات المطور. ويمكنك ملاحظة ظهور قائمة جديدة في الأعلى اسمها "Develop". تملك هذه القائمة العديد من الخيارات والأوامر الخاصة بالمطور.

ب. الإدخال متعدد الأسطر

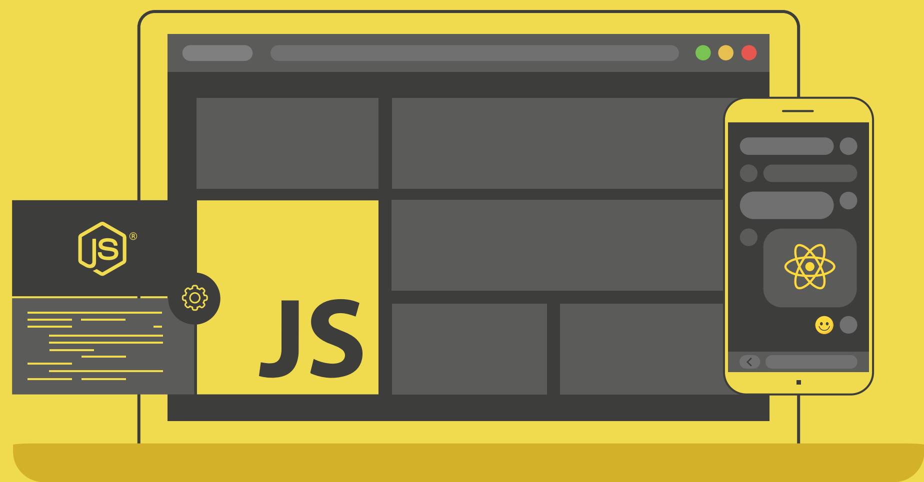
عادةً يتم تنفيذ الشيفرة البرمجية في نافذة الأوامر سطرًا تلو الآخر عند الضغط على زر الإدخال Enter. ولكتابة أمر متعدد الأسطر، اضغط على Shift+Enter (يُمكِّنك ذلك من الانتقال إلى السطر التالي دون تنفيذ السطر الحالي، وعند الانتهاء من إدخال كامل الأمر، اضغط على Enter فقط لتنفيذ هذا الأمر المتعدد الأسطر).

1.4.3 الخلاصة

- توفِّر أدوات المطوِّر لك عدة أدوات متطورة تساعدك على تنقيح الأخطاء، وتنفيذ الأوامر، وفحص المتغيرات، وغيرها من المهام.
- يمكن الوصول إليها باستخدام الاختصار F12 في معظم متصفِّحات نظام Windows. أمَّا في نظام Mac، استخدم الاختصار Cmd+Opt+J لمتصفِّح Chrome، واستخدم Cmd+Opt+C لمتصفِّح Safari (ولكن عليك أن تفعَل قائمة المطوِّر في البداية).

الآن أصبحت البيئة جاهزة لديك، فاستعد للتعمق في لغة JavaScript في الفصل التالي.

دورة تطوير التطبيقات باستخدام لغة JavaScript



احترف تطوير التطبيقات بلغة جافا سكريبت
انطلاقًا من أبسط المفاهيم وحتى بناء تطبيقات حقيقية

التحق بالدورة الآن

2. أساسيات جافاسكريت

يتضمن هذا الفصل الأقسام التالية:

1. المثال الأول: أهلًا بالعالم!
2. بنية الشيفرة البرمجية
3. الوضع الصارم: النمط الحديث لكتابة الشيفرة
4. المتغيرات variables
5. أنواع البيانات Data Types
6. الدوال التفاعلية: alert، prompt، confirm
7. التحويل بين الأنواع
8. العاملات operators
9. عاملات الموازنة
10. العاملات الشرطية
11. العاملات المنطقية
12. عامل الاستبدال اللاغي ??
13. حلقتا التكرار while و for
14. التعليمة switch
15. الدوال في JavaScript
16. تعابير الدوال
17. أساسيات الدوال السهمية
18. مراجعة لما سبق

2.1 المثال الأول: مرحبًا يا عالم!

أبقِ في ذهنك أنَ هذا الكتاب هو عن أساس لغة جافاسكربت JavaScript core المستقلة عن أي منصة. سنتعرف لاحقًا على Node.JS والمنصات التي تستخدمها.

الآن، نحن بحاجة إلى بيئة للعمل وتشغيل السكربتات التي سنكتبها، ويبدو المتصفح خيارًا جيدًا. سنقلّل من استعمال التعليمات التي تعمل فقط على المتصفح (مثل alert) لكي لا يضيع مجهودك بالتركيز عليها خاصةً إن كنت تنوي العمل في بيئة أخرى (مثل Node.JS). وسنركّز على لغة JavaScript ضمن المتصفح في الجزء القادم من هذا الكتاب.

في البداية، سنتعرَّف على طريقة إضافة السكربت إلى صفحة الويب. من أجل البيئات التي تعمل على الخادم (مثل Node.JS)، بإمكانك تنفيذ السكربت عن طريق أمرٍ مثل `node my.js`.

2.1.1 الوسم script

من الممكن تضمين برامج لغة JavaScript في أي جزء من مستند HTML باستخدام الوسم <script>. على سبيل المثال:

```
<!DOCTYPE HTML>
<html>

<body>

<p>Before the script… </p>

<script>
  alert('Hello, world!');
</script>

<p>...After the script.</p>

</body>

</html>
```

يحتوي الوسم `<script>` على شيفرة JavaScript تُنفَّذ تلقائيًا عندما يعالج المتصفح الوسم.

2.1.2 الاستعمال القديم للعنصر script

يملك الوسم <script> عددًا من الخاصيات، ونادرًا ما يتم استخدامها الآن، ولكن ما زال بإمكانك رؤيتها في الشيفرات البرمجية القديمة.

ا. الخاصية type

```
<script type=...>
```

تتطلب معايير لغة HTML القديمة، HTML4، إسناد قيمة للخاصية type في الوسم <script>، وعادةً ما تكون "type="text/javascript، لكنَّ هذا الأمر لم يعد ضروريًا. كما أنَّ معنى هذه الخاصية تغيَّر كاملًا وفق معايير لغة HTML بنسختها الحالية، HTML5. وأصبحت تستخدم مع وحدات (modules) لغة JavaScript. ولكنَّه موضوع متقدِّم سنتحدث عنه لاحقًا في جزء آخر من هذا الكتاب.

ب. الخاصية language

```
<script language=..>
```

تُستخدَم هذه الخاصية لتحديد اللغة المكتوب بها السكربت، ولكنَّها لم تعد مهمة الآن، لأنَّ لغة JavaScript أصبحت هي اللغة الافتراضيَّة، ولم تعد هناك حاجة لاستخدام هذه الخاصيَّة.

ج. التعليقات قبل وبعد السكربتات

من الممكن أن تجد تعليقات ضمن الوسم <script> في كتب ومراجع لغة JavaScript القديمة، مثل:

```
<script type="text/javascript"><!--
  ...
//--></script>
```

تعمل هذه التعليقات على إخفاء الشيفرة البرمجية عن المتصفحات القديمة والتي لا تعرف معالجة الوسم <script> ولم تعد مستخدمة في لغة JavaScript بنسختها الحالية. وبما أن نسخ المتصفحات في السنوات الخمسة عشر الماضية لا تملك هذه المشكلة، فإنَّ هذا النوع من التعليقات يمكِّنك من تحديد الشيفرات البرمجية القديمة للغة JavaScript.

2.1.3 الاستعمال الحديث للعنصر script

عندما تكون الشيفرة البرمجية للغة JavaScript طويلة، قد ترغب في وضعها في ملف مستقل. ويتم ربط ملف السكربت إلى HTML عن طريق الخاصية src:

```
<script src="/path/to/script.js"></script>
```

هنا المسار `/path/to/script.js` هو المسار الكامل لملف السكربت (ابتداءً من جذر الموقع).

بإمكانك أيضًا إدخال المسار نسبةً للصفحة الحالية. مثلًا، المسار `src="script.js"` معناه أنَّ الملف script.js موجود في نفس المجلد الذي توجد فيه الصفحة.

بالإمكان أيضًا استخدام عنوان الموقع كاملًا، كما في المثال التالي:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js"
></script>
```

وإن أردت ربط عدة سكربتات مع نفس الصفحة، فاستخدم الوسم `<script>` لكل سكربت على حدة:

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
```

يمكنك العمل بالقاعدة التالية، "السكربتات البسيطة فقط تُضمَّن مع HTML والسكربتات الأكثر تعقيدًا تُكتَب في ملفات مستقلة". والفائدة من وجود ملف مستقل هي إمكانية تنزيل المتصفح لهذا السكربت مرة واحدة وتخزينه في ذاكرة مؤقتة (cache). وبذلك، تتمكن الصفحات الأخرى المرتبطة بنفس السكربت من طلبه وجلبه من الذاكرة المؤقتة عوضًا عن تنزيله مرةً أخرى. وبذلك يتم فعليًا تنزيله مرة واحدة، مما يقلل حركة البيانات عبر الإنترنت ويُسرِّع تنزيل صفحات الويب ثم معالجتها.

انتبه إلى أنَّه يجري تجاهل السكربت المكتوب ضمن الوسم `<script>` في حال ربطه مع ملف مستقل أي عند إسناد قيمة أو مسار معين للخاصية `src`. بعبارة أخرى، ليس بإمكان الوسم `<script>` أن يُنفِّذ شيفرتين برمجيَّتين في آن واحد: شيفرة قادمة من الخاصية `src` وشيفرة برمجية مكتوبة بداخله. فمثلًا، لن تُنفَّذ التعليمة البرمجية `alert(1)` المكتوبة ضمن الوسم في هذا المثال:

```
<script src="file.js">
  alert(1);  // سيُتجاهل
</script>
```

يجب أن تختار أحد الأمرين: ملف خارجي مستقل للسكربت `<script src="...">`، أو وسم `<script>` عادي يحوي ضمنه على الشيفرة البرمجية المراد تنفيذها.

بالإمكان فصل السكربت في المثال السابق إلى وسمين `<script>` منفصلين ليعمل عملًا صحيحًا:

```
<script src="file.js"></script>
<script>
  alert(1);
</script>
```

2.1.4 الخلاصة

- نستخدم الوسم `<script>` لتضمين الشيفرة البرمجية للغة JavaScript في صفحة الويب.
- لم نعد في الوقت الحالي نستعمل الخاصيتين `type` و `language`.
- يمكن ربط السكربت الموجود في ملف خارجي باستخدام الوسم `<script>`.

هنالك الكثير لتعلمه عن السكربتات الممكن تنفيذها في المتصفح وتأثيرها على صفحات الويب. ولكن أبقِ في ذهنك أن هذا الجزء الكتاب يُركِّز على لغة JavaScript ويجب ألا نشوش أنفسنا بالتركيز على معـايير تنفيـذ السكربتات ضمن كل متصفح بشكل خاص. سنسـتخدم المتصـفح وسـيلةً لتنفيـذ سـكربتات JavaScript لأنـه مناسب في حالتنا للتعلم عن طريق الإنترنت، ولكنه إحدى الطرق الكثيرة لتشغيل JavaScript.

2.1.5 تمارين

أ. إظهار تنبيه

الأهمية: ★★★★★

أنشئ صفحةً تُظهِر الرسالة "I'm JavaScript".

بإمكانك تنفيذها في صفحة تجريبية sandbox، أو على قرصك الصـلب. لا يهم ولكن تأكـد من أنهـا تعمـل بالشكل الصحيح. بعد كتابة شيفرتك، نفِّذها ثم تأكد من الإجابة المرفقة.

الحل:

- عرض توضيحي
- فتح الحل في بيئة تجريبية حية

ب. إظهار تنبيه باستخدام سكربت خارجي

الأهمية: ★★★★★

استخدم الحل في التمرين السابق (إظهار تنبيه)، وعدلـه لاسـتخراج الشـيفرة البرمجيـة المـراد تنفيـذها إلى سكربت خارجي باسم "alert.js" موجود في نفس المجلد. افتح الصفحة وتأكد من عمل التنبيه.

الحل:

الشيفرة البرمجية لمستند HTML:

```html
<!DOCTYPE html>
<html>

<body>

  <script src="alert.js"></script>

</body>

</html>
```

ويحتوي الملف `alert.js` في نفس المجلد على الشيفرة البرمجية التالية:

```javascript
alert("I'm JavaScript!");
```

2.2 بنية الشيفرة البرمجية

في هذا الفصل، سنبدأ بتعلم أساسيات كتابة الشيفرة البرمجية في لغة JavaScript. هل أنت مستعد؟ هيا إذًا لننطلق!

2.2.1 التعليمات البرمجية

التعليمات Statements هي صياغة التراكيب والأوامر التي تنفِّذ المهام والأفعال في السكربت. رأيت سابقًا التعليمة التالية، `alert('Hello, world!')` التي تظهر الرسالة "Hello, world!".

بإمكانك استخدام تعليمات بقدر ما تريد في شيفرتك البرمجية. ويمكن فصل التعليمات البرمجية عن بعضها بالفاصلة المنقوطة `;` .

مثلًا، بإمكاننا فصل "Hello World" إلى تنبيهين منفصلين بالشكل التالي:

```
alert('Hello'); alert('World');
```

تُكتب التعليمات البرمجية عادةً على أسطر منفصلة لتسهيل قراءة الشيفرة البرمجية:

```
alert('Hello');
alert('World');
```

2.2.2 الفاصلة المنقوطة

يمكن حذف الفاصلة المنقوطة من آخر التعابير البرمجية في معظم الحالات عند الانتقال إلى سطر جديد. أي أنَّ الشيفرة البرمجية التالية ستعمل أيضًا عملًا صحيحًا:

```
alert('Hello')
alert('World')
```

تُفسِّر لغة JavaScript الانتقال إلى سطر جديد بفاصلة منقوطة (ضمنيًا). وهذا ما يُسمى (الإضافة التلقائية للفاصلة المنقوطة [Automatic semicolon insertion]).

في معظم الحالات، يتضمن السطر الجديد فاصلةً منقوطةً افتراضيةً في آخره. ولكن هذا لا يشمل جميع الحالات. هناك حالات لا يحوي فيها السطر الجديد بالضرورة فاصلةً منقوطةً افتراضية. إليك مثلًا:

```
alert(3+
1
+2);
```

خرج الشيفرة البرمجية السابقة هو 6 لأنَّ لغة JavaScript لا تضيف الفاصلة المنقوطة في هذه الحالة. فمن البديهي أنَّه عند انتهاء السطر بإشارة "+" يكون التعبير ناقصًا (incomplete expression)، وبالتالي لا يتطلب وجود فاصلة منقوطة. لذلك، تعمل الشيفرة البرمجية كما هو مطلوب.

لا تعتمد دائمًا على JavaScript لإضافة الفاصلة المنقوطة لأنَّه توجد حالات تفشل فيها لغة JavaScript بإضافة الفاصلة المنقوطة عندما تكون مطلوبة. والأخطاء التي تحدث نتيجة هذه الحالات صعبة الإيجاد والإصلاح (ستكتشف ذلك قريبًا خلال الشيفرات التي ستكتبها).

إذا كنت ترغب في الاطلاع على مثال واقعي عن هذه الحالة، إليك الشيفرة البرمجية التالية:

```
[1, 2].foreach(alert)
```

لا تقلق إن كانت هذه الشيفرة صعبة الفهم عليك، فليس هناك حاجة الآن لفهم معنى الأقواس المعقوفة `[]` أو التعبير البرمجي `forEach`، فسندرسهم لاحقًا. ولكن أبقِ في ذهنك أن خرج هذه الشيفرة البرمجية هو إظهار 1 ثم 2.

سنضيف الآن تنبيهًا (alert) قبل الشيفرة البرمجية السابقة دون وضع الفاصلة المنقوطة في نهاية العبارة البرمجية:

```
alert("There will be an error")
[1 ,2].forEach(alert)
```

عند تنفيذ الشيفرة البرمجية آنذاك، سيتم إظهار التنبيه الأول فقط ثم سنحصل على خطأ. تعود الشيفرة البرمجية للعمل بشكل صحيح مرة أخرى عند إضافة الفاصلة المنقوطة بعد التنبيه الأول:

```
alert("All fine now");
[1 ,2].forEach(alert)
```

تظهر لدينا الآن الرسالة "All fine now"، ثم 1 و 2.

الخطأ الذي حدث في الحالة الأولى (حالة عدم إضافة الفاصلة المنقوطة)، سببه أنَّ لغة JavaScript لا تضيف الفاصلة المنقوطة تلقائيًا عند الانتقال إلى سطر جديد في حال وجود الأقواس المعقوفة `[…]` في بداية هذا السطر. وبالتالي لن تُضاف الفاصلة المنقوطة تلقائيًا في الحالة الأولى وسيتم التعامل مع الشيفرة البرمجية على أنَّها تعليمة واحدة أي سيراها المحرك بالشكل التالي:

```
alert("There will be an error")[1, 2].forEach(alert)
```

ولكنَّهما عبارتين برمجيتين منفصلتين وليستا عبارة واحدة، وبالتالي عملية الدمج في هذه الحالة خطأ. من الممكن أن تتكرر هذه الحالة ضمن شروط أخرى.

بناءً على ما سبق، ننصحك بإضافة الفاصلة المنقوطة بين التعابير البرمجيَّة حتى لو قمت بفصلها بأسطر جديدة. وهذه هي القاعدة الأكثر اتباعًا بين مستخدمي JavaScript. لنراجع سويةً ما ذُكر سابقًا، من الممكن الاستغناء عن الفاصلة المنقوطة في معظم الحالات، ولكن من الأفضل إضافتها في آخر العبارة البرمجية - وخاصةً بالنسبة للمبتدئين - تجنبًا للوقوع في أخطاء عصية التنقيح والتصحيح أنت في غنًى عنها.

2.2.3 التعليقات

تصبح البرامج أكثر تعقيدًا مع مرور الوقت، ويكون ضروريًا إضافة التعليقات لشرح عمل الشيفرة البرمجية. يمكن وضع التعليقات في أي مكان ضمن السكربت دون أن تؤثر على تنفيذه، لأنَّ المحرك ببساطة يتجاهل جميع التعليقات.

التعليقات المكتوبة على سطر واحد تبدأ بخطين مائلين (forward slash) بالشكل `//`. ويكون الجزء التالي للخطين المائلين على نفس السطر تعليقًا. ومن الممكن أن يشغل التعليق سطرًا كاملًا أو يأتي التعليق بعد العبارة البرمجية.

إليك المثال التالي الذي يشرح ما سبق:

```
// يمتد هذا التعليق على كامل السطر فقط
alert('Hello');
alert('World'); // هذا تعليق يلي تعبيرًا برمجيًا
```

وإن أردت كتابة تعليقات تمتد على عدَّة أسطر، فابدأ التعليق متعدد الأسطر بخط مائل يليه رمز النجمة (أي `/*`) ، وأنهِ التعليق برمز النجمة ثم الخط المائل (أي `*/`) . إليك المثال التالي:

```
/* يُظهر هذا المثال تنبيهين
وهذا التعليق متعدد
الأسطر
*/
alert('Hello');
alert('World');
```

يجري تجاهل كل ما يقع داخل التعليق متعدد الأسطر، وبالتالي لا تُنفَّذ أية شيفرة برمجية موجودة داخل `/*…*/`. أحيانًا، يكون من المفيد إلغاء تفعيل جزء من الشيفرة البرمجية مؤقتًا أثناء تنقيح الأخطاء:

```
/* تعليق جزء من الشيفرة
alert('Hello');
*/
alert('World');
```

استخدام الاختصارات

من الممكن تعليق سطر واحد من الشيفرة البرمجية بالضغط على الاختصار Ctrl+/ في معظم المُحرِّرات وبيئات التطوير. ومن أجل التعليقات متعددة الأسطر من الممكن استخدام الاختصار Ctrl+Shift+/، (عليك أن تُحدِّد جزءًا من الشيفرة البرمجية ثم الضغط على الاختصار). في الأجهزة التي تعمل على الماك، جرِّب Cmd عوضًا عن Ctrl.

التعليقات المتداخلة متعددة الأسطر غير مدعومة. أي لا يمكنك وضع /*…*/ داخل /*…*/ آخر، إذ ستنتهي هذه الشيفرة البرمجية بخطأ:

```
/*
      /*تعليق متداخل*/
*/
alert('World');
```

لا تتردد أبدًا في وضع التعليقات ضمن شيفرتك البرمجية وشرح ما الذي يجري فيها لأنَّك عندما تعود إليها لاحقًا، ستجد غالبًا أنَّك نسيت وظيفة كل جزء من شيفرتك.

قد تزيد التعليقات من حجم الشيفرة ولكن هذه ليست بمشكلة. هناك العديد من الأدوات التي تُقلِّص الشيفرة البرمجية قبل نشرها على خادم الإنتاج، إذ تَحذِف التعليقات من السكربت نهائيًّا ولا يبقَ لها بذلك أي أثر. في تلك الحالة، لا يكون للتعليقات أي آثار سلبية عند الإنتاج. لا تقلق مرة أخرى إن لم تتضح لك الصورة ولم تفهم بعض المصطلحات (مثل ما الذي يقصد بالإنتاج) فكل شيء سيتضح تدريجيًّا.

سنخصِّص لاحقًا من هذا الكتاب فصلًا يتحدث عن جودة الشيفرة البرمجية والذي يشرح طريقة كتابة التعليقات بأفضل شكل.

2.3 الوضع الصارم: النمط الحديث لكتابة الشيفرات

تطورت لغة JavaScript خلال فترة طويلة دون أي مشاكل في التوافق. وجرى التحقق من استمرار عمل جميع وظائفها في كل مرة تضاف إليها خصائص جديدة.

وكان لذلك فائدة كبيرة في استمرار عمل الشيفرات البرمجية الموجودة دون تعطلها. ولكن كان له ناحية سلبية أيضًا، هي أنَّ الأخطاء والخصائص التي أضافها مطورو اللغة بقيت مع اللغة إلى الأبد.

واستمرت هذا الحال حتى ظهور ECMAScript 5 (اختصارًا ES5) في عام 2009. التي أضافت خصائص جديدة إلى اللغة وعدَّلت على بعض الخصائص الموجودة سابقًا. ولضمان استمرار عمل الشيفرات البرمجية السابقة، فإنَّ معظم هذه التعديلات كانت غير فعَّالة بشكل افتراضي؛ ويجب عليك تفعيلها لاستخدامها صراحةً باستخدام الموجه الخاص: `"use strict"`.

2.3.1 الموجه "use strict"

يبدو الموجه كسلسلة نصية: `"use-strict"` أو `'use strict'`. وعندما يوضع في بداية السكربت، يعمل السكربت حسب النمط والطريقة الحالية.

إليك المثال التالي:

```
"use strict";
//هذه الشيفرة البرمجية تعمل في الوضع الصارم
...
```

قريبًا، سندرس الدوال (طريقة لجمع الأوامر). ولكن في نظرة استباقية للموضوع، لاحظ أنه بالإمكان وضع `"use strict"` في بداية معظم أنواع الدوال عوضًا عن كامل السكربت. وبذلك يتم تفعيل الوضع الصارم (strict mode) ضمن الدوال فقط. لكن عادةً يستخدم المبرمجون هذا الوضع لكامل السكربت.

تأكد من وضع `"use script"` في بداية السكربتات، وإلا فإنه من الممكن ألا يُفعَّل الوضع الصارم. فمثلًا، الوضع الصارم غير مفعّل في المثال التالي:

```
alert("some code");
// سيجري تجاهل الموجه "use strict" إن لم يُستعمَل في أول سطر
"use strict"
// الوضع الصارم غير مُفعَّل
```

بإمكانك كتابة التعليقات فقط قبل استخدام الموجه `"use strict"`.

> ليس هناك طريقة لإلغاء الوضع الصارم `"use strict"`، أي لا يوجد موجه آخر مثل `"no use strict"` والذي يثني المحرك عن عمله ويلغي تفعيل الوضع الصارم، لذلك عند بدء استخدام الوضع الصارم، لا توجد طريقة لإلغائه والعودة إلى الوضع الافتراضي.

2.3.2 طرفية المتصفح

أبقِ في ذهنك أن طرفية المتصفح (console) لا تستخدم الوضع الصارم لاختبار وتنفيذ الشيفرات المكتوبة فيها؛ أي أنها لا تستخدم `"use strict"` افتراضيًا.

عندما يوجد هناك اختلاف بين `"use strict"` والنمط الافتراضي في بعض الأحيان، قد تحصل على ناتج خطأ. ولن تنجح محاولتك في تفعيل الوضع الصارم (strict) بالضغط على الاختصار Shift+Enter لإدخال عدة أسطر، ثم استخدام `"use strict"` في البداية وذلك بسبب طريقة معالجة الطرفية للشيفرة البرمجية داخليًا.

الطريقة الأفضل للتأكد من عمل `"use strict"` هو كتابة الشيفرة البرمجية ضمن الطرفية كما يلي:

```
(function() {
'use strict';
// … ضع شيفرتك هنا …
})()
```

2.3.3 استخدم "use strict" دومًا

سنتعلم لاحقًا الاختلافات بين الوضع الصارم والوضع الافتراضي الذي تعمل ضمنه الشيفرات. ففي الفصل القادم، ستلاحظ الاختلافات بين هذين الوضعين أثناء تعلمك للخصائص الجديدة. من الجيد أنه لا يوجد الكثير من هذه الاختلافات، وفي حال وجودها فهي لتحسين عمل الشيفرة البرمجية. ولكن في الوقت الحالي تكفي معرفتك بهذه الأمور العامة:

1. الموجه `"use strict"` يحوِّل المحرك إلى النمط الحديث (modern)، مما يغير من طريقة تعامله مع بعض الخصائص الموجودة سابقًا. سندرس ذلك بالتفصيل في الأجزاء القادمة من هذا الكتاب.
2. يفعِّل الوضع الصارم بوضع الموجه `"use strict"` في بداية السكربت أو الدالة. العديد من خصائص اللغة، مثل الأصناف (classes) والوحدات (modules)، تُفعِّل الوضع الصارم (strict) تلقائيًا.
3. الوضع الصارم (strict) مدعوم من قبل جميع المتصفحات الحالية.
4. أنصحك دائمًا بوضع `"use strict"` في بداية السكربت. جميع الأمثلة في هذا الكتاب تفترض العمل ضمن الوضع الصارم إلا إذا ذُكِر غير ذلك وهذا في بعض الحالات النادرة.

2.4 المتغيرات Variables

سوف تحتاج، بالتأكيد، في مرحلة ما من عملك على تطوير التطبيقات في لغة JavaScript للتعامل مع المعلومات. فمثلًا، بإمكانك تخيل العمل على التطبيقين التاليين:

1. متجر إلكتروني – يمكن أن تتضمن المعلومات البضائع التي تباع وسلة التسوق.
2. تطبيق محادثة فورية – يمكن أن تتضمن المعلومات، والمستخدمين، والرسائل وغيرها.

إن أردنا تخزين هذه المعلومات، نحتاج إلى شيء يخزِّنها ويحفظها لنا، وهذا ما تمثله المتغيرات تمامًا. فمثل المتغيرات كمثل الأوعية والآنية التي تحفظ وتخزِّن كل ما يوضع فيها.

2.4.1 المتغير

المتغير هو "مَخزَن مُسمَى" (named storage) للبيانات، وهذا المخزن يقع في الذاكرة. بإمكانك استخدام المتغيرات لتخزين البضائع، والزوار وغيرها من البيانات.

إن أردت إنشاء متغير جديد في لغة JavaScript، استخدم الكلمة المفتاحية `let`. تنشئ العبارة البرمجية التالية (أو بمعنى آخر تصرِّح عن، أو تُعرِّف) متغيرًا جديدًا باسم "message":

```
let message;
```

الآن، نستطيع وضع بعض البيانات فيه باستخدام معامل الإسناد `=`:

```
let message;
message='Hello'; //store the string
```

بعد تنفيذ هذه الشيفرة، تُحفَظ السلسلة النصية السابقة في منطقة من الذاكرة مرتبطة بهذا المتغير. ويمكننا الوصول إليها باستخدام اسم المتغير:

```
let message
mesaage = 'Hello!';

alert(message); // اظهار محتوى المتغير
```

وللاختصار، من الممكن الدمج بين التصريح عن المتغير وإسناد قيمة معينة له في نفس السطر بالشكل التالي:

```
let message = 'Hello!'; // التصريح عن المتغير وإسناد قيمة له
alert(message); // Hello!
```

كما من الممكن أيضًا التصريح عن عدة متغيرات في نفس السطر:

```
let user = 'Ahmad', age = 25, message = 'Hello';
```

يبدو هذا أقصر، ولكن لا أنصحك باستخدام هذه الطريقة. استخدم سطرًا مستقلًا لكل متغير لتسهيل قراءة الشيفرة البرمجية.

تبدو المتغيرات المُتعدِّدة الأسطر (المكتوبة على أسطر منفصلة) أطول قليلًا، ولكنَّها أسهل للقراءة:

```
let user = 'Ahmad';
let age =25;
let message = 'Hello';
```

يصرِّح بعض المبرمجين عن عدة متغيرات بأسلوب الأسطر المتعددة بالشكل التالي:

```
let user = 'Ahmad',
   age = 25,
   message = 'Hello';
```

أو حتى من الممكن استخدام أسلوب "الفاصلة أولًا" (comma-first):

```
let user = 'Ahmad'
   , age = 25
   , message = 'Hello';
```

برمجيًا، جميع هذه الحالات متماثلة وظيفيًّا. لذلك، يعود الأمر لك لتختار الأنسب والأفضل.

استخدام var بدلًا من let

في السكربتات القديمة، من الممكن أن تلاحظ استخدام الكلمة المفتاحية `var` بدلًا من `let` للتصريح عن المتغيرات:

```
var message = 'Hello';
```

وللكلمة المفتاحية `var` نفس عمل `let` تقريبًا. فهي تصرح أيضًا عن المتغيرات ولكنها تختلف عنها قليلًا، ومن الممكن أن تعتبرها، الطريقة "التقليدية" (old-school) للتصريح عن المتغيرات.

هناك فروق دقيقة بين `let` و `var`، لكنها حاليًا غير مهمة بالنسبة لك. سندرس هذه الفروق بالتفصيل في فصل "المتغير القديم".

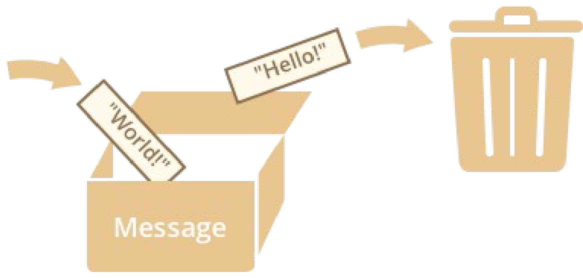
2.4.2 مثال واقعي للمتغيرات

إن أردت فهم مبدأ عمل المتغيرات فهمًا جيدًا، فتخيل "صندوقًا"، وعليه لُصاقَة باسم مميز. فمثلًا، بإمكانك تخيل المتغير message بصندوق مسمى بالاسم `"message"`، وداخل هذا الصندوق القيمة "Hello!":

بإمكانك وضع أي قيمة في الصندوق، كما بإمكانك تغيير هذه القيمة مرارًا وتكرارًا:

```
let message;
message = 'Hello!';
message = 'World!'; // value changed
alert(message);
```

ولكن لاحظ أنه عندما تتغير القيمة، تحذف البيانات السابقة من المتغير:



بإمكانك أيضًا التصريح عن متغيرين ونسخ البيانات من أحدهما إلى الآخر.

```
let hello = 'Hello world!';
let message;
// message نسخ القيمة 'Hello world' من المتغير hello إلى المتغير
message = hello;
// يملك المتغيران الآن البيانات نفسها
alert(hello); //Hello world!
alert(message); // Hello world!
```

يجب أن يُصرَّح عن متغير مرة واحدة فقط، ويؤدي التصريح المتكرر عن متغير إلى خطأ من نوع SyntaxError:

```
let message = "This";

// تصريح متكرر
let message = "That";
// SyntaxError: 'message' has already been declared
```

لذلك، يجب التصريح عن المتغير مرة واحد باستعمال `let` ثم استعمال اسم المتغير دون تلك الكلمة المفتاحية.

اللغات الوظيفية

من الأمور التي تجدر ملاحظتها أيضًا أن اللغات البرمجية الوظيفية، مثل Scala أو Erlang، تمنع تغيير قيم المتغيرات بعد إسنادها إليها. أي عندما تخزِّن القيمة في الصندوق، في مثل تلك اللغات، فإنها تبقى فيه للأبد. وعند الحاجة إلى تخزين بيانات أخرى، تجبرك اللغة على خلق صندوق آخر (التصريح عن متغير جديد)، ولا يمكنك آنذاك إعادة استخدام الصندوق السابق ولا حتى إعادة الصندوق الحالي الذي أنشأته. هل يمكنك تخيل كومة الصناديق التي ستراكم بعد فترة من الزمن؟!

يبدو الأمر غريبًا وصعبًا قليلًا، ولكن فعليًا هذه اللغات قادرة على إجراء تطويرات جادة ومهمة. كما أنَ بعض المجالات تستغل هذه المحدودية وتعتبرها فائدة كما في الحوسبة المتوازية مثلًا. دراسة مثل هذه اللغات جيد لتوسيع مداركك وآفاقك (حتى لو كنت لا تخطط لاستخدامها قريبًا).

2.4.3 تسمية المتغيرات

هنالك شرطان لأسماء المتغيرات في JavaScript:

1. يجب ألا يحتوي اسم المتغير إلا على حروف، و أرقام، و الرمزين $ و _ فقط.
2. يمنع استخدام رقم في أول حرف من الاسم، أي يجب ألا يبدأ اسم المتغير برقم.

أمثلة عن التسمية الصحيحة للمتغيرات:

```
let userName;
let test123;
```

عندما يتألف الاسم من عدة كلمات، يستخدم عادةً أسلوب "سنام الجمل" (camelCase). حيث تكتب الكلمات متتالية دون أي فاصل، وتبدأ كل كلمة بحرف كبير: myVeryLongName.

الأمر المميز الآخر هو أنه من الممكن استخدام رمز الدولار $ والشرطة التحتية _ في أسماء المتغيرات. وهما عبارة عن رمزين عاديين، كأي حرف آخر، وليس لهما معنًى خاص.

أمثلة عن بعض الأسماء المسموحة:

```
let $=1; // "$" التصريح عن متغير اسمه
let _=2; // "_" والتصريح عن متغير آخر اسمه

alert($ + _); // 3
```

وأمثلة عن بعض أسماء المتغيرات المكتوبة خطأً:

```
let 1a; // لا يمكن بدء اسم المتغير برقم
let my-name; //  لا يسمح باستعمال الشرطة '-' في أسماء المتغيرات
```

حالة الأحرف مهمة

المتغير الذي اسمه apple مختلفٌ تمامًا عن المتغير الذي اسمه AppLE.

استخدام أحرف من لغات أخرى غير الإنجليزية

من الممكن استخدام أحرف من أي لغة غير الإنجليزية وحتى الأحرف العربية أو السيريلية أو الهيروغليفية، ولكن لا ننصحك بذلك. فمثلًا:

```
let имя = '...';
let 我 = '...';
let رسالة = '...';
```

برمجيًا، لا توجد أية أخطاء هنا، ومثل هذه الأسماء ممكنة، ولكن هناك تقليد عالمي باستخدام أسماء متغيرات إنجليزية. فحتى لو كنت تكتب سكربت صغيرًا ولكن من الممكن أن يبقى فترة طويلة ويصل إلى مبرمجين من بلدان أخرى في وقت ما.

الأسماء المحجوزة

هناك قائمة من الكلمات المحجوزة، والتي ليس ممكنًا استخدامها كأسماء للمتغيرات لأنها مستخدمة من قبل اللغة نفسها. مثلًا، الكلمات التالية: `let`، و `class`، و `return`، و `function` محجوزة. وسينتج الشيفرة البرمجية التالية عنها خطأ في الصياغة:

```
let let = 5; // خطأ
let return = 5;// خطأ
```

الإسناد بعيدًا عن الوضع الصارم

نحتاج عادةً إلى التصريح عن متغير قبل استخدامه. ولكن سابقًا، كان من المسموح برمجيًا خلق المتغير فقط بإسناد قيمة له دون استخدام `let` أو أي كلمة مفتاحية أخرى. ما زال هذا ممكنًا، ولكن بشرط عدم استخدام الموجه `"use strict"` في السكربت وذلك لضمان استمرارية التوافق مع السكربتات السابقة.

```
// هنا "use strict" انتبه إلى عدم استخدام الموجه
num = 5; // إن لم يكن موجودًا "num" سيُنشأ المتغير
alert(num); // 5
```

ولكن أبقِ في ذهنك أنها طريقة سيئة، وسينتج عنها خطأ عند عملك ضمن الوضع الصارم (strict mode):

```
"use strict"
num = 5; // خطأ، لم يُصرَّح عن المتغير
```

2.4.4 الثوابت

للتصريح عن ثابت، وهو المتغير الذي لا تتبدل قيمته، استخدم const بدلًا من let:

```
const myBirthday = '18.04.1982';
```

المتغيرات التي يُصرَّح عنها باستخدام const تُسمَّى "الثوابت" ولا يمكن أن تتغير قيمتها؛ وأي محاولة للقيام بذلك ينتج عنها خطأ عند تنفيذ الشيفرة:

```
const myBirthday = '18.0401982';
myBirthday = '01.01.2001'; // error. can't reassign the constant!
```

يستخدم المبرمج الثابت عندما يكون متأكدًا أن قيمة المتغير لن تتبدل أو لا يجب أن تتبدل ضمن البرنامج، ويصرح عنه باستخدام const. أمَّا الفائدة من ذلك فهي ضمان والتحقق من وصول هذه الحقيقة إلى الجميع.

ا. متى تستخدم الأحرف الكبيرة في تسمية الثوابت؟

هنالك عرف منتشر في استخدام الثوابت كأسماء مستعارة أو مرادفات للقيم صعبة الحفظ أو القيم العددية الثابتة المعروفة قبل التنفيذ. تسمى مثل هذه الثوابت باستخدام الأحرف الكبيرة والشرطة السفلية (_) كما في هذا المثال:

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// .. عندما نريد انتقاء لون
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

والفائدة من ذلك:

- حفظ وتَذكُّر COLOR_ORANGE أسهل بكثير من "#FF7F00".
- احتمال كتابة "#FF7F00" بشكل خطأ أكبر بكثير من كتابة COLOR_ORANGE.
- قراءة الاسم COLOR_ORANGE في الشيفرة البرمجية له معنًى جلي خلافًا لقراءة #FF7F00.

ربما تتساءل الآن وأنت في حيرة، متى تستخدم الأحرف الكبيرة لتسمية الثوابت ومتى نسميها بالشكل الاعتيادي؟ حسنًا، سنوضح ذلك الآن.

عندما تصرِّح عن ثابت، فهذا يعني أن قيمة هذا المتغير لن تتبدل أبدًا. ولكن مـيز بين نـوعين من الثـوابت: الأول هو الثوابت التي تكون قيمتها معروفة قبل التنفيذ (كما في المثال السابق، فالقيمة الســت عشـرية للـون الأحمر معروفة سابقًا وثابتة). أما النوع الثاني، فهو الثوابت التي تحســب قيمتهـا أثنـاء عمـل السـكربت، وخلال التنفيذ، ولكنها لا تتغير بعد إسناد قيمة لها.

مثال على ذلك:

```
const pageLoadTime = /* الوقت الذي تستغرقه الصفحة للتحميل */;
```

قيمة الثابت pageLoadTime تكون غير معروفة قبل تحميل الصـفحة، لـذلك يتم تسـمية الثـابت بشـكل عادي. ولكنه ثابت لأنَّ قيمته لا تتغير بعد إسنادها له. بمعنى آخر، الثوابت المسماة بأحرف كبيرة تُسـتخدَم فقـط كمرادفات للقيم صعبة الكتابة في الشيفرة البرمجية (hard-coded).

2.4.5 تسمية الأشياء تسمية صحيحة

بما أننا نتكلم عن المتغيرات، لابد أن نذكر أمـرًا مهمًّا للغايـة متعلـق بتسـمية المتغـيرات. أرجـوك أن تختـار أسماء منطقيـة للمتغـيرات، وأن تأخـذ الـوقت الكـافي للتفكـير بالأسـماء المناسـبة لهـا. لا تُعـدُّ مهمـة تسـمية المتغيرات مهمةً بسيطةً، فهي واحدة من المهارات الأكثر أهمية وتعقيـدًا في البرمجـة وبإمكانـك بنظـرة سـريعة على أسماء المتغيرات المستخدمة في الشيفرة البرمجية معرفة مدى خبرة المطور الذي كتبه.

في الواقع، وعند العمل على أي مشروع، فإن أغلب الوقت يمضـي في تعـديل وتوسـيع الشـيفرة البرمجيـة الموجودة بدلًا من كتابة شيفرة برمجية جديدة من الصفر. لذلك، عندما تعـود إلى شـيفرتك البرمجيـة بعـد القيـام بأشياء أخرى لفترة، يكون سهلًا بالنسبة إليك إيجاد المعلومات المسماة بشكل جيد. أو بمعنى آخر، عنـدما تملـك المتغيرات أسماء جيدة. رجـائي لـك، مـرةً أخـرى، أن تمضـي بعض الـوقت في التفكـير في الأسـماء الصـحيحة للمتغيرات قبل التصريح عنها. القيام بذلك سيعود عليك بالنفع بشكل رائع.

وإليك بعض النصائح الجيدة لاتباعها في تسمية المتغيرات:

- استخدم أسماء يمكن للآخرين قراءتها وفهمها مثل userName أو shoppingCart.
- ابتعد عن الاختصارات أو الأسماء القصيرة مثل a، أو b، أو c إلا إذا كان العمل واضحًا لك بشكل جيد.
- اجعل أسماء المتغيرات تصف محتواها ولكن لا تبالغ كثيرًا، بل حاول أن تكـون مختصـرة أيضًـا. ومثـال عن الأسماء السيئة data و value. مثل هـذه الأسـماء لا تعـبر عن أي شـيء. ومن الممكن اسـتخدامها فقـط إذا كـان سـياق الشـيفرة البرمجيـة، يوضـح بشـكل بـديهي واسـتثنائي، أي بيانـات أو قيم يشـير إليها المتغير.

- اتفق على شروط معينة ضمن فريقك أو حتى مع نفسك في تسمية المتغيرات. فإذا كان اسم زائر الموقع هو user، فإن اسم المتغيرات المرتبطة به تكون أسماؤها `currentUser` أو `newUser` عوضًا عن `currentVisitor` أو `NewManInTown`.

يبدو ذلك بسيطًا، أليس كذلك؟ بالتأكيد، إنها عملية بسيطة. ولكن اختيار أسماء جيدة ومختصرة ليس بالأمر السهل في الحياة العملية، ولكن ننصحك بإيلاء أسماء المتغيرات أهمية كبيرة واختيارها بعناية مطلقة.

هل تعيد استخدام المتغير أم تخلق متغيرًا جديدًا؟

هناك بعض المبرمجين الكسالى الذين يفضلون إعادة استخدام المتغيرات الموجودة عوضًا عن خلق متغيرات جديدة. وبالنتيجة، تصبح متغيراتهم مثل الصناديق التي يرمي فيها الناس أشياء مختلفة من دون تغيير مسمياتهم. ماذا يوجد الآن داخل الصندوق؟ لا أحد يعلم، ونحتاج إلى الاقتراب والتحقق من محتواه. قد يوفر هؤلاء المبرمجون بعض الوقت لعدم تصريحهم عن متغيرات جديدة. ولكنهم يخسرون عشرة أضعاف هذا الوقت عند استكشاف الأخطاء.

المتغير الإضافي هو أمر جيد وليس سيء. وتعمل "مُصغِّرات" (minifiers) لغة JavaScript الحالية والمتصفح على تحسين الشيفرة البرمجية بشكل جيد (أي ضغطها وتصغير حجمها)، وذلك لتفادي أية مشاكل في أدائها. كما أن استخدام المتغيرات المختلفة للقيم المختلفة يساعد المحرك أيضًا في تحسين شيفرتك البرمجية.

2.4.6 الخلاصة

بإمكاننا التصريح عن المتغيرات لتخزين البيانات باستخدام الكلمات المفتاحية `var`، أو `let`، أو `const`.

- `let`: تستخدم للتصريح عن المتغيرات في النسخة الحالية. كما يجب استخدام الوضع الصارم (strict mode) لاستخدام `let` في V8.
- `var`: هي الطريقة التقليدية للتصريح عن المتغيرات. عادةً، لا نستخدم هذه الطريقة على الإطلاق، ولكن سنتعرف على الفروقات الدقيقة بينها وبين `let` في فصل المتغير القديم `var`، وهذا أمر ضروري في حال اضطررت لاستخدامهم.
- `const`: مثل `let`، ولكن قيمة المتغير لا يمكن تعديلها. كما يجب تسمية المتغيرات بطريقة تسمح لنا بفهم القيم المسندة إليها بسهولة.

2.4.7 تمارين

ا. العمل مع المتغيرات

الأهمية: ★★☆☆☆

1. صرِّح عن المتغيرين: `admin` و `name`.
2. أسند القيمة "Ahmad" إلى المتغير `name`.

3. انسخ قيمة `name` إلى المتغير `admin`.

4. اعرض قيمة المتغير `admin` باستخدام التنبيه alert (يجب أن يكون الخرج "Ahmad").

الحل:

في الشيفرة البرمجية التالية، كل سطر يمثل أحد الأمور المطلوبة في قائمة المهام:

```
let admin, name; // يمكننا التصريح عن متغيرين في الوقت نفسه
name = "Ahmad";
admin= name;
alert (admin); // "Ahmad"
```

ب. اختيار الأسماء المناسبة

الأهمية: ★★★☆☆

1. أنشئ متغيرًا وليكن اسمه كوكبنا (our planet). كيف يمكنك تسمية هذا المتغير؟

2. أنشئ متغيرًا لتخزين اسم الزائر الحالي للموقع. كيف بإمكانك اختيار اسم هذا المتغير؟

الحل:

أولًا: المتغير باسم كوكبنا، هذا أمر سهل:

```
let ourPlanetName = "Earth";
```

لاحظ أنه بالإمكان استخدام اسم أقصر من ذلك مثل planet، ولكن ذلك قد لا يكون واضحًا بالشكل الكافي، أي كوكب نقصد؟ لذلك من الجيد أن تكون أكثر تفصيلًا. على الأقل عندما يكون اسم المتغير ليس طويلًا `isNotTooLong`.

ثانيًا: اسم المتغير الذي يحوي بيانات الزائر الحالي للموقع:

```
let currnetUserName = "Ahmad";
```

مرةً أخرى، بالإمكان اختصار هذا الاسم إلى username، إذا كنت متأكدًا أن هذا الزائر هو الزائر الحالي. المحرِّرات الحالية فيها ميزة الإكمال التلقائي والتي تجعل من السهل كتابة أسماء المتغيرات الطويلة. لا تختصر في اختيار أسماء المتغيرات، فأي اسم يتألف من 3 كلمات هو أمر عادي ومقبول. وإذا كانت ميزة الإكمال التلقائي في محررك سيئة أو غير مناسبة، لا تتردد في تغييره.

ج. الثوابت بالأحرف الكبيرة؟

الأهمية: ★★★★☆

تفحص الشيفرة البرمجية التالية:

```
const birthday = '18.04.1982';
const age = someCode(birthday);
```

هنا لدينا الثابت birthday الذي يحوي تاريخًا والثابت age الذي سيُحسَب من birthday عن طريق الشيفرة البرمجية someCode (لم يتم كتابتها للاختصار، ولأن هذه التفاصيل غير مهمة الآن).

هل يكون صحيحًا استخدام الأحرف الكبيرة لتسمية الثابت birthday أو الثابت age أو كلاهما؟

```
const BIRTHDAY   = '18.04.1982'; // make uppercase?
const AGE = someCode(BIRTHDAY); // make uppercase?
```

الحل:

نستخدم عادةً الأحرف الكبيرة لتسمية الثوابت كمرادفات للقيم صعبة الكتابة في الشيفرة البرمجية (hard-coded). أو بمعنى آخر، عندما تكون قيمة الثابت معروفة قبل التنفيذ ومكتوبة بشكل مباشر في الشيفرة البرمجية.

في هذه الشيفرة البرمجية، يمثل الثابت birthday هذه الحالة تمامًا. لذلك نستطيع تسميته بأحرف كبيرة. وبشكل معاكس، فإن الثابت age يُحسَب أثناء عمل السكربت، اليوم يكون لنا عمر معين ولكن يختلف عمرنا في السنة القادمة، هو ثابت من ناحية عدم تغير قيمته أثناء تنفيذ الشيفرة البرمجية ولكنه أقل ثباتًا من الثابت birthday. وبما أننا نحسب قيمته، لذلك يفضل إبقاء اسمه بالأحرف الصغيرة.

2.5 أنواع البيانات Data Types

يمكن أن يحتوي المتغير على أي نوع من أنواع البيانات في JavaScript. أي من الممكن أن يكون متغير من نوع سلسلة نصية في وقت ما، ثم يتغير محتواه إلى قيمة عددية وهكذا دواليك.

```
// لا يوجد أي خطأ
let message = "hello";
message = 123456;
```

تسمى اللغات البرمجية التي تسمح بتغيير نوع القيم المسندة إلى المتغير بلغات برمجة "ديناميكية النوع" (dynamically typed)، ومعنى ذلك أنه توجد أنواع للبيانات ولكن لا يتم ربط المتغير بنوع معين منها.

هنالك سبعة أنواع أساسية في لغة JavaScript. سنذكرها الآن بشكل عام، وسنتحدث في الفصول القادمة عن كل نوع منها بالتفصيل.

2.5.1 النوع number: الأعداد

```
let n=123;
n=12.345;
```

يمكن تمثيل الأعداد، بما فيها الأعداد الصحيحة (integers) والعشرية (floating point)، في JavaScript عبر النوع `number`.

هنالك العديد من العمليات التي يمكن تنفيذها على المتغيرات العددية، مثل، الضرب `*`، والقسمة `/`، والجمع `+`، والطرح `-`، وغيرها من العمليات الرياضية.

كما أن هناك "قيم عددية خاصة"، بالإضافة إلى الأعداد العادية، والتي تنتمي أيضًا إلى هذا النوع من البيانات مثل: `Infinity`، و `-Infinity`، و `NaN`. وهاك شرحها.

تمثل `Infinity` قيمة اللانهاية الرياضية، وهي قيمة خاصة أكبر من أي عدد موجب آخر. ومن الممكن أن نحصل عليها في عدة حالات منها قسمة عدد على الصفر:

```
alert (1/0); // Infinity
```

أو بالإمكان الإشارة إليها بشكل مباشر:

```
alert (Infinity); // Infinity
```

أما `NaN` فهي قيمة ناتجة عن اختصار العبارة "Not a Number" (ليس عددًا) وتمثل خطأً حسابيًّا، أو حالة عدم تعيين. وهي نتيجة عملية رياضية خاطئة أو غير معروفة. إليك المثال التالي:

```
alert( "not a number"/2 ); // NaN
```

وتتصف القيمة NaN بأنها "لاصقة" (sticky)، أي بمعنى عندما ُتنفَّذ أي عملية على NaN، فالقيمة الناتجة هي NaN أيضًا.

```
alert ("not a number" / 2+5); // NaN
```

لذلك عند وجود القيمة NaN في أي مكان من التعابير الرياضية، فستطغى على كامل النتيجة.

وتنتمي القيم العددية الخاصة شكليًا فقط إلى نوع القيم العددية، لأنَّها في واقع الأمر لا تعبر عن أعداد بمفهومها الشائع. سنتكلم لاحقًا عن التعامل مع الأعداد في فصل (الأعداد).

العمليات الرياضية "آمنة"

ممارسة الرياضيات آمنة في JavaScript، وبإمكانك القيام بأي عملية حسابية، مثل: القسمة على صفر، والتعامل مع السلاسل الغير عددية على أنَّها أعداد، وغيرها من العمليات. ثق تمامًا أن السكربت لن ينتهي بخطأ فادح (أو يتوقف عن العمل). أسوأ ما في الأمر أنك ستحصل على النتيجة NaN (وهذه هي أحد ميزات JavaScript).

2.5.2 النوع BigInt: الأعداد الكبيرة

لا يمكن في JavaScript تمثيل أعداد كبيرة موجبة أكبر من $2^{53} - 1$ (أي 9007199254740991) ولا أعداد سالبة أصغر من $-(2^{53} - 1)$ وهذا عائد إلى قيود من داخل اللغة نفسها، والحقيقة أن ذلك المجال من الأعداد كافٍ في أغلب الحالات ولكن ماذا لو احتجنا في حالات أخرى إلى أعداد كبيرة مثل حالات التشفير والتعمية أو تمثيل تواريخ بدقة كبيرة تصل إلى المايكرو ثانية؟ هنا جاء النوع BigInt الذي أضيف حديثًا إلى اللغة ليمثل الأعداد الكبيرة، ويمكن إنشاء عدد كبير من هذا النوع بإضافة اللاحقة n إلى نهايته:

```
// BigInt يشير المحرف n إلى أن العدد هو من النوع
const bigInt = 1234567890123456789012345678901234567890n;
```

بما أن الأعداد الكبيرة نادرة الاستخدام، فلن نتوسع فيها هنا ولكن خصصنا لها فصلًا كاملًا باسم "النوع BigInt: الأعداد الكبيرة" يمكنك الرجوع إليه متى ما احتجت إلى استعمالها.

مشاكل في التوافقية

النوع BigInt مدعوم إلى الآن في متصفحات فيرفوكس Firfox وكروم Chrome ومتصفح Edge وسفاري Safari باستثناء المتصفح IE إنترنت إكسبلور، ويمكنك تفقد قسم التوافقية في توثيق MDN لمعرفة أي إصدارات تدعم هذه الميزة.

2.5.3 النوع string: السلاسل النصية (النصوص)

تمثَّل النصوص (سنطلق عليها "سلاسل نصية" من الآن وصاعدًا) عبر النوع string. يجب أن تُحاط السلسلة النصية في JavaScript بإشارتي اقتباس.

```
let str = "Hello";
let str2 = 'Single quotes are ok too';
let phrase = `can embed ${str}`;
```

يوجد في لغة JavaScript ثلاثة أنواع من إشارات الاقتباس وهي:

1. المزدوجة: مثل "Hello".
2. المفردة: مثل 'Hello'.
3. المائلة: مثل `Hello`.

إشارتي الاقتباس المفرد والمزدوج هما عبارة عن إشارات اقتباس تتصف بأنها "بسيطة" ولا يوجد أي فرق بينها في JavaScript. أما إشارة الاقتباس المائلة، فهي عبارة عن إشارات اقتباس لها "وظيفة إضافية"، إذ تسمح لك بإضافة متغيرات وتعابير إلى السلسلة النصية بعد إحاطتها بـ `${…}`. فمثلًا:

```
let name = "أحمد";

// تضمين متغير
alert (`مرحبًا، ${name}!`); // مرحبًا، أحمد!

// تضمين تعبير
alert (`the result is ${1+2}`); // النتيجة هي 3
```

يُحسب التعبير داخل `${…}` وتصبح نتيجته جزءًا من السلسلة النصية. بإمكاننا وضع أي شيء هناك مثل: المتغير `name` أو التعبير الرياضي `1 + 2` أو حتى تعابير برمجية أخرى أكثر تعقيدًا.

ولكن أبقِ في ذهنك أن هذا مقتصر فقط على إشارات الاقتباس المائلة. ولا تستطيع إشارات الاقتباس الأخرى القيام بمثل هذا العمل!

```
alert (" the result is ${1+2}" ); // النتيجة هي ${1+2}
                                  // فلا تفعل إشارات الاقتباس المزدوجة أي شيء
```

إذا كنت تشعر بصعوبة الأمر، لا تقلق. سنتحدث عنه لاحقًا بشكل موسع في فصل السلاسل النصية.

المحارف ليس لها نوع مخصص للأسف

يوجد في بعض اللغات البرمجية نوعًا خاصًّا من البيانات للمحرف المفرد (character) ويسمى `char` في لغة C و Java مثلًا.

لا يوجد مثل هذا النوع في JavaScript. ويوجد فقط نوع واحد وهو `string` (سلسلة نصية) الذي من الممكن أن يحتوي على محرف واحد أو أكثر.

2.5.4 النوع boolean: قيمة ثنائية (بوليانية)

يأخذ النوع `boolean` إحدى القيمتين: `true` (صح) أو `false` (خطأ) وتدعى هاتان القيمتان بالقيم الثنائية المنطقية أو البوليانية (مؤلفة من قيمتين فقط، صح وخطأ). من الشائع استخدام هذا النوع لحفظ البيانات الـتي لهـا قيمـة من إحـدى قيمـتين فقـط (نعم / لا): `true` تعـني "نعم، صـحيح"، و `false` تعـني "لا، خطـأ". إليك المثال التالي:

```
let nameFieldChecked = true; // name نعم، جرى تحديد الحقل
let ageFieldChecked = false; // age لا، الحقل غير محدّد
```

أضف إلى ذلك أن القيم المنطقية تعبِّر عن ناتج عمليات الموازنة:

```
let isGraeter = 4>1;
alert(isGreater); // true إذ نتيجة الموازنة محقَّقة أي
```

2.5.5 القيمة الخالية: null

لا تنتمي القيمة الخالية `null` إلى أي نوع من أنواع البيانات المذكورة سابقًا ولكن لها نوع خاص. يملك هـذا النوع الخاص قيمة واحدة هي `null`:

```
let age = null;
```

لا يعبِّر النوع `null` في لغة JavaScript عن مرجع لكائن غير موجود أو مؤشر خـالي كمـا في لغـات برمجيـة أخرى. ولكنه عبارة عن قيمة خاصة تمثِّل "لا شيء"، أو "فارغ"، أو "قيمة غير معلومة". فمثلًا، الشيفرة البرمجيـة السابقة تعدُّ المتغير `age` غير معلوم أو فارغ لسبب ما.

2.5.6 القيمة الغير معرفة: undefined

يمكن تمييز نوع آخر أيضًا للبيانات وهو "غير مُعرَّف" الذي يمثلـه النـوع `undefined`. ويشـكل هـذا النـوع نوعًا خاصًّا قائمًا بنفسه تمامًا مثل النوع `null`. أما معنى "غير معرَّف" أي أنه لم تُسنَد أية قيمة للمتغير بعد:

```
let x;
alert (x); // "undefined" تُعرَض القيمة
```

تقنيًا، من الممكن إسناد القيمة undefined لأي متغير:

```
let x=123;
x = undefined;
alert(x); // "undefined"
```

ولكن لا أنصحك بالقيام بذلك. نستخدم عادةً القيمة null لإسناد القيمة الخالية أو غير المعروفة لمتغير، ونستخدم undefined للتحقق فيما إذا كان للمتغير قيمة أو لا.

2.5.7 النوع object والنوع symbol: الكائنات والرموز

تسمى جميع الأنواع السابقة بالأنواع "الأساسية" (primitive) لأن قيمها تحتوي على شيء واحد فقط (سلسلة نصية أو عدد أو أي شيء آخر) بينما النوع object (كائن) هو نوعٌ خاصٌّ. ولكن تُستخدَم الكائنات لتخزين مجموعة من البيانات والكيانات (entity) الأكثر تعقيدًا. سنتعامل معها في فصل الكائنات بعد الانتهاء من دراسة الأنواع الأساسية.

يستخدم النوع symbol (رمز) لتشكيل معرِّف (identifier) مُميَّز للكائنات. يُفضَّل دراسة هذا النوع بعد الانتهاء من الكائنات ولكن ذكره هنا ضروري لاستكمال جميع أنواع البيانات.

2.5.8 العامل typeof

يُحدِّد العامل typeof نوع الشيء المُمرَّر إليه. ويكون مفيدًا عندما تحتاج إلى معالجة القيم بطرق مختلفة حسب نوعها، أو فقط القيام بفحص سريع لنوع البيانات. ويمكن كتابة الصياغة بطريقتين:

1. typeof x: عامل operator
2. typeof(x): تابع method

بمعنى آخر، من الممكن استخدامه مع أو بدون الأقواس، وتكون النتيجة واحدة. نتيجة استدعاء typeof x هو سلسلة نصية تمثل اسم النوع:

```
typeof undefined     // "undefined"
typeof 0             // "number"
typeof true          // "boolean"
typeof "foo"         // "string"
typeof Symbol("id") // "symbol"
typeof Math          // "object" (1)
```

```
typeof null         // "object" (2)
typeof alert        // "function" (3)
```

قد تحتاج لتوضيح نتائج الأسطر الثلاث الأخيرة:

1. `Math` هو كائن مضمَّن في JavaScript ويوفر العمليات الرياضية. سنتحدث عنه لاحقًا في فصل الأعداد، وذكرناه هنا فقط كمثال عن نوع الكائن.
2. نتيجة `typeof null` هي "object" أي كائن. أليس هذا خطأ؟! بالتأكيد. وقد تم الاعتراف بذلك رسميًا، ولكن بقيت هذه النتيجة من أجل المحافظة على التوافقية. القيمة "null" ليست كائنًا بل هي قيمة لنوع منفصل بحد ذاته. لذلك، مرةً أخرى، نؤكد أنَّ هذا خطأ من اللغة نفسها.
3. نتيجة `typeof alert` هي "function" (دالة)، لأنَّ `alert` هو دالة في لغة JavaScript. سنتحدث لاحقًا عن التوابع في الفصول القادمة وستلاحظ أنَّه لا يوجد نوع خاص اسمه "function" في لغة JavaScript. تنتمي التوابع إلى نوع الكائنات ولكن لها معاملة خاصة في `typeof`. شكليًا، هذا خطأ، ولكنه مفيد جدًا عند التطبيق العملي.

2.5.9 الخلاصة

هناك سبعة أنواع أساسية من أنواع البيانات في JavaScript هي:

- `number`: يمثِّل الأعداد بكل أنوعها، الصحيحة والعشرية.
- `string`: يمثِّل السلاسل النصية. ويمكن أن تحتوي السلسلة النصية على محرف واحد أو أكثر، ولكن لا يوجد نوع خاص من أجل المحرف الواحد.
- `boolean`: يمثِّل القيم المنطقية (صح / خطأ).
- `null`: يمثِّل القيم الخاوية. وهو نوع خاص يملك قيمة وحيدة وهي القيمة `null`.
- `undefined`: قيمة غير معرَّفة تكون قيمةً للمتغيرات التي لم تسند قيمة محدَّدة لها بعد. وهو نوع خاص يملك قيمة وحيدة وهي `undefined`.
- `object`: كائن من أجل بنى البيانات الأكثر تعقيدًا.
- `symbol`: رمز للمعرفات الخاصة.

ويسمح المعامل `typeof` بمعرفة أنوع القيم المخزنة في المتغيرات:

- له شكلين: `typeof x` أو `typeof(x)`.
- يعيد سلسلة نصية تحوي اسم النوع، مثل "string".

- من أجل `null` يعطي القيمة "object" وهذا خطأ في اللغة، فالقيمة الخالية ليست من نوع الكائن.

سنركز في الفصل القادم على القيم الأساسية ثم ننتقل إلى الحديث عن الكائنات.

2.5.10 تمارين

ا. إشارات اقتباس السلاسل النصية

الأهمية: ★★★★★

ما هو خرج هذا السكربت؟

```
let name " Ilya";
alert (`hello ${1}`); //?
alert (`hello ${"name"}`); //?
alert (`hello ${name}`); //?
```

الحل:

تُضمن الفاصلة العليا الخلفية التعابير الموجودة داخل `${…}` في السلسلة النصية:

```
let name " Ilya";
// 1 قيمة التعبير هو
alert (`hello ${1}`); // hello1

// "name" قيمة التعبير هي السلسلة النصية
alert (`hello ${"name"}`); // hello name

// name قيمة التعبير هو ما ضُمِّن بالمتغير
alert (`hello ${name}`); // hello Ilya
```

2.6 الدوال التفاعلية: alert، prompt، confirm

سنتطرَّق في هذا الفصل من الكتاب إلى لغة JavaScript كما هي بدون تعديلات خاصة بالبيئة. لكن ما زلنا نستخدم المتصفح بيئةً تجريبيةً، لذلك يجب أن تتعرف على عددٍ قليلٍ من دوال واجهة المستخدم الخاصة به. ستتعرف في هذا الفصل على هذه الدوال التفاعلية الخاصة بالمتصفح.

2.6.1 الدالة alert

الصياغة:

```
alert(message);
```

تعرض هذه الدالة رسالة نصية وتوقف تنفيذ السكربت مؤقتًا حتى يضغط المستخدم على "موافق" (OK). إليك الشيفرة البسيطة التالية مثلًا:

```
alert("مرحبًا");
```

تسمى الرسالة النصية التي تظهر على شكل نافذة صغيرة تدعى "النافذة المنبثقة الشرطية" (modal window، وهي عنصر تحكم رسومي)؛ تعني كلمة "شرطية" أنه لا يمكن للزائر التفاعل مع بقية الصفحة، أو الضغط على أزرار أخرى وما إلى ذلك، إذ تشترط عليه التفاعل معها فقط، أي حتى يضغط على "موافق" (Ok) في هذه الحالة.

2.6.2 الدالة prompt

تقبل الدالة `prompt` وسيطين (arguments) لتكون صياغتها بالشكل التالي:

```
result = prompt(title, [default]);
```

تعرض هذه الدالة نافذة منبثقة شرطية مع رسالة نصية مخصَّصة، وحقل إدخال للمستخدم، وزرَّين (موافق OK وإلغاء CANCEL).

- الوسيط `title`: هو عبارة عن النص الذي سيعرض للمستخدم.
- الوسيط `default`: هو وسيط اختياري (لذلك وضع ضمن القوسين []) يمثِّل القيمة الأولية لحقل الإدخال الخاص بالمستخدم.

قد يكتب المستخدم شيئًا ما في حقل الإدخال، ثمَّ يضغط على موافق Ok. أو يمكنه إلغاء الإدخال عند الضغط على إلغاء CANCEL أو الضغط على مفتاح الهروب Esc.

استدعاء الدالة prompt يرجع سلسلة نصية تمثِّل القيمة التي أدخلها المستخدم في حقل الإدخال أو يرجع القيمة null إذا تم الخروج من النافذة وإلغائها. جرب نفِّذ المثال التالي في الطرفية وعدل عليه:

```
let age = prompt('كم عمرك؟', 100);
alert(`عمرك ${age} سنة!`);
```

في IE (أي المتصفح Internet Explorer)، دائمًا ما يتم إضافة الوسيط default. أي هذا الوسيط اختياري في جميع المتصفحات باستثناء المتصفح IE الذي يعدُّه إجباريًّا، وإذا لم نحدِّد قيمته، يفترض المتصفح Internet Explorer أنَّ قيمته "undefined".

نفِّذ هذه الشيفرة في متصفح Internet Explorer لرؤية الناتج:

```
let test = prompt("Test");
```

لجعل الدالة prompt تعمل جيدًا في المتصفح IE، نوصي دائمًا بتمرير قيمة الوسيط الثاني default:

```
let test = prompt("Test", ''); // <-- for IE
```

2.6.3 الدالة confirm

الصياغة:

```
result = confirm(question);
```

تُظهر الدالة confirm نافذة منبثقة شرطية تحتوي على سؤال question، وزريّن (موافق OK وإلغاء CANCEL). تكون النتيجة true إذا ضغط المستخدم على الزر "Ok" وتكون false عدا ذلك. جرِّب المثال التالي في طرفيتك:

```
let isBoss = confirm("Are you the boss?");
alert( isBoss );
```

2.6.4 الخلاصة

ذكرنا في هذا الفصل ثلاثة دوال للتفاعل مع مستخدمي الموقع وهي:

- الدالة alert: تعرض رسالة لإعلام المستخدم بشيء ما، وُتعطل كافة عمليات الصفحة حتى يتفاعل مع هذه الرسالة.
- الدالة prompt: تعرض رسالة تطلب من المستخدم إدخال شيء ما في حقل إدخال خاص لتعيد القيمة المدخلة في سلسلة نصية، أو ترجع القيمة null إذا تم العملية.

- الدالة `confirm`: تعرض رسالة (بمثابة سؤال) وتنتظر من المستخدم الرد عليها بـالقبول أو الـرفض، أي تكون النتيجة `true` إذا تم الضغط على زر "Ok" أو تكون `false` عدا ذلك.

كل هذه الدوال مشروطة: فهي تتوقف عن تنفيذ السكربت ولا تسمح للمستخدم بالتفاعل مع بقية الصفحة حتى يتم التفاعل مع النافذة التي تعرضها. هناك اثنين من القيود التي تشترك بها جميع الدوال المذكورة أعلاه:

1. يحدد المتصفح الموقع الذي ستظهر فيه النافذة، وعادة ما يكون في الوسط أو الأعلى.
2. يعتمد شكل النافذة أيضًا على المتصفح، ولا يمكننا تعديله.

هذا هو ثمن البساطة. هنـاك طـرق أخـرى لإظهـار نوافـذ أكـثر جمـالًا وفاعليـة، ولكن إذا كـانت التنسـيقات الجمالية غير مهمة، فهذه الدوال تفي بالغرض.

2.6.5 تمارين

ا. صفحة بسيطة

الأهمية: ★★★★☆

أنشئ صفحة ويب تطلب اسمًا ما ثم تعرضه.

الحل:

شيفرة JavaScript:

```
let name = prompt("What is your name?", "");
alert(name);
```

الشيفرة كاملة:

```
<!DOCTYPE html>
<html>
<body>
  <script>
    'use strict';
    let name = prompt("What is your name?", "");
    alert(name);
  </script>
</body>
</html>
```

2.7 التحويل بين الأنواع

تُحوِّل الدوال (functions) والعاملات (operators)، في أغلب الأحيان، القيم التي تصلها إلى النوع الذي يناسبها وهذا ما يسمى "تحويل النوع" (Type Conversion). تُحوِّل الدالة `alert` مثلًا أي قيمة إلى سلسلة نصية تلقائيًا لإظهارها، وتُحوِّل والعمليات الرياضية القيم التي تستعملها إلى أعداد. كما توجد العديد من الحالات التي نكون فيها بحاجة لتحويل قيمة بصريح العبارة إلى النوع المطلوب.

لاحظ أنَّ هذا الفصل لا يتحدث عن الكائنات. سندرس الأنواع الأساسية أولًا، وبعدها ننتقل إلى الكائنات، ثم سنتعلم طريقة تحويل الكائن في فصل التحويل من كائن إلى أساسي.

2.7.1 التحويل إلى سلسلة نصية

يحدث التحويل إلى سلسلة نصية عندما نحتاج إلى الشكل النصي لقيمة ما. مثلًا، تُحوِّل الدالة `alert(value)` القيمة `value` المُمرَّرة إليها إلى سلسلة نصية لإظهارها. ويمكن أيضًا استدعاء الدالة `String(value)` لتحويل أي قيمة `value` إلى سلسلة نصية:

```
let value=true;
alert(typeof value); // Boolean

value = String(value); // "true" أصبحت الآن سلسلة نصية قيمتها
alert(tyoeof value); // string
```

يكون ناتج التحويل إلى سلسلة نصية عادةً واضحًا، فالقيمة المنطقية `false` تصبح `"false"`، والقيمة الخالية `null` تصبح `"null"` وهكذا.

2.7.2 التحويل إلى عدد

يحدث التحويل العددي في الدوال والتعابير الرياضية تلقائيًا. كما في حالة تطبيق عملية القسمة `/` على قيمتين غير عدديتين:

```
alert("6"/"2"); // الناتج: 3، إذ تحول السلاسل النصية إلى أعداد
```

بإمكاننا استخدام الدالة Number(value) لتحويل القيمة `value` المُمرَّرة إليها بشكل صريح إلى عدد:

```
let str = "123";
alert(typeof str); // string
let num =  Number(str); // 123 يتحول إلى العدد
alert(typeof num); // number
```

يكون التحويل الصريح ضروريًّا عندما نقرأ القيمة من مصدر نصي مثل العنصر text في النمـوذج <form> ويكون المطلوب إدخال عدد. ويكون ناتج التحويل NaN (اختصـارًا للعبـارة Not a Number)، إذا كـان من غـير الممكن تشكيل عدد من السلسلة النصية. إليك مثلًا:

```
let age = Number("an arbitrary string instead of a number");

alert(age); // NaN, فشلت عملية التحويل
```

قواعد التحويل العددي:

القيمة	ناتج التحويل
Undefined	NaN
null	0
true و false	1 و 0
string (سلسلة نصية)	يتم إزالة المسافات الفارغة من البداية والنهاية. إذا كانت السلسلة النصية المتبقية فارغة، تكون النتيجة 0، أو تتم قراءة العدد من السلسلة النصية. والخطأ يعطي النتيجة NaN.

أمثلة:

```
alert (Number("     123   ")); // 123
alert(Number("123z")); // NaN (error reading a number at "z")
alert(Number(true)); // 1
alert(Number(false)); // 0
```

لاحظ أنَّ القيمتين null و undefined لهما خرج مختلف هنا: قيمة null تصبح 0، بينمـا undefined تصبح NaN.

ا. جمع السلاسل النصية عبر المعامل +

تُحوَّل أغلب العمليات الرياضية القيم إلى أعداد. ولكن هناك استثناء وحيد يحدث عن الجمـع عـبر المعامـل "+" وكون إحدى القيم المطلوب جمعها سلسلة نصية. يجري آنـذاك تحويـل القيمـة الأخـرى إلى سلسـلة نصـية أيضًا ثمَّ تُضاف إلى القيمة الأخرى وتُجمعان معًا في سلسلة نصية واحدة. إليـك المثـال التـالي الـذي يوضـح مـا سبق:

```
alert(1+'2'); // '12' (سلسلة نصية على اليمين)
alert('1'+2); // '12' (سلسلة نصية على اليسار)
```

يحدث هذا فقط عندما يكون أحد المعاملات على الأقل سلسلة نصية. أمَّا في الحالات الأخرى، تُحوَّل القيم إلى أعداد.

2.7.3 التحويل إلى قيمة منطقية

التحويل المنطقي وهو الأسهل من بين جميع عمليات التحويل. يحدث في العمليات المنطقية (ستتعرف لاحقًا على التحقق من الشروط وغيرها من العمليات المماثلة)، ولكن من الممكن تطبيقه أيضًا بشكل صريح عن طريق استدعاء Boolean(value).

قاعدة التحويل:

- القيم التي تكون فارغة، مثل 0، والسلسلة النصية الفارغة، والقيمة `undefined`، والقيمة `NaN` تُحوَّل جميعها إلى `false`.
- أمَّا ما تبقى، فيُحوَّل إلى القيمة `true`.

مثلًا:

```
alert(Boolean(1)); // true
alert(Boolean(0)); // false
alert(Boolean("hello")); // true
alert(Boolean ("")); // false
```

لاحظ أنَّ السلسلة العددية التي تحتوي على 0 تصبح `true`. ولكن هناك بعض اللغات البرمجية (مثل PHP) تحول "0" إلى القيمة `false`. ولكن في لغة JavaScript، تُحوَّل السلسلة النصية الغير فارغة دائمًا إلى القيمة `true`:

```
alert(Boolean("0"); // true
alert(Boolean(" "); // true إلى (وأي سلسلة نصية غير فارغة) تحول المسافات
```

2.7.4 الخلاصة

التحويلات الثلاث الأكثر انتشارًا للأنواع هي التحويل إلى سلسلة نصية، أو عدد، أو قيمة منطقية.

- التحويل إلى سلسلة نصية: يحدث عندما نريد إظهار خرج معين ويمكن تنفيذه عن طريق String(value). ويكون خرج التحويل إلى سلسلة نصية واضحًا بالنسبة للأنواع الأساسية.
- التحويل إلى عدد: يحدث في العمليات الرياضية ومن الممكن تنفيذه باستخدام `Number(value)`. ويتبع هذا التحويل القواعد التالية:

القيمة	ناتج التحويل
Undefined	NaN
null	0
true و false	1 و 0
string	يتم قراءة السلسلة النصية كما هي وتجاهل الفراغات من البداية والنهاية. السلسلة الفارغة تصبح 0. وعند وجود خطأ في نتيجة التحويل تكون النتيجة NaN.

- التحويل إلى قيم منطقية: يحدث في العمليات المنطقية ويمكن تنفيذه باستخدام Boolean(value). ويتبع هذا التحويل القواعد التالية:

القيمة	ناتج التحويل
0, null, undefined, NaN, ""	false
القيم الأخرى	true

معظم هذه القواعد سهلة الفهم والحفظ. ولكن هناك بعض الاستثناءات أو الأخطاء الشائعة مثل:

- القيمة `undefined` كعدد هي NaN وليست 0.
- "0" والسلاسل النصية التي تحتوي على فراغات فقط مثل " " تُحوَّل إلى القيمة المنطقية `true`.

تذكر أننا لم ندرس الكائنات بعد وسنعود إليها لاحقًا في فصل التحويل من الكائنات إلى الأنواع الأساسية المخصص للكائنات ولكن بعد تعلم المزيد من مبادئ JavaScript.

2.7.5 تمارين

ا. تحويلات الأنواع

الأهمية: ★★★★★

ما هي نتيجة التعابير التالية؟

```
""+1+0
""-1+0
true+false
6/"3"
"2"*"3"
4+5+"px"
"$"+4+5
"4"-2
```

```
"4px"-2
7/0
"   -9   "+5
"   -9   "-5
null+1
undefined+1
```

فكر جيدًا، ثم اكتب حلك وقارنه مع الإجابة.

الحل:

```
""+1+0  ="10" // (1)
""-1+0 = -1 //(2)
true+false
6/"3" =2
"2"*"3" = 6
4+5+"px" = "9px"
"$"+4+5 = "$45"
"4"-2 = 2
"4px"-2 = NaN
7/0 = Infinity
"   -9   "+5 = "   -9   5" //(3)
"   -9   "-5 = -14 //(4)
null+1 = 1 //(5)
undefined+1 = NaN //(6)
```

1. الجمع مع سلسلة نصية مثل `1 +""` يحول العدد 1 إلى سلسلة نصية: `""+1="1"`، ثم لدينا `"1"+0` التي تنطبق عليها القاعدة نفسها.
2. الطرح `-` (مثل معظم العمليات الرياضية) يعمل فقط مع القيم العددية، وبالتالي يحول السلسلة النصية الفارغة "" إلى 0.
3. الجمع مع سلسلة نصية يضم العدد 5 إلى السلسلة النصية.
4. تحول عملية الطرح دائمًا القيم إلى أعداد. لذلك يحول القيمة `"   -9   "` إلى `-9` (ويتجاهل الفراغات حولها).
5. تصبح قيمة null هي 0 بعد تحويلها إلى عدد.
6. القيمة الغير معرفة undefined تصبح NaN بعد تحويلها إلى عدد.

2.8 عوامل Operators

تعرَّفنا على العديد من العاملات الحسابية في المدرسة، مثل الجمع +، الضرب *، الطرح - وما إلى ذلك. في هذا الفصل، سنشرح عاملات أخرى لم نتعرف عليها في المدرسة بالإضافة إلى تلك المعاملات.

2.8.1 مصطلحات جديدة: أحادي، ثنائي، عامل

قبل أن نمضي قدمًا في هذا الفصل، دعنا نتعرَّف على بعض المصطلحات الجديدة.

- المُعامَلات (operands، ومفردها مُعامَل): هي العناصر (القيمة) التي تُنفَّذ عملية العاملات (operators) عليها (أي التي تتعامل معها لذلك سمي مُعامَل). على سبيل المثال، في عملية الضرب 5 * 2، يوجد مُعامَلان؛ المُعامَل الأيسر هو 2 والأيمن هو 5. يطلق عليها البعض أحيانًا الاسم "وسائط" (arguments) بدلًا من "مُعامَلات" (operands).
- العامل الأحادي: يكون العامل أحاديًّا أو وحيدًا (unary) إذا كان يتعامل مع مُعامَل (operand) واحد فقط. على سبيل المثال، عملية النفي هي عملية أحادية، إذ تعكس إشارة العدد، فيصبح سالبًا - إذا كان موجبًا + والعكس صحيح:

```
let x = 1;

x = -x;
alert( x ); // -1, تم تطبيق النفي الأحادي
```

- العامل الثنائي: يكون العامل ثنائيًّا (binary) إذا كان يتعامل مع مُعامَلان (operands). يوضح المثال التالي إشارة السالب التي في المثال السابق عندما تكون في المعامل الثنائي:

```
let x = 1, y = 3;
alert( y - x ); // 2, معامل الطرح الثنائي يطرح القيم
```

تقنيًّا، نتحدث هنا عن نوعين من العاملات لإشارة واحدة - هما: النفي الأحادي (مُعامَل واحد: عكس الإشارة) والطرح (مُعامَلان: عملية طرح)، لذا يجب التفريق بينهما.

2.8.2 وصل سلاسل نصية عبر العامل +

الآن، سنتعرَّف على ميِّزات خاصة بعاملات JavaScript تتجاوز تلك التي تعلمناها في المدرسة.

المتعارف عليه أنَّ عامل الجمع + يجمع الأرقام. لكن إذا تم تطبيق عامل الجمع + على سلاسل نصية، فإنه يُوصِل هذه السلاسل النصية مع بعضها بعضًا ويضعها في سلسلة نصية واحدة.

```
let s = "my" + "string";
alert(s); // mystring
```

لاحظ أنه يتم تحويل العدد إلى سلسلة نصية عن طريق وضعه داخل علامات الاقتباس ("") ويعامل آنذاك معاملة السلسلة النصية. ولاحظ أيضًا إذا كان أحد العاملين (operands) عبارة عن سلسلة نصية، فسيُحوَّل الآخر تلقائيًا إلى سلسلة نصية. إليك المثال التالي:

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

كما رأيت، لا يهم ما إذا كان المُعامَل الأول عبارة عن سلسلة نصية أم الثاني. القاعدة بسيطة: إذا كان أحد المُعامَلين عبارة عن سلسلة نصية، فسيُحوَّل الآخر إلى سلسلة نصية أيضًا.

لكن انتبه! المُعامَلات تبدأ من اليسار إلى اليمين. فإذا كان هناك عددان متبوعان بسلسلة نصية، فستُجمَع الأعداد قبل تحويلها إلى سلسلة نصية:

```
alert(2 + 2 + '1' ); // "221" وليس "41" أي أن الناتج
```

خاصية وصل السلاسل النصية وجمعها في سلسلة واحدة (String concatenation) ميزة خاصة بمعامل الجمع +، بينما تعمل المعاملات الحسابية الأخرى مع الأعداد فقط. فعلى سبيل المثال، الطرح والقسمة:

```
alert( 2 - '1' ); // 1
alert( '6' / '2' ); // 3
```

2.8.3 خاصية التحويل العددي

يوجد لعامل الجمع + صيغتان: الأول عامل الجمع الثنائي كما في خاصية دمج السلاسل النصية، والثاني عامل الجمع الأحادي كما في هذه الخاصية. عامل الجمع الأحادي أو بمعنى آخر، عامل الجمع + المطبق على قيمة واحدة، لا يُحدث تغييرًا على الأعداد. ولكن إذا لم يكن المُعامَل عددًا، فإن عملية الجمع الأحادي تحوِّله إلى عدد. اطلع على المثال التالي:

```
// لا تأثير على الأعداد
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2
```

```
// تحويل غير الأعداد
alert( +true ); // 1
alert( +"" );   // 0
```

يقوم هذا المعامل هنا مقام الدالة `Number`، لكنه أقصر وأبسط.

تنشأ الحاجة لتحويل السلاسل النصية إلى أعداد في كثير من الأحيان. على سبيل المثال، إذا كنا نحصل على القيم من قالب HTML، فهي غالبًا ما تكون عبارة عن سلاسل نصية. ولكن ماذا لو أردنا دمجها؟ معامل الجمع الثنائي سيضيفها كسلاسل نصية:

```
let apples = "2";
let oranges = "3";

alert( apples + oranges ); // "23", معامل الجمع الثنائي يدمج السلاسل النصية
```

إذا كنا نريد أن نعاملها كأعداد، نحتاج إلى تحويلها ثم دمجها:

```
let apples = "2";
let oranges = "3";

// كلا القيمتين حُولتا إلى أعداد قبل تطبيق معامل الجمع الثنائي
alert( +apples + +oranges ); // 5

// بصيغة أطول
// alert( Number(apples) + Number(oranges) ); // 5
```

من وجهة نظر عالم رياضيات، تبدو كثرة عاملات الجمع غريبة. ولكن من وجهة نظر المبرمج، لا يوجد شيء مميز: يتم تطبيق معاملات الجمع الأحادي أولاً لتحويل السلسلة النصية إلى عدد، ثم يجمع معامل الجمع الثنائي العددين الناتجين.

لماذا يتم تطبيق عاملات الجمع الأحادية على القيم قبل تطبيق عملية الجمع؟ هذا يعود للأولوية الأعلى في التنفيذ كما سنرى بعد قليل.

2.8.4 ترتيب أولوية العمليات

إذا كان في المعادلة أكثر من عامل واحد، يتم تحديد ترتيب التنفيذ حسب أولوية هذه العاملات. في المدرسة، تعلمنا جميعًا أنه يجب حساب الضرب في المعادلة `1 + 2 * 2` قبل الجمع، هذا تحديدًا ما يسمى أولوية العملية الرياضية. أي أن عملية الضرب لها أولوية أعلى من الجمع.

تتخطى الأقواس أولوية أي عامل، لذلك إذا لم نكن راضين عن أولوية العمليات (operations) في المعادلة، يمكننا استخدام الأقواس لتغييرها. على سبيل المثال: `(1 + 2) * 2`.

هناك العديد من العاملات في JavaScript. كل معامل لديه رقم في ترتيب الأولوية، والمعامل صاحب الأولوية الأعلى يُنفَّذ أولًا. إذا كانت الأولوية لمجموعة عاملات في نفس الترتيب، يكون التنفيذ من اليسار إلى اليمين. إليك مقتطف من جدول الأولوية (لست بحاجة إلى حفظ ذلك، لكن لاحظ أن العاملات الأحادية أعلى أولوية من العاملات الثنائية المقابلة).

الأولوية	اسم المعامل	إشارة المعامل
...	...	...
16	الجمع الأحادي	+
16	النفي الأحادي	-
14	الضرب	*
14	القسمة	/
13	الجمع الثنائي	+
13	الطرح الثنائي	-
...	...	...
3	الإسناد	=
...	...	...

كما ترى، عامل الجمع الأحادي له الأولوية 16 وهي أعلى من 13 لعامل الجمع الثنائي. لهذا السبب، يُنفَّذ عامل الجمع الأحادي أولًا في المعادلة `+apples + +oranges` الذي يمثِّل عملية التحويل قبل عملية الجمع وهذا هو المطلوب.

2.8.5 عامل الإسناد =

لاحظ أنَّ معامل الإسناد = مدرجٌ في جدول الأولوية الخاص بالمعاملات مع أولوية منخفضة للغاية وهي 3. لهذا السبب، عندما نسند قيمة لمتغير، مثل `x = 2 * 2 + 1`، تُنفَّذ العمليات الحسابية حسب أولويتها ثمَّ تُسنَد القيمة الناتجة (5) إلى المتغيِّر x وتُخزَّن فيه.

```
let x = 2 * 2 + 1;

alert( x ); // 5
```

من الممكن سَلسَلة معاملات الإسناد:

```
let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4
alert( b ); // 4
alert( c ); // 4
```

يتم تنفيذ سَلسَلة عاملات الإسناد من اليمين إلى اليسار. أولاً، يُقيَّم التعبير `2 + 2` الموجود في أقصى اليمين ثم تُسنَد القيمة الناتجة عنه إلى المتغيرات الموجودة على اليسار: `c`، و `b` و `a`. في النهاية، تشترك جميع تلك المتغيرات في القيمة النهائية الناتجة.

ا. عامل الإسناد = يُرجع قيمة

العامل دائما يرجع قيمة وهذا واضح في عامل الجمع `+` أو الضرب `*`، ويتبع عامل الإسناد `=` أيضًا هذه القاعدة. فالاستدعاء `x = Value` يُخزِّن القيمة في `x` ثم يرجعها.

يوضح المثال التالي كيفية الاستفادة من هذه الميزة باستخدام القيمة الناتجة عن عملية الإسناد في معادلة أخرى أكثر تعقيدًا، وبذلك نضرب عصفورين بحجر واحد:

```
let a = 1;
let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3
alert( c ); // 0
```

في المثال أعلاه، تكون نتيجة `a = b + 1` هي القيمة التي جرى إسنادها إلى `a` أي `3`، ثم يتم طرح من هذه القيمة `3`. شيفرة مضحكة، أليس كذلك؟! لكن يجب أن نفهم كيف تعمل لأنه في بعض الأحيان نراها في مكتبات خارجية كتبها آخرون، ولا يُفضل إطلاقًا أن نكتب مثل هذه الشيفرات، لأنَّ هذه الحيل لا تجعل من الشيفرة أكثر وضوحًا أو أكثر قابلية للقراءة.

2.8.6 عامل باقي القسمة %

عامل باقي القسمة `%`، لا يرتبط بالنسب المئوية. فالناتج من عملية مثل `a % b` هو العدد المتبقي من قسمة العدد الصحيح `a` على العدد الصحيح `b`. إليك مثال عنه:

```
alert( 5 % 2 ); // 1 (هو الباقي من قسمة 5 على 2)
alert( 8 % 3 ); // 2 (هو الباقي من قسمة 8 على 3)
alert( 6 % 3 ); // 0 (هو الباقي من قسمة 6 على 3)
```

2.8.7 عامل القوة **

عامل القوة ** هو إضافة حديثة للغة JavaScript. ليكن لدينا الرقم الطبيعي b، فيكون ناتج العملية a ** b هو العدد a مضروبًا في نفسه عدد b من المرات:

```
alert( 2 ** 2 ); // 4  (2 * 2)
alert( 2 ** 3 ); // 8  (2 * 2 * 2)
alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2)
```

يعمل عامل القوة مع الأعداد غير الصحيحة أيضًا مثل:

```
alert( 4 ** (1/2) ); // 2 (معامل القوة ½ هو نفس الجذر التربيعي)
alert( 8 ** (1/3) ); // 2 (معامل القوة ⅓ هو نفس الجذر التكعيبي)
```

2.8.8 عاملات الزيادة ++ والنقصان --

تُعدُّ الزيادة على العدد أو إنقاصه بمقدار 1 من أكثر العمليات العددية شيوعًا. لذلك، هناك عاملات خاصة لهذا الغرض:

- عامل الزيادة ++ يزيد على قيمة المتغير مقدار 1:

```
let counter = 2;
counter++;      // ببساطة counter = counter + 1 هو اختصار للعملية
alert( counter ); // 3
```

- عامل النقصان -- ينقص من قيمة المتغير مقدار 1:

```
let counter = 2;
counter--;      // counter = counter - 1 هو اختصار للعملية
alert( counter ); // 1
```

> عاملات الزيادة / النقصان خاصة بالمتغيرات فقط، ومحاولة تطبيقها على قيم عددية مثل ++5 سيعطي خطأ.

يمكن وضع هذين العاملين ++، -- قبل المتغير أو بعده.

- عندما يتبع العامل المتغير، هذا ما يسمى "النموذج اللاحق" (postfix form) مثل counter++.

- وبالمثل، "النموذج السابق" (prefix form)، عندما يأتي العامل قبل المتغير مثل ‎++counter.

كلا النموذجين يقومان بالعمل نفسه وهو الزيادة على قيمة المتغير counter بمقدار 1. إذن، هل هناك فرق؟ نعم، ولكن لا يمكننا رؤيته إلا إذا استخدمنا القيمة المعادة من العملية تلك.

دعني أوضح لك أكثر. كما نعلم، جميع العاملات ترجع قيمة وكذلك أيضًا عاملات الزيادة/النقصان. يعيد النموذج السابق القيمة الجديدة بينما يعيد النموذج اللاحق القيمة القديمة (قبل الزيادة / النقصان). إليك مثال على ذلك ليتضح كل الفرق وضوح الشمس:

```
let counter = 1;
let a = ++counter; // (*)

alert(a); // 2
```

في السطر المُميّز بالنجمة (*)، يزيد النموذج السابق ‎++counter قيمة المتغير counter ويعيد القيمة الجديدة الناتجة وهي 2. لذا، تُظهر الدالة alert القيمة 2. الآن، دعني استخدم النموذج اللاحق بدلًا من النموذج السابق لمعرفة الفرق:

```
let counter = 1;
let a = counter++; // (*)
alert(a); // 1
```

في السطر المُميّز بالنجمة (*)، يزيد النموذج اللاحق ‎++counter قيمة المتغير counter لكنه يعيد القيمة القديمة (قبل الزيادة). لذا، تُظهر الدالة alert القيمة 1.

مختصر القول:

- إذا لم يتم استخدام القيمة الناتجة من معاملي الزيادة والإنقاص، فلا يوجد فرق في النموذج الذي يتم استخدامه سواءً سابق أو لاحق:

```
let counter = 0;
counter++;
++counter;
alert( counter ); // 2, كلا السطرين يؤديان نفس الغرض
```

- أمَّا إذا كنت ترغب في زيادة قيمة المتغير واستخدامها مباشرةً، فأنت بحاجة إلى استعمال النموذج السابق:

```
let counter = 0;
alert( ++counter ); // 1
```

- وإذا كنت ترغب في زيادة قيمة المتغير ولكنك تريد استخدام قيمته السابقة، فاستعمل النموذج اللاحق:

```
let counter = 0;
alert( counter++ ); // 0
```

ا. عاملات زيادة / نقصان داخل عاملات أخرى

يمكن استخدام عاملات زيادة / نقصان (`++` / `--`) داخل المعادلات أيضًا، وأولويتها في التنفيذ أعلى من معظم العاملات الحسابية الأخرى. إليك المثال التالي:

```
let counter = 1;
alert( 2 * ++counter ); // 4
```

وازنه مع المثال التالي:

```
let counter = 1;
alert( 2 * counter++ ); // 2, لأن counter++ يرجع القيمة القديمة
```

رغم أنَّه مقبول من الناحية التقنية، إلا أنَّ هذه الشيفرة عادة ما تجعل الشيفرة الكلية صعبة القراءة. اتباع نموذج "سطر واحد يفعل أشياء متعددة" ليس جيدًا ويولد شيفرة صعبة القراءة والتنقيح.

أثناء تَصفُّح الشيفرة سريعًا يمكن أن لا تنتبه على شيء مثل `counter++`، أي لن تلاحظ أنَّ قيمة المتغير قد زادت. فتخيل ما سيحصل إن حدث خطأ ما مع وجود مئات مثل تلك الأسطر التي تنجز عدة أشياء سويةً! لذا أنصحك باستخدام أسلوب "سطر واحد - إجراء واحد" (one line – one action):

```
let counter = 1;
alert( 2 * counter );
counter++;
```

2.8.9 العاملات الثنائية

تَعدُّ العاملات الثنائية Bitwise operators المُعامَلات التي تطبَّق عليها على أنَّها أعداد صحيحة بحجم 32 بت وتعمل على مستوى تمثيلها الثنائي أو البتي (binary representation). هذه العوامل ليست خاصة بـ JavaScript، وهي مدعومة في معظم لغات البرمجة.

إليك قائمة بالعاملات الثنائية (Bitwise operators):

- العامل AND (&)
- العامل OR (|)
- العامل XOR (^)
- العامل NOT (~)
- عامل الإزاحة نحو اليسار LEFT SHIFT (<<)
- عامل الإزاحة نحو اليمين RIGHT SHIFT (>>)
- عامل الإزاحة نحو اليمين مع إدخال أصفار ZERO-FILL RIGHT SHIFT (>>>)

نادرًا ما تحتاج إلى استخدام مثل هذه العاملات. وتحتاج لفهمها إلى الخوض في التمثيل الثنائي للأعداد (الصيغة الثنائية البِتيَّة المقابلة للصيغة العشرية المفهومة للبشر) وليس من الجيد القيام بذلك الآن، خاصة أننا لسنا بحاجة إلى ذلك في هذا الوقت. إذا كنت فضولي يمكنك قراءة توثيق المعاملات الثنائية في JavaScript في موسوعة حسوب، وهذا كافيًا الآن. وسنشرحها عمليًّا عندما تنشأ حاجة حقيقية لها في مراحل متقدمة.

2.8.10 عاملات التعديل المباشر على المتغير

غالبًا ما نحتاج إلى تطبيق عامل على متغير ما وتخزين القيمة الجديدة في المتغير نفسه. انظر مثلًا إلى المثال التالي:

```
let n = 2;
n = n + 5;
n = n * 2;
```

يمكن اختصار الشيفرة السابقة باستخدام += و *=:

```
let n = 2;
n += 5; // (n = n + 5 وهذا يماثل) n = 7 أي تصبح
n *= 2; // (n = n * 2 وهذا يماثل) n = 14 تصبح الآن
alert( n ); // 14
```

عاملات التعديل والإسناد modify-and-assign متاحة لجميع العاملات الحسابية والثنائية مثل: /= و -= وتملك هذه العاملات الأولوية نفسها التي لدى معامل الإسناد العادي =، لذا فهي تُنفَّذ بعد معظم المعاملات الأخرى:

```
let n = 2;
n *= 3 + 5;
```

```
alert( n ); // 16  (القسم الأيمن يُقيَّم أولًا وهذا يماثل n *= 8)
```

2.8.11 عامل الفاصلة ,

عامل الفاصلة (Comma)ـ , هو واحد من أندر العاملات وأكثرها غرابة. في بعض الأحيان يتم استخدامه لكتابة شيفرة أقصر، لذلك نحن بحاجة إلى التعرف عليه لفهم ذلك.

يتيح لنا عامل الفاصلة تقييم العديد من المعادلات عند فصلها بفاصلة , . يجري تقييم كل قسم منها ولكن يتم إرجاع نتيجة آخر واحد فقط. تفحَّص المثال التالي:

```
let a = (1 + 2, 3 + 4);
alert( a ); // 7 (الناتج من جمع 4+3)
```

هنا، يُقيَّم القسم الأول من المعادلة 1+2 ويُتجاهَل قيمته ثم بعد ذلك يُقيَّم القسم الثاني 3+4 وتعاد قيمته كنتيجة نهائية.

عامل الفاصلة له أولوية منخفضة جدًا

يرجى ملاحظة أن عامل الفاصلة له أولوية منخفضة للغاية، أقل من عامل الإسناد =، لذلك الأقواس مهمة في المثال أعلاه. بدون الأقواس: a = 1 + 2 ، 3 + 4 يُنفَّذ عامل الجمع + أولاً، فتصبح المعادلة a= 3,7، ثم يُنفَّذ معامل الإسناد = فتصبح a=3، وأخيرًا لا يُطبَّق أي عامل على العدد الموجود بعد الفاصلة، أي يُتجاهَل العدد 7.

لماذا نحتاج إلى عامل يتجاهل كل القيم باستثناء قيمة الجزء الأخير؟ في بعض الأحيان، يستخدمه المبرمجون في بنيات أكثر تعقيدًا لوضع العديد من الإجراءات في سطر واحد. قد تكون أبسط تلك البنيات المعقدة هي بنية الحلقية التكرارية التالية:

```
// ثلاث معاملات في سطر واحد
for (a = 1, b = 3, c = a * b; a < 10; a++) {
 ...
}
```

تُستخدَم هذه الحيل في العديد من أطر JavaScript. لكنها في العادة لا تحسِّن قابلية القراءة للشيفرة، لذلك يجب عليك التفكير جيدًا قبل استخدامها.

2.8.12 تمارين

ا. النموذج السابق والنموذج اللاحق

الأهمية: ★★★★★

ما هي القيم النهائية للمتغيرات d ،c ،b ،a بعد تنفيذ الشيفرة التالية؟

```
let a = 1, b = 1;
let c = ++a; // ?
let d = b++; // ?
```

الحل:

الإجابة هي:

- a = 2
- b = 2
- c = 2
- d = 1

```
let a = 1, b = 1;

alert( ++a ); // 2, النموذج السابق يُرجع القيمة الجديدة
alert( b++ ); // 1, النموذج اللاحق يُرجع القيمة القديمة

alert( a ); // 2, تتم الزيادة مرة واحدة
alert( b ); // 2, تتم الزيادة مرة واحدة
```

ب. نتيجة عملية الإسناد

الأهمية: ★★★☆☆

ما هي قيم المتغيرين a و x بعد تنفيذ الشيفرة التالية؟

```
let a = 2;
let x = 1 + (a *= 2);
```

الحل:

الإجابة هي: a = 4 (مضروبًا في العدد 2) لنحصل على x = 5 (عبارة عن مجموع العدد 1 والعدد 4).

2.9 عاملات الموازنة

لا شك أنَّك تعرَّفت مسبقًا على عاملات الموازنة من أحد دروس الرياضيات في المدرسة وهي:

- أكبر/أصغر من: `a > b` - `a < b`.
- أكبر/أصغر من أو يساوي: `a >= b` - `a <= b`.
- المساواة: `a == b` (يرجى ملاحظة أنَّ هذا العامل عبارة عن علامة يساوي مزدوجة =. العلامة المنفردة `a = b` خاصة بمعامل الإسناد الذي تحدثنا عنه في الفصل السابق).
- عدم المساواة: يُعبر عنه في الرياضيات بالرمز ≠، ولكن يُعبِّر عنه برمجيًّا في JavaScript بإشارة يساوي منفردة مع إشارة تعجب قبلها بالشكل: `a != b`.

2.9.1 النتيجة عبارة عن قيمة منطقية

مثل كل العاملات الأخرى، تُرجِع معاملات الموازنة قيمة منطقية دومًا.

- `true`: تعني "نعم" أو "صواب" أو "الموازنة مُحقَّقة".
- `False`: تعني "لا" أو "خطأ" أو "الموازنة غير مُحقَّقة".

إليك أمثلة عن هذه العاملات:

```
alert( 2 > 1 );  // true (الموازنة صحيحة)
alert( 2 == 1 ); // false (الموازنة خطأ)
alert( 2 != 1 ); // true (الموازنة صحيحة)
```

يمكن إسناد نتيجة عامل الموازنة لمتغير تمامًا مثل أي قيمة:

```
let result = 5 > 4; // إسناد الناتج من عملية الموازنة لمتغير
alert( result ); // true
```

2.9.2 موازنة السلاسل النصية

لمعرفة ما إذا كانت السلسلة النصية أكبر من الأخرى، تستخدم JavaScript ما يسمى بترتيب القاموس (dictionary) أو ترتيب المعجم (lexicographical). بمعنى آخر، تتم موازنة السلاسل النصية حرفًا تلو الآخر. على سبيل المثال:

```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
```

```
alert( 'Bee' > 'Be' ); // true
```

خوارزمية موازنة سلسلتين نصيتين بسيطة:

1. موازنة الحرف الأول من كلا السلسلتين.
2. إذا كان الحرف الأول من السلسلة النصية الأولى أكبر (أو أقل) من السلسلة النصية الأخرى، حينها تكون الأولى أكبر (أو أقل) من الثانية وبذلك تنتهي عملية الموازنة.
3. لكن إذا كانت الأحرف الأولى في كلا السلسلتين متماثلة، وازن الأحرف الثانية بالطريقة نفسها.
4. كرر حتى نهاية أي سلسلة.
5. إذا انتهت كلا السلسلتين بالطول نفسه، فهما متساويتان. خلاف ذلك يكون السلسلة النصية الأطول هو الأكبر.

في الأمثلة أعلاه، تحصل الموازنة `'Z' > 'A'` على النتيجة في الخطوة الأولى بينما تُوازن السلسلتين النصيتين `"Glow"` و `"Glee"` حرفًا تلو الآخر حتى الحرف الأخير بالتسلسل التالي:

1. `G` هو نفسه `G`.
2. `l` هو نفسه `l`.
3. `o` أكبر من `e`. توقف هنا، السلسلة الأولى أكبر.

ليس قاموسًا حقيقيًا، ولكنه ترتيب ترميز اليونيكود

إنَّ خوارزمية الموازنة المذكورة أعلاه تعادل تقريبًا تلك المستخدمة في القواميس ودفاتر الهاتف، ولكن مع اختلاف بسيط يتعلق بحالة الحرف. فالحرف الكبير `"A"` لا يساوي الحرف الصغير `"a"` كما تعتقد فأيهما أكبر برأيك؟ الحرف الصغير `"a"` طبعًا. أسمعك تسأل لماذا؟ لأنَّ الحرف الصغير له رقم تسلسلي أكبر في جدول الترميز المقابل له في جدول اليونيكود المستخدم في JavaScript، حسنًا، سكفي إلى هنا وسنعاود الحديث عن هذا الموضوع والنتائج المترتبة عليه في فصل السلاسل النصية.

2.9.3 موازنة بين أنواع البيانات المختلفة

عند موازنة أنواع بيانات مختلفة، تحوِّل JavaScript القيم إلى أعداد. على سبيل المثال:

```
alert( '2' > 1 ); // true, تُحوَّل السلسلة النصية '2' إلى العدد 2
alert( '01' == 1 ); // true, تُحوَّل السلسلة النصية '01' إلى العدد 1
```

بالنسبة للقيم المنطقية، `true` تصبح `1` و `false` تصبح `0`. على سبيل المثال:

```
alert( true == 1 ); // true
alert( false == 0 ); // true
```

نتيجة مضحكة أليس كذلك؟!

من الممكن في الوقت نفسه أن تكون القيمتان:

- متساويتين.
- واحدة منهما true والأخرى false.

إليك مثال عن حالة مماثلة:

```
let a = 0;
alert( Boolean(a) ); // false

let b = "0";
alert( Boolean(b) ); // true

alert(a == b); // true!
```

من وجهة نظر JavaScript، هذه النتيجة طبيعية تمامًا. تُحوِّل عملية التحقق من المساواة القيم باستخدام التحويل العددي (وبالتالي السلسلة النصية "0" تساوي القيمة 0)، بينما يستخدم التحويل الصريح Boolean مجموعة أخرى من القواعد.

2.9.4 معامل المساواة الصارمة

معامل المساواة العادية == لديه مشكلة، وهي عدم قدرته على التمييز بين 0 و false:

```
alert( 0 == false ); // true
```

يحدث الشيء نفسه مع السلسلة النصية الفارغة:

```
alert( '' == false ); // true
```

يحدث هذا لأنَّ الأنواع المختلفة من العاملات تُحول إلى أعداد بواسطة معامل المساواة ==. السلسلة النصية الفارغة مثل false، تصبح صفرًا. ولكن ماذا نفعل إذا كنا نرغب في التمييز بين 0 و false؟ عامل المساواة الصارمة === يتحقَّق من المساواة دون تحويل القيم. بمعنى آخر، إذا كان a و b قيمتين من نوعين مختلفين، فسوف تُرجع المعادلة a === b على الفور false دون محاولة تحويلها، وبذلك يتحقق العامل === من تساوي النوع أولًا ثم تساوي القيمة.

دعنا نجرب:

```
alert( 0 === false ); // false, لأن القيمتين من نوعين مختلفين
```

هناك أيضًا عامل عدم مساواة صارم ‎!==‎ مماثل للعامل ‎!=‎.

يعد عامل المساواة الصارمة أطول قليلاً في الكتابة، ولكنه أكثر وضوحًا ويترك مساحة أقل للأخطاء.

2.9.5 الموازنة مع قيمة فارغة null وغير معرفة undefined

دعنا نرى المزيد من الحالات الخاصة. تسلك عملية الموازنة سلوكًا غير منطقي عند موازنة قيمة فارغة null أو غير مُعرَّفة undefined مع قيم أخرى، إذ هاتان القيمتان من نوعين مختلفين.

تذكر أنَّه للتحقق من المساواة بصرامة، استعمل العامل ‎===‎ دومًا:

```
alert( null === undefined ); // false
```

للتحقق من المساواة دون أخذ النوع بالحسبان، استعمل العامل ‎==‎. بمعنى أنَّ القيمتين null و undefined متساويتان فقط مع بعضهما ولكن ليستا متساويتان مع أي قيمة أخرى.

```
alert( null == undefined ); // true
```

أ. الرياضيات وعاملات الموازنة الأخرى ‎< > <= >=‎

تُحوَّل القيمة الفارغة وغير المُعرَّفة null و undefined إلى عدد: null تصبح 0، بينما undefined تصبح NaN. الآن دعنا نرى بعض الأشياء المضحكة التي تحدث عندما نطبق هذه القواعد، والأهم من ذلك هو كيفية تجنب هذا الفخ.

ب. نتيجة غريبة: null مقابل 0

دعنا نوازن null مع 0:

```
alert( null > 0 );  // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3) true
```

رياضيًّا، هذا غريب، إذ تشير النتيجة الأخيرة إلى أنَّ "null أكبر من أو تساوي 0"، لذلك يجب في إحدى الموازنات أعلاه أن تكون صحيحة true، لكن كلاهما خطأ.

السبب هو أنَّ التحقق من المساواة ‎==‎ وعاملات الموازنة ‎> < >= <=‎ تعمل بشكل مختلف. تحول الموازنات القيمة الفارغة null إلى عدد، وتعاملها على أنها 0 ولهذا السبب، نتيجة الموازنة في المثال السابق في السطر (3) ‎null >= 0‎ مُحقَّقة أي true وفي السطر (1) خطأ ‎null > 0‎.

من ناحية أخرى، يتم التحقق من المساواة `==` على `undefined` و `null` بدون أي تحويل، حيث أنه يساوي كل منهما الآخر ولا يساويها أي قيمة أخرى. لهذا السبب، نتيجة الموازنة في المثال السابق في السطر (2) `null == 0` غير محقَّقة.

القيمة غير المُعرَّفة `undefined` غير قابلة للموازنة أي لا ينبغي موازنتها `undefined` مع القيم الأخرى:

```
alert( undefined > 0 ); // false (1)
alert( undefined < 0 ); // false (2)
alert( undefined == 0 ); // false (3)
```

لماذا يُكره الصفر كثيرًا؟ دائما خطأ! حصلنا على هذه النتائج لأن:

- معاملات الموازنة في السطر `(1)` و`(2)` في المثال السابق تُرجع `false`، لأن القيمة `undefined` تُحوّل إلى `NaN` وهي عبارة عن قيمة رقمية خاصة تعيد `false` لكافة الموازنات.
- تُرجع عملية التحقق من المساواة في السطر `(3)` القيمة `false`، لأن القيمة غير المُعرَّفة `undefined` تساوي فقط القيمة `null` و `undefined`، ولا تساوي قيم أخرى.

2.9.6 تجنب المشاكل

لماذا ذكرنا هذه الأمثلة؟ هل يجب أن نتذكر هذه الخصائص في كل وقت؟ نعم، ولا فلا توجد إجابة دقيقة. على أي حال، ستصبح هذه الأشياء التي تراها صعبة ومعقدة تدريجيًا مألوفة وبديهية، ومع ذلك هناك طريقة جيدة للتهرب من مثل هذه الأمثلة:

- تعامل بالصورة المألوفة مع أي موازنة بها `undefined/null`، باستثناء المساواة الصارمة `===` فهي تحتاج معاملة استثنائية.
- لا تستخدم عاملات الموازنة `=> < > =<` مع متغير قد يأخذ إحدى القيمتين `null/undefined`، إلا إذا كنت متأكدًا حقًا مما تفعله. إذا كان المتغير عبارة عن هاتين القيمتين، فتحقق منها بشكل منفصل.

2.9.7 الخلاصة

- تُرجع عاملات الموازنة قيمة منطقية.
- تُوازن السلاسل النصية حرفًا تلو الآخر حسب ترتيب القاموس (dictionary).
- عند موازنة قيم من أنواع بيانات مختلفة، يتم تحويلها إلى أرقام (باستثناء التحقق من المساواة الصارمة).
- القيم الفارغة `null` وغير المعرَّفة `undefined` تساوي `==` بعضها بعضًا، ولا تساوي أي قيمة أخرى.

- كن حذرًا عند استخدام معاملات موازنة مثـل < أو > مـع متغيـرات يمكن أن تكـون فارغـة null أو غـير معرَّفة undefined ويفضل التحقق من ذلك بشكل منفصل.

2.9.8 تمارين

ا. الموازنات

الأهمية: ★★★★★

ماذا ستكون نتيجة هذه الموازنات

```
5 > 4
"apple" > "pineapple"
"2" > "12"
undefined == null
undefined === null
null == "\n0\n"
null === +"\n0\n"
```

الحل:

```
5 > 4                  → true
"apple" > "pineapple"  → false
"2" > "12"             → true
undefined == null      → true
undefined === null     → false
null == "\n0\n"        → false
null === +"\n0\n"      → false
```

بعض الأسباب:

1. نتيجة الموازنة واضحة، صحيحة.
2. حسب موازنة القاموس، النتيجة خاطئة.
3. حسب موازنة القاموس مرة أُخرى، الحرف الأول من القيمة "2" أكبر من الحرف الأول من القيمة "1".
4. القيم null و undefined تساويان بعضهما فقط.
5. المساواة الصارمة صارمة فيما يخص نوع المتغير، وعند اختلاف نوع طرفي العامل فـإن النتيجـة تصـبح خاطئة.

6. انظر للنقطة رقم (4).
7. مساواة صارمة لنوعان مختلفان من القيم.

2.10 العاملات الشرطية

نحتاج في بعض الأحيان إلى تنفيذ إجراءات مختلفة بناءً على شروط مختلفة. للقيام بذلك، يمكنك استخدام التعبير الشرطي `if` والمعامل الشرطي `?` الذي يسمى أيضًا "معامل علامة استفهام" (question mark operator، أو المعامل الثلاثي كما سنرى من صياغته).

2.10.1 التعبير الشرطي if

يُقيِّم التعبير الشرطي `if` شرطًا، فإذا تحقَّق `true`، فيُنفِّذ مجموعة من الشيفرات البرمجية. على سبيل المثال:

```
let year = prompt('In which year was ECMAScript-2015 specification published?', '');
if (year == 2015) alert( 'You are right!' );
```

في المثال أعلاه، يكون الشرط عبارة عن فحص بسيط لعملية المساواة `year == 2015`، ولكنه قد يصبح أكثر تعقيدًا. إذا كنت ترغب في تنفيذ أكثر من تعبير واحد، يجب أن تضع الشيفرة المطلوبة داخل الأقواس المعقوصة `{}`:

```
if (year == 2015) {
  alert( "That's correct!" );
  alert( "You're so smart!" );
}
```

من الأفضل استخدام الأقواس المعقوصة `{}` في كل مرة تستخدم فيها التعبير الشرطي `if`، حتى إذا كنت تريد تنفيذ تعبير واحد فقط لأنَّها تُسهِّل قراءة الشيفرة.

2.10.2 التحويل المنطقي

يُقيِّم التعبير الشرطي `if` التعبير الموجود بين القوسين، ثم تُحوَّل النتيجة إلى قيمة منطقية. هل تتذكر قواعد التحويل من الفصل التحويل بين الأنواع؟ إذن، لنتذكرها سويةً:

- يُحوَّل العدد `0`، والسلسلة النصية الفارغة `""`، و `null`، و `undefined`، و`NaN` جميعها إلى القيمة `false`. يُطلَق عليها بسبب ذلك "قيم زائفة خاطئة" (falsy values).
- تُحوَّل القيم الأخرى (أي باستثناء ما سبق) إلى القيمة المنطقية `true`، لذلك يطلق عليها "القيم الصادقة الصحيحة" (truthy values).

لذلك، لن تُنفَّذ الشيفرة البرمجية التالية بناءً على الشرط المعطى:

```
if (0) { // false القيمة 0 تُقيَّم إلى القيمة
  ...
}
```

بينما ستُنفَّذ شيفرة الشرط التالي:

```
if (1) { // true ألقيمة 1 تُقيَّم إلى القيمة
  ...
}
```

يمكنك أيضًا تمرير قيمة منطقية قُيِّمَت مسبقًا إلى الشرط `if` بالشكل التالي:

```
let cond = (year == 2015); // قيمة عملية التحقق من المساواة هي قيمة منطقية
if (cond) {
  ...
}
```

2.10.3 الكتلة الشرطية else

قد يحتوي التعبير الشرطي `if` على كتلة اختيارية تسمى `else` تُنفَّذ عندما يكون الشـرط غـير محقَّق. أي إن تحقق الشرط، فنفِّذ كذا، أو نفِّذ كذا. إليك المثال التالي:

```
let year = prompt('In which year was the ECMAScript-2015 specification published?', '');
if (year == 2015) {
  alert( 'You guessed it right!' );
} else {
  alert( 'How can you be so wrong?' ); // أي قيمة باستثناء 2015
}
```

2.10.4 الشروط المتعددة else if

تود في بعض الأحيان التحقق من العديد من الحالات لشرط ما. التعبير الشرطي `else if` (أو إذا كان كذا، فنفِّذ كذا) تفي بهذا الغرض. اطلع على هذا المثال:

```
let year = prompt('In which year was the ECMAScript-2015 specification published?', '');
if (year < 2015) {
  alert( 'Too early...' );
} else if (year > 2015) {
```

```
    alert( 'Too late' );
  } else {
    alert( 'Exactly!' );
  }
```

في الشيفرة أعلاه، تتحقق JavaScript أولاً من الشرط `year < 2015`. فإذا كان ذلك غير مُحقَّق، فسيتم الانتقال إلى الشرط التالي `year > 2015`. وإذا كان هذا أيضًا غير محقَّق، فستُنفَّذ الكتلة المرتبطة بالفرع `else` أي تُنفَّذ الدالة `alert`. يمكن أن يكون هناك أكثر من فرع `else if`، والكتلة الشرطية الأخيرة `else` اختيارية.

2.10.5 العامل الشرطي ?

تحتاج في بعض الأحيان إلى إسناد قيمة لمُتغيِّر وفقًا لشرط ما. يمكن تحقيق ذلك بالشكل التالي:

```
let accessAllowed;
let age = prompt('How old are you?', '');
if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}
alert(accessAllowed);
```

يتيح لك المعامل الشرطي `?` أو "علامة الاستفهام" القيام بذلك بطريقة أقصر وأبسط.

يُمثَّل هذا العامل بعلامة استفهام `?` ويُطلَق عليه في بعض الأحيان "العامل الثلاثي" (ternary)، لتعامله مع ثلاثة مُعامَلات (operands) وهو العامل الوحيد في JavaScript الذي يحتوي على هذا العدد. الصيغة الخاصة به:

```
let result = condition ? value1 : value2;
```

يتم تقييم الشرط `condition`: فإذا كان محقَّقًا، يُرجِع `value1`، عدا ذلك يُرجِع `value2`.

على سبيل المثال:

```
let accessAllowed = (age > 18) ? true : false;
```

من الناحية التقنية، يمكنك حذف الأقواس الموجودة حول `age > 18`. المعامل الشرطي هذا (أي علامة الاستفهام) له أولوية منخفضة، لذلك يُنفَّذ بعد معامل الموازنة `>`.

يشبه المثال التالي المثال السابق مع اختلاف طفيف جدًا:

```
// يُنفَّذ معامل الموازنة age > 18 أولًا
// (لا حاجة لاستخدام الأقواس)
let accessAllowed = age > 18 ? true : false;
```

لكن الأقواس تجعل الشيفرة أسهل للقراءة، لذلك نوصي باستخدامها.

> في المثال أعلاه، تجنب استخدام عامل علامة الاستفهام لأنَّ عامل الموازنة نفسه يُرجِع `true/false`؛ أي المثال السابق يكافئ:
>
> ```
> let accessAllowed = age > 18;
> ```

2.10.6 العامل الشرطي ؟ المتعدد

يمكن لسلسلة من العاملات الشرطية `?` إرجاع القيمة التي تعتمد على أكثر من شرط واحد. اطلع بتفحُّص على المثال التالي:

```
let age = prompt('age?', 18);
let message = (age < 3) ? 'Hi, baby!' :
  (age < 18) ? 'Hello!' :
  (age < 100) ? 'Greetings!' :
  'What an unusual age!';
alert( message );
```

قد يكون من الصعب فهم الشيفرة السابقة في البداية. ولكن بعد التمعُّن بها قليلًا، يمكنك أن ترى أنَّها مجرد تسلسل عادي من الاختبارات:

1. يتحقق العامل `؟` الأول من الشرط `age < 3`.
2. إذا كان الشرط السابق محقَّقًا، فسيُرجِع `Hi, baby!`؛ خلاف ذلك، فإنَّه يستمر في التحقق من التعبير بعد النقطتين `:`، ويتحقق من الشرط `age < 18`.
3. إذا كان الشرط السابق محقَّقًا، فسيُرجِع `Hello!`؛ خلاف ذلك، فإنه يستمر في التحقق من التعبير بعد النقطتين `:` الثانية، ويَتحقَّق من الشرط `age < 100`.
4. إذا كان الشرط السابق محقَّقًا، فسيُرجِع `Greetings!` خلاف ذلك، فإنه يستمر في التحقق من التعبير بعد النقطتين `:` الأخيرتين، ويتحقق من الشرط `What an unusual age!`.

إليك تنفيذ المثال السابق باستخدام `if..else` فقط:

```
if (age < 3) {
  message = 'Hi, baby!';
} else if (age < 18) {
  message = 'Hello!';
} else if (age < 100) {
  message = 'Greetings!';
} else {
  message = 'What an unusual age!';
}
```

ا. الاستخدام غير التقليدي للعامل ؟

يُستخدَم في بعض الأحيان عامل علامة الاستفهام ? بديلًا عن المعامل الشرطي if بالشكل التالي:

```
let company = prompt('Which company created JavaScript?', '');
(company == 'Netscape') ?
   alert('Right!') : alert('Wrong.');
```

اعتمادًا على الشرط `company == 'Netscape'`، إمَّا أن يُنفَّذ التعبير الأول أو الثاني بعد العامل ? وتُظهِر الدالة alert القيمة الناتجة بناءً على الشرط. على أي حال، لا نوصي باستخدام عامل علامة الاستفهام ? بهذه الطريقة. صحيح أنَّ الشيفرة السابقة أقصر من شيفرة المعامل الشرطي if المقابلة، الذي يُطبِّقه بعض المبرمجين لكنَّه يولد شيفرة صعبة القراءة.

سنعيد كتابة الشيفرة السابقة باستخدام الشرط if:

```
let company = prompt('Which company created JavaScript?', '');
if (company == 'Netscape') {
  alert('Right!');
} else {
  alert('Wrong.');
}
```

عند إلقاء نظرة على الشيفرة للوهلة الأولى، من السهل فهم كتل التعليمات البرمجية التي تمتد عموديًّا لعدة أسطر بشكل أسرع من تلك الأفقية الطويلة.

الغرض من عامل علامة الاستفهام ? هو إرجاع قيمة ما حسب الشرط المُعطى ويُفضَّل استخدامه لذلك الغرض فقط. استخدم المعامل الشرطي if عندما تحتاج لتنفيذ فروع مختلفة من الشيفرة.

2.10.7 تمارين

أ. التعبير الشرطي if (سلسلة نصية مع صفر)

الأهمية: ★★★★★

هل ستُنفَّذ الدالة `alert` داخل الشرط التالي؟

```
if ("0") {
  alert( 'Hello' );
}
```

الحل:

نعم، سوف تُنفَّذ وتظهر الرسالة. الغرض من هذا التمرين هو التذكير بأنَّ أي سلسلة نصية باستثناء الفارغة منها (من ضمنها `"0"` الغير فارغة) تُحوَّل إلى القيمة `true` في السياق المنطقي.

ب. اسم JavaScript

الأهمية: ★★☆☆☆

باستخدام الصيغة `if..else`، اكتب الشيفرة التي تسأل: "ما هو الاسم الرسمي للغة JavaScript؟". إذا أدخل المستخدم "ECMAScript"، تخرج الشيفرة "صحيح!"، وإلا - تُخرج: "ألا تعرف؟ ECMAScript!"

الحل:

```
<!DOCTYPE html>
<html>
<body>
  <script>
    'use strict';
    let value = prompt('ما هو الاسم الرسمي لجافاسكربت؟', '');
    if (value == 'ECMAScript') {
      alert('صحيح!');
    } else {
      alert("ألا تعرف الاسم الرسمي؟ إنه ECMAScript!");
    }
  </script>
</body>
</html>
```

ج. إظهار إشارة

الأهمية: ★★☆☆☆

باستخدام الصيغة `if..else`، اكتب الشيفرة التي تحصل على عدد عن طريق الدالة `prompt` ثم أظهر عبر الدالة `alert` القيمة:

- 1 إذا كان العدد أكبر من صفر.
- 1- إذا كان العدد أقل من صفر.
- 0 إذا كان العدد يساوي الصفر.

في هذا التمرين، نفترض أنَّ القيمة المُدخلة دائمًا عدد.

الحل:

```
let value = prompt('Type a number', 0);
if (value > 0) {
  alert( 1 );
} else if (value < 0) {
  alert( -1 );
} else {
  alert( 0 );
}
```

د. تحويل التعبير الشرطي if إلى صيغة العامل ?

الأهمية: ★★★★★

أعد كتابة التعبير الشرطي `if` باستخدام العامل الثلاثي `?`

```
if (a + b < 4) {
  result = 'Below';
} else {
  result = 'Over';
}
```

الحل:

```
result = (a + b < 4) ? 'Below' : 'Over';
```

ه. تحويل التعبير الشرطي if..else إلى صيغة العامل ?

الأهمية: ★★★★★

أعد كتابة التعبير الشرطي `if..else` باستخدام العامل الثلاثي `?`. لتسهيل قراءة الشيفرة، يوصى بتقسيمها إلى أسطر متعددة.

```
let message;
if (login == 'Employee') {
  message = 'Hello';
} else if (login == 'Director') {
  message = 'Greetings';
} else if (login == '') {
  message = 'No login';
} else {
  message = '';
}
```

الحل:

```
let message = (login == 'Employee') ? 'Hello' :
  (login == 'Director') ? 'Greetings' :
  (login == '') ? 'No login' :
  '';
```

2.11 العاملات المنطقية

هنــــاك ثلاثــــة عــــاملات منطقيــــة في JavaScript وهي: `||` (OR)، و `&&` (AND)، و `!` (NOT).
رغم أنَّها تسمى عاملات منطقية (logical operators)، إلا أنه يمكن تطبيقها على أي نـوع من أنـواع البيانـات وليس فقط على البيانات المنطقية. دعنا نرى التفاصيل.

2.11.1 العامل || (OR) المنطقي

يُمثَّل عامل OR المنطقي بخطين عموديين `||`:

```
result = a || b;
```

في لغات البرمجة القديمة، يعالج المعامل OR المنطقي البيانات المنطقية فقط. إذا كانت أي من وسـائطه arguments تحمـل القيمـة `true`، فـإن المعامـل يُرجـع القيمـة `true`، عـدا ذلـك يُرجـع `false`. ولكن في JavaScript، المعامل أصعب قليلاً وأكثر فائدة، دعنا نرى ما يحدث مع البيانات المنطقية.

هناك أربع مجموعات منطقية محتملة:

```
alert( true || true );   // true
alert( false || true );  // true
alert( true || false );  // true
alert( false || false ); // false
```

كما ترى، النتيجة صحيحة `true` دائمًا باستثناء الحالة الـتي يكـون فيهـا كلا المُعـامَلين (operands) خطـأ `false`. إذا لم يكن المُعامَل منطقيًا، فسيُحوَّل إلى قيمة منطقيـة لتتم عمليـة التقـييم. على سـبيل المثـال، يتم التعامل مع العدد `1` على أنَّه `true`، والعدد `0` على أنَّه `false`:

```
if (1 || 0) { // true || false يقابل التعبير تمامًا
  alert( 'truthy!' );
}
```

غالبًا ما يُستخدَم العامل `||` في الجُمل الشرطية `if` لاختبار مـا إذا كـان أي من الشـروط محقَّق `true`. على سبيل المثال:

```
let hour = 9;
if (hour < 10 || hour > 18) {
  alert( 'The office is closed.' );
}
```

يمكن التحقُّق من عدَّة شروط في الوقت نفسه مثل:

```
let hour = 12;
let isWeekend = true;
if (hour < 10 || hour > 18 || isWeekend) {
  alert( 'The office is closed.' ); //الوقت عطلة
}
```

ا. العامل OR يرجع القيمة الصحيحة الأولى

المنطق الموصوف أعلاه قديم إلى حد ما. لنوضِّح ميزات إضافية في JavaScript. الخوارزمية الموسعة تعمل على النحو التالي، وبالنظر إلى قيم العامل OR المتسَلسَلة في المثال التالي:

```
result = value1 || value2 || value3;
```

يقوم العامل المنطقي || بالتالي:

- يقيم المُعامَلات (operand) من اليسار إلى اليمين.
- يحوِّل جميع المُعامَلات (operands) إلى قيم منطقية إن لم تكن كذلك. إذا كانت النتيجة صحيحة `true`، يتوقف ويرجع القيمة الأصلية لذلك العامل.
- إذا قيِّمَت جميع المُعامَلات وكانت جميعها `false` خطأ، يرجع المُعامَل الأخير.

تُرجَع القيمة بشكلها الأصلي. بمعنى آخر، سلسلة من العامل || تُرجع القيمة الصحيحة الأولى أو القيمة الأخيرة إذا لم يتم العثور على قيمة صحيحة.

على سبيل المثال:

```
alert( 1 || 0 ); // 1 العدد 1 عبارة عن القيمة الصحيحة
alert( true || 'no matter what' ); // (true is truthy)

alert( null || 1 ); // 1 القيمة الصحيحة الأولى هي
alert( null || 0 || 1 ); // 1 القيمة الصحيحة الأولى هي
alert( undefined || null || 0 ); // 0 جميع القيم خاطئة، يرجع القيمة الأخيرة
```

هذه المميزات تؤدي إلى بعض الاستخدامات المثيرة للاهتمام موازنةً بالعامل المنطقي (OR) الكلاسيكي، أو المنطقي فقط.

أولًا، الحصول على أول قيمة صحيحة من قائمة المتغيرات أو التعبيرات

تخيل أن لدينا قائمة من المتغيرات التي يمكن أن تحتوي على بيانات أو تكون فارغة/غير محددة null/undefined. كيف يمكننا العثور على أول قيمة مدخلة من البيانات؟ سنفعل ذلك باستخدام العامل المنطقي ||:

```
let firstName = "";
let lastName = "";
let nickName = "SuperCoder";

alert( firstName || lastName || nickName || "Anonymous"); // SuperCoder
```

إذا آل تقييم كل المتغيرات إلى خطأ، فستكون النتيجة "Anonymous".

ثانيًا، تقييم الطريق الأقصر (Short-circuit)

لا تكون المُعامَلات (operands) قيمًا فقط، بل يمكن أن تكون تعبيرات عشوائية. العامل المنطقي OR يفحصها ويقيِّمها من اليسار إلى اليمين، ويتوقف التقييم عند الوصول إلى أول قيمة صحيحة وتُرجَع تلك القيمة. تسمى هذه العملية تقييم "الطريق الأقصر" أو "الدارة القصيرة" لأنَّها تختصر عملية التقييم قدر الإمكان من اليسار إلى اليمين.

يظهر هذا بوضوح عندما يكون التعبير المعطى كمُعامَل ثانٍ، له تأثير جانبي مثل إسناد متغير. في المثال أدناه، لا يتم إسناد x:

```
let x;
true || (x = 1);
alert(x); // undefined, because (x = 1) not evaluated
```

إذا كان المُعامَل الأول خطأ false، يُقيِّم العامل المنطقي || المُعامَل الثاني، وبالتالي تستمر عملية الإسناد:

```
let x;
false || (x = 1);
alert(x); // 1
```

عملية الإسناد عملية بسيطة. قد تكون هناك آثار جانبية، ولكن لن تظهر إذا كان التقييم لم يصل إليها.

كما نرى، فإن حالة الاستخدام هذه هي "طريقة أقصر للقيام بالمعامل الشرطي`if`". يُحوَّل العامـل الأول إلى قيمة منطقية؛ فإذا كان خطأ، فسيُقيَّم العامل الثاني. في أغلب الأحيان، من الأفضل استخدام المعامل الشرطي `if` بشكله الاعتيادي للحفاظ على سهولة فهم الشيفرة، ولكن لا يكون دائمًا في متناول اليد.

2.11.2 العامل && (AND) المنطقي

يُمثَّل العامل AND المنطقي بعلامتي `&&`:

```
result = a && b;
```

عنـد كتابـة شـيفرة بلغـات البرمجـة القديمـة، يُرجِـع المعامـل AND المنطقي `true` إذا كـان كلا المُعـامَلان صحيحين ويُرجِع `false` عدا ذلك:

```
alert( true && true );   // true
alert( false && true );  // false
alert( true && false );  // false
alert( false && false ); // false
```

مثال مع الشرط `if`:

```
let hour = 12;
let minute = 30;
if (hour == 12 && minute == 30) {
  alert( 'The time is 12:30' );
}
```

تمامًا كما هو الحال مع العامل المنطقي OR، يُسمَح باستعمال أي قيمة مع العامل المنطقي AND:

```
if (1 && 0) { // تقييم كأنها صواب أو خطأ
  alert( "won't work, because the result is falsy" );
}
```

ا. العامل AND يرجع القيمة الخطأ الأولى

بالنظر إلى قيم العامل AND المتعدِّدة في المثال التالي:

```
result = value1 && value2 && value3;
```

يقوم العامل المنطقي AND بالتالي:

- يقيم المُعامَلات من اليسار إلى اليمين.

- يُحوِّل جميع المُعامَلات إلى قيم منطقية. إذا كانت النتيجة `false`، يتوقف ويرجع القيمة الأصلية لذلك المُعامَل.
- إذا قيِّمَت جميع المُعامَلات وكانت جميعها صحيحة، يُرجَع المُعامَل الأخير.

بمعنى آخر، سلسلة من العامل AND تُرجع القيمة الخطأ الأولى أو القيمة الأخيرة إذا لم يُعثَر على أي قيمة خطأ.

القواعد المذكورة أعلاه تشبه قواعد العامل المنطقي OR. الفرق هو أن العامل المنطقي AND يُرجع القيمة الخطأ الأولى بينما يُرجع العامل المنطقي OR القيمة الصحيحة الأولى.

على سبيل المثال:

```
// إذا كان العامل الأول صحيحًا، يرجع
// المعامل المنطقي AND العامل الثاني
alert( 1 && 0 ); // 0
alert( 1 && 5 ); // 5
// إذا كان العامل الأول خطأ، يرجع المعامل
// المنطقي AND هذا العامل ويتجاهل العامل الثاني
alert( null && 5 ); // null
alert( 0 && "no matter what" ); // 0
```

يمكننا أيضا سَلسَلة العديد من القيم على التوالي. انظر كيف تُرجَع أول قيمة خطأ:

```
alert( 1 && 2 && null && 3 ); // null
```

عندما تكون جميع القيم صحيحة، تُرجَع آخر قيمة:

```
alert( 1 && 2 && 3 ); // 3, القيمة الأخيرة
```

ب. أولوية التنفيذ

أولوية العامل المنطقي `&&` في التنفيذ أعلى من أولوية العامل المنطقي `||` لذلك، التعبير التالي `a && b || c && d` هو نفس التعبير `(a && b) || (c && d)` مع الأقواس التي لها الأولية دومًا.

ج. لا تستعمل || أو && مكان الشرط if

يوظف البعض أحيانًا العامل المنطقي `&&` ليحل محل `if` بوصفه طريقًا مختصرًا لكتابة تعبير شرطي. على سبيل المثال:

```
let x = 1;
(x > 0) && alert( 'Greater than zero!' );
```

لن ينفَّذ الإجراء في الجزء الأيمن الخاص بالمعامل المنطقي && إلا إذا وصل التقييم إليه، أي فقط إذا كان x>0 محقَّقًا (صحيحًا). في الحقيقة، يمكننا أيضًا كتابة المثال نفسه بالشكل:

```
let x = 1;

if (x > 0) alert( 'Greater than zero!' );
```

يبدو المثال الأول الذي استعملنا فيه المعامل المنطقي && أقصر، ولكن المثال الثاني الذي استعلمنا فيه if أوضح أضف إلى سهولة قراءة الشيفرة. لذلك، نوصي باستخدام كل معامل للغرض المناسب له: أي يمكنك استخدام التعليمة الشرطية if عند وجود شرط، واستعمل المعامل المنطقي && عند التعامل مع القيم المنطقية.

2.11.3 العامل المنطقي ! (NOT)

العامل المنطقي NOT يُعبَّر عنه بعلامة التعجب ! إذ الصيغة الخاصة به بسيطة للغاية:

```
result = !value;
```

يقبل العامل المنطقي NOT مُعامَلًا واحدًا (operand) ويقوم بما يلي:

- يُحوِّل المُعامَل إلى قيمة منطقية: true/false.
- يرجع القيمة العكسية لتلك القيمة المنطقية.

على سبيل المثال:

```
alert( !true ); // false
alert( !0 ); // true
```

يُستخدَم العامل المنطقي NOT المزدوج (أي !!) في بعض الأوقات لتحويل قيمة معينة إلى قيمة منطقية:

```
alert( !!"non-empty string" ); // true
alert( !!null ); // false
```

بمعنى، يحوِّل العامل المنطقي NOT الأول (أي !"non-empty string") القيمة إلى قيمة منطقية (!true) ويرجع القيمة العكسية (false)، ويرجع العامل المنطقي NOT الثاني

(أي `"non-empty string"!!`) القيمة العكسية مرة أخرى للقيمة التي أعادهـا العامـل الأول (`true` النـاتج النهائي). في النهاية، يوفر لنا المعامل `!!` وسيلةً لتحويل أي قيمة إلى القيمة المنطقية المقابلة لها بسهولة.

هناك طريقة مطوَّلة أكثر قليلاً لفعل الشيء نفسه وذلك باستخدام الدالة `Boolean`:

```
alert( Boolean("non-empty string") ); // true
alert( Boolean(null) ); // false
```

أولوية العامل المنطقي `!` هي الأعلى بين جميع العاملات المنطقية الأخرى، لذلك يُنفَّذ أولًا قبل العامـل `&&` والعامل `||`.

2.11.4 تمارين

ا. ما هي نتيجة العامل OR

الأهمية: ★★★★★

ما ناتج الشيفرة التالية؟

```
alert( null || 2 || undefined );
```

الحل:

الإجابة هي 2، هذه هي القيمة الصحيحة الأولى.

```
alert( null || 2 || undefined );
```

ب. ما هي نتيجة سلسلة من العامل OR للدالة alert

الأهمية: ★★★☆☆

ما ناتج الشيفرة التالية؟

```
alert( alert(1) || 2 || alert(3) );
```

الحل:

الإجابة هي: أولًا 1 ثم 2

```
alert( alert(1) || 2 || alert(3) );
```

استدعاء الدالة `alert` لا يُرجع قيمة. بصيغة أخرى، يرجع `undefined`.

1. يُقيِّم عامل OR الأول `||` المُعامَل الأيسر `alert(1)` وهذا يُظهِر رسالةً تحوي 1.

2. ترجع الدالة `alert` القيمة `undefined`، لذلك ينتقل العامل OR للمُعامَل التالي بحثًا عن قيمة صحيحة.

3. المُعامَل الثاني قيمته 2 أي قيمة صحيحة (يُقيَّم إلى `true`)، وبذلك يتوقف تنفيذ سلسلة العامل OR ويُرجَع المُعامَل 2 ثم يُعرَض بواسطة الدالة `alert`.

لن يظهر الرقم 3، لأنَّ التقييم انتهى عند المُعامَل الثاني و لن يصل إلى `alert(3)`.

ج. ما هي نتيجة العامل AND

الأهمية: ★★★★★

ما ناتج الشيفرة التالية؟

```
alert( 1 && null && 2 );
```

الحل:

الإجابة هي: `null` لأنَّها القيمة الخطأ الأولى في التعبير:

```
alert( 1 && null && 2 );
```

د. ما هي نتيجة سلسلة من العامل AND للدالة alert

الأهمية: ★★★☆☆

ماذا ستُظهر الشيفرة التالية؟

```
alert( alert(1) && alert(2) );
```

الحل:

الإجابة هي: 1 ثم `undefined`:

```
alert( alert(1) && alert(2) );
```

استدعاء الدالة `alert` يُرجِع `undefined` (هي فقط تُظهر رسالة للمستخدم، أي ليس هناك شيء ذا معنى لإعادته). لهذا السبب، يقيِّم العامل `&&` المُعامَل الأيسر (المخرجات 1) ، ويتوقف على الفور، لأنَّ القيمة `undefined` هي قيمة خطأ (أي `false`). والعامل `&&` يبحث عن أول قيمة خطأ ويُرجعُها، فقط.

ه. الناتج من السلسلة (OR AND OR)

الأهمية: ★★★★★

ما الناتج من الشيفرة التالية؟

```
alert( null || 2 && 3 || 4 );
```

الحل:

الإجابة هي: 3.

```
alert( null || 2 && 3 || 4 );
```

أولوية العامل && في التنفيذ أكبر من أولوية العامل ||، لذلك يُنفَّذ أولًا. نتيجة لذلك 2 && 3 = 3، يصبح التعبير:

```
null || 3 || 4
```

الآن، الإجابة هي القيمة الصحيحة الأولى: 3.

و. حصر قيمة متغير ضمن مجال

الأهمية: ★★★☆☆

اكتب شرطًا (أي if) للتحقق من أنَّ قيمة المتغير age محصورة بين 14 و 90 (داخلة ضمن المجال).

الحل:

```
if (age >= 14 && age <= 90)
```

ز. حصر متغير خارج مجال

الأهمية: ★★★☆☆

اكتب شرطًا (أي if) للتحقق من أنَّ قيمة المتغير age لا تقع ضمن 14 و 90 (داخلة ضمن المجال). أنشئ تعبيرين مختلفين: الأول باستخدام عامل النفي !، والثاني دونه.

الحل:

التعبير الأول:

```
if (!(age >= 14 && age <= 90))
```

التعبير الثاني:

```
if (age < 14 || age > 90)
```

ح. سؤال باستخدام التعبير الشرطي if

الأهمية: ★★★★★

أي دالة من الدوال `alert` سوف تُنفَّذ؟ ماذا ستكون نتائج التعبيرات الموجودة في داخل التعبير الشرطي `if(...)`:

```
if (-1 || 0) alert( 'first' );
if (-1 && 0) alert( 'second' );
if (null || -1 && 1) alert( 'third' );
```

الحل:

الإجابة هي: التعبير الأول والثالث والسبب:

```
// يعمل
// ناتج 0 || -1 هو -1، أول قيمة صحيحة
if (-1 || 0) alert( 'first' );

// لا يعمل
// ناتج 0 && -1 هو 0، قيمة خطأ
if (-1 && 0) alert( 'second' );

// يُنفَّذ بالشكل التالي
// المعامل && له أولوية في التنفيذ أكبر من المعامل ||
// يُنفذ أولًا 1 && -1 ثم المعامل ||
// null || -1 && 1  ->  null || 1  ->  1
if (null || -1 && 1) alert( 'third' );
```

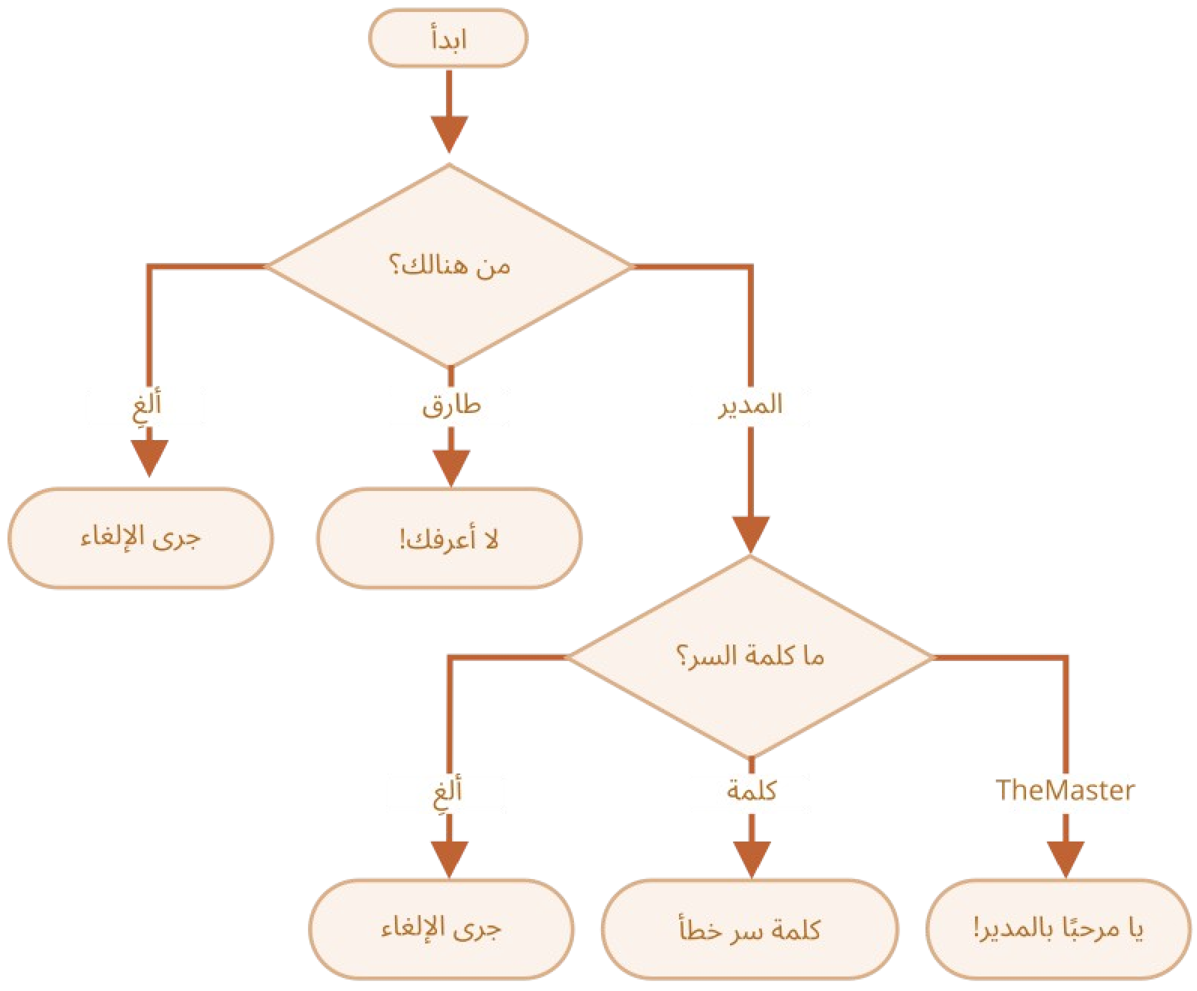
ط. التحقق من تسجيل الدخول

الأهمية: ★★★☆☆

اكتب الشيفرة التي تطلب من المستخدم تسجيل الدخول بواسطة الدالة `prompt`. إذا أدخل المستخدم `Admin`، فاطلب كلمة مرور بواسطة الدالة `prompt`، إذا كان الإدخال عبارة عن سطر فارغ أو مفتاح الهروب Esc - فأظهر "تم الإلغاء"، إذا كان عبارة عن سلسلة نصية أخرى - فأظهر "لا أعرفك".

تُفحَص كلمة المرور على النحو التالي:

- إذا كانت تساوي "TheMaster"، سيُعرَض "مرحبًا!"،
- سلسلة نصية أخرى، سيُعرَض "خطأ"،
- سلسلة نصية فارغة أو "Cancel"، أظهر "إلغاء".



يُرجى استخدام الشرط المتداخل (nested) للمحافظة على سهولة قراءة الشيفرة.

ملاحظة: تمرير قيمة فارغة للدالة `prompt` يُرجِع سلسلة فارغة ''. الضغط على مفتاح الهروب ESC أثناء تنفيذ الدالة `prompt` يُرجِع `null`.

الحل:

```
let userName = prompt("من أنت؟", '');

if (userName == 'Admin') {

  let pass = prompt('ما كلمة المرور؟', '');

  if (pass == 'TheMaster') {
    alert( 'مرحبًا!' );
  } else if (pass == '' || pass == null) {
    alert( 'إلغاء' );
```

```
    } else {
      alert( 'خطأ' );
    }

  } else if (userName == '' || userName == null) {
    alert( 'إلغاء' );
  } else {
    alert( "لا أعرفك" );
  }
```

لاحظ أن المحاذاة الشاقولية أو محاذاة الأقواس داخل الشرط `if` غير مطلوبة من الناحية التقنية، لكنها تجعل الشيفرة سهلة القراءة.

2.12 عامل الاستبدال اللاغي "??"

> هذه إضافة حديثة للغة. لذلك تحتاج بعض المتصفحات القديمة لترقيع هذا النقص لأن ما سنشرحه هو إضافة حديثة للغة.

يوفر عامل الاستبدال اللاغي `??` صيغة قصيرة لاختيار أول متغير مُعرَّف (defined) من القائمة. نتيجة `a ?? b` هي:

- سيُعيد `a` إذا لم تكن فارغة `null` أو غير مُعرَّفة `undefined`،
- وإلا سيُعيد `b`.

إذًا `x = a ?? b` هي اختصار:

```
x = (a !== null && a !== undefined) ? a : b;
```

إليك مثال أطول لتوضيح الأمر.

تخيل أن لدينا مستخدم، وهناك متغيرات `FirstName` و`lastName` أو `nickName` لاسمهم الأول واسم العائلة أو اللقب. ويمكن أن تكون جميعها غير معرفة (undefined)، إذا قرر المستخدم عدم إدخال أي قيمة.

نرغب في عرض اسم المستخدم: في حال أدخل أحد هذه المتغيرات الثلاثة، أو إظهار الاسم "مجهول" إذا لم يُعيَّن أي شيئ. دعنا نستخدم العامل `??` لتحديد أول متغيّر معرّف:

```
let firstName = null;
let lastName = null;
let nickName = "Supercoder";

// سيعرض أول قيمة غير فارغة أو غير معرفة
alert(firstName ?? lastName ?? nickName ?? "Anonymous"); // Supercoder
```

2.12.1 الموازنة باستخدام العامل ||

يمكن استخدام المعامل `||` بنفس طريقة استخدام `??`. في الواقع، يمكننا استعمال `||` مكان `??` في الشيفرة أعلاه والحصول على نفس النتيجة، كما شرحناها في الفصل السابق، المعاملات المنطقية.

الفرق المهم هو أن:

- إن `||` تُعيد القيمة الحقيقية الأولى.
- بينما تُعيد `??` أول قيمة معرّفة.

هذا مهم جدًا عندما نرغب في التعامل مع القيم غير المعرفة أو الفارغة (null/undefined) بطريقة مختلفة عن القيمة 0. فمثلًا، إليك هذه الحالة:

```
height = height ?? 100;
```

تُعيِّن هذه الشيفرة البرمجية المتغير height بالقيمة 100 في حال كان غير مُعرَّف.

دعونا نوازنه بالمعامل ||:

```
let height = 0;

alert(height || 100); // 100
alert(height ?? 100); // 0
```

هنا، height || 100 يعامل الارتفاع الصفري على أنه غير مُعرَّف، تمامًا مثل القيمة الفارغة null أو غير المعرفة undefined أو أي قيمة خاطئة أخرى. إذًا ستكون النتيجة هي 100. أما height ?? 100، يُعيد 100 فقط إذا كان المُتغيِّر height فارغًا null أو غير مُعرَّف undefined. لذلك ستعرضُ الشيفرة السابقة قيمة الارتفاع 0 كما هي.

يعتمد السلوك الأفضل على حالة الاستخدام التي نواجهها. عندما تكون قيمة الارتفاع 0 ونريدها أن تؤخذ بالحسبان، فمن الأفضل استعمال العامل ??.

2.12.2 أولوية عامل ??

إن أولوية العامل ?? منخفضة نوعًا ما: 5 في جدول MDN. إذًا يُقيَّم ?? بعد معظم العمليات الأخرى، ولكن قبل = و ?. إذا احتجنا إلى اختيار قيمة بـ ?? في تعبير معقد، ففكر في إضافة الأقواس، هكذا:

```
let height = null;
let width = null;
// هام: استخدم الأقواس
let area = (height ?? 100) * (width ?? 50);
alert(area); // 5000
```

وإلا إذا حذفنا الأقواس، فإن عملية الضرب * لها أسبقية أعلى معامل ?? وستُنفَذ قبلها.

سيكون ذلك مشابه لهذا المثال:

```
// يمكن أن تكون غير صحيحة
let area = height ?? (100 * width) ?? 50;
```

هناك أيضًا قيود متعلقة باللغة.

استعمال ?? مع && أو ||

لأسباب تتعلق بالسلامة، تحظر JavaScript استخدام العامل ?? مع العامل && والعامل || إلا إذا جرى تحديد أولوية التنفيذ بوضوح عبر استعمال الأقواس.

لاحظ الخطأ في الصياغة الموجود في الشيفرة أدناه:

```
let x = 1 && 2 ?? 3; // Syntax error
```

بالتأكيد إن القيد قابل للنقاش، ولكنه أُضيف إلى المواصفات القياسية للغة بغرض تجنب الأخطاء البرمجية، إذ يبدأ الناس في التبديل من || إلى ??.

استخدم الأقواس الصريحة لتجنب الأمر، هكذا:

```
let x = (1 && 2) ?? 3; // Works

alert(x); // 2
```

2.12.3 الخلاصة

- يوفر عامل الاستبدال اللاغي ?? طريقة مختصرة لاختيار قيمة "مُعرَّفة" من القائمة.
- يستخدم لتعيين القيم الافتراضية للمتغيرات:

```
// أسند القيمة 100 إلى المتغير height إذا كان هذا الأخير فارغًا أو غير معرف
height = height ?? 100;
```

- عامل ?? له أولوية منخفضة جدًا، وأعلى قليلًا من العاملين ? و =.
- يحظر استخدامه مع العاملين || أو && بدون أقواس صريحة.

دورة تطوير واجهات المستخدم

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

2.13 حلقتا التكرار while و for

نحتاج في بعض الأحيان لتكرار إجراء ما مثل إخراج قيم من قائمة واحدة تلو الأخرى أو تشغيل نفس الشيفرة للأعداد من 1 إلى 10 لكل عدد على حدة.

حلقات التكرار (loops) عبارة عن وسيلة لتكرار شيفرة ما عدة مرات.

2.13.1 حلقة التكرار while

الصيغة الخاصة بها:

```
while (condition) {
  // الشيفرة المراد تكرار تنفيذها
  // تدعى جسم الحلقة
}
```

طالما كان الشرط `condition` مُحققًا، أي `true`، تُنفذ الشيفرة الموجودة في جسم الحلقة. على سبيل المثال، تطبع حلقة التكرار أدناه قيمة المتغير `i` طالما كان الشرط `i < 3` مُحققًا:

```
let i = 0;
while (i < 3) { // إظهار 0 ثم 1 ثم 2
  alert( i );
  i++;
}
```

يُسمى التنفيذ الواحد من جسم حلقة التكرار "تكرارًا" (iteration). ينفِّذ المثال السابق ثلاثة تكرارات. إذا كان `i++` غير موجود في المثال أعلاه، ستُكرَّر الحلقة (نظريًا) إلى اللانهاية. أمَّا عمليًا، يوقف المتصفح تكرار مثل هذه الحلقات اللانهائية عند حدٍّ معيَّن، ويمكنك إنهاء العملية أيضًا من طرف الخادم في JavaScript.

شرط حلقة التكرار غير مقصور على تعبيرات الموازنة، بل يمكن أن يكون أي تعبير أو متغير: يُقيّم الشرط ويُحوّل إلى قيمة منطقية بواسطة حلقة التكرار `while`. على سبيل المثال، الطريقة الأقصر لكتابة `while (i != 0)` هي `while (i)`:

```
let i = 3;
while (i) {
  alert( i );
  i--;
}
```

عندما تصل قيمة i إلى الصفر، تقيِّم داخل حلقة while إلى القيمة false وتتوقف آنذاك الحلقة عن العمل.

ليس هناك حاجة للأقواس المعقوصة عند كتابة سطر برمجي واحد

إذا كان جسم حلقة التكرار عبارة عن سطر واحد، يمكنك حذف الأقواس المعقوصة {…}:

```
let i = 3;
while (i) alert(i--);
```

2.13.2 حلقة التكرار do..while

يمكن إزاحة شرط حلقة التكرار إلى أسفل جسم الحلقة باستخدام الصيغة do..while:

```
do {
  // جسم الحلقة
} while (condition);
```

ستُنفذ أولًا الشيفرة الموجودة في جسم الحلقة ثم يتم التحقق من الشرط؛ فإذا كان الشرط مُحققًا، تُكرَّر هذه العملية مرة أخرى. إليك المثال التالي:

```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

يمكنك استخدام هذه الصيغة عندما ترغب في تنفيذ الشيفرة مرة واحدة على الأقل بغض النظر عن كون الشرط محققًا أم لا وعادة ما تُفضل الصيغة الأخرى: while(…) {…}.

2.13.3 حلقة التكرار for

تعد حلقة التكرار for أكثر حلقات التكرار شيوعًا وتصاغ بالشكل التالي:

```
for (begin; condition; step) {
  // ... جسم الحلقة ...
}
```

إليك المثال التالي لتتعرف ماهية هذه الأجزاء begin; condition; step. تُنفِّذ حلقة التكرار أدناه الدالة alert(i) لقيمة المتغيِّر i العددية من 0 إلى 3 (باستثناء العدد 3، لأن الشرط i < 3 لا يشمل العدد 3):

```
for (let i = 0; i < 3; i++) { // إظهار 0 ثم 1 ثم 2
  alert(i);
}
```

يعرض الجدول التالي شرحًا مفصلًا لأجزاء حلقة التكرار السابقة:

الجزء	الشيفرة المقابلة	الوظيفة
البدء (التهيئة)	`i = 0`	يُنفَّذ مرةً واحدةً لحظة ولوج الحلقة
الشرط	`i < 3`	يُتحقَّق من هذا الشرط قبل كل تكرار (دورة تنفيذ) للحلقة، فإذا كان غير محقَّق (`false`)، يوقف تنفيذ الحلقة.
الجسم	`alert(i)`	يُنفَّذ ما دام الشرط محقَّقًا (`true`).
الخطوة	`i++`	يُنفَّذ بعد تنفيذ جسم الحلقة في كل تكرار.

تعمل خوارزمية حلقة التكرار كالتالي:

- دخول الحلقة:
  - -> إذا تحقَّق الشرط -> نفِّذ جسم الحلقة ثمَّ نفِّذ الخطوة
  - -> إذا تحقَّق الشرط -> نفِّذ جسم الحلقة ثمَّ نفِّذ الخطوة
  - -> إذا تحقَّق الشرط -> نفِّذ جسم الحلقة ثمَّ نفِّذ الخطوة
  - -> ...

في حال كنت مبتدئًا، قد يساعدك ذلك في العودة إلى المثال وتدوين آلية تنفيذه خطوة بخطوة على الـورق. إليك شرح لما يحدث في هذه الحالة:

```
// for (let i = 0; i < 3; i++) alert(i)
// دخول الحلقة
let i = 0
// إذا تحقّق الشرط -> نفّذ جسم الحلقة ثمّ نفّذ الخطوة
if (i < 3) { alert(i); i++ }
// إذا تحقّق الشرط -> نفّذ جسم الحلقة ثمّ نفّذ الخطوة
if (i < 3) { alert(i); i++ }
// إذا تحقّق الشرط -> نفّذ جسم الحلقة ثمّ نفّذ الخطوة
if (i < 3) { alert(i); i++ }
```

```
// i == 3 عند 3 < 3 أوقف الحلقة لعدم تحقُّق الشرط
```

ا. التصريح عن المتغيرات داخل نطاق الحلقة

يُصرَّح عن المتغير `i` المسمى "بالعدَّاد" مباشرةً ضمن حلقة التكرار، لـذا تكـون متاحـةً ضـمن نطاقهـا فقـط وغير ظاهرة للعموم.

```
for (let i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
alert(i); // خطأ! لا يوجد متغيِّر بهذا الاسم
```

بدلًا من تعريف متغير جديد، يمكنك استخدام متغير معرَّف مسبقًا:

```
let i = 0;
for (i = 0; i < 3; i++) { // استعمال متغير موجود مسبقًا
  alert(i); // 0, 1, 2
}
alert(i); // يعرض القيمة 3 لوجوده ضمن النطاق العام
```

ب. تجاهل بعض أجزاء التحكم بحلقة التكرار for

يمكن تجاهل أي جزء من أجزاء الحلقة `begin; condition; step` مثل حذف جزء التهيئة `begin` في حال لم يكن وجوده مهمًا كما في المثال التالي:

```
let i = 0; // المتغيِّر جاهز
for (; i < 3; i++) { // فلا حاجة لقسم التهيئة
  alert( i ); // 0, 1, 2
}
```

يمكنك أيضًا حذف الخطوة `step`:

```
let i = 0;
for (; i < 3;) {
  alert( i++ );
}
```

فتصبح حينئذٍ الحلقة `for` مطابقة للحلقة `while (i < 3)`. يمكنك فعليًا إزالة جميع الأجزاء، وخلق حلقة تكرار لانهائية:

```
for (;;) {
  // تكرار لا نهائي
}
```

يرجى التحقق من وجود الفاصلتين المنقوطتين `;` في صيغة حلقة التكرار `for`. خلاف ذلك، سيُطلَق خطأ.

2.13.4 إيقاف حلقة التكرار

يتوقف تنفيذ حلقة التكرار عند عدم تحقُّق الشرط أي أصبح التقييم المنطقي للشرط `false`. مع ذلك، يمكنك إيقاف تنفيذ الحلقة في أي وقت باستخدام التعليمة `break`.

في المثال التالي، تطلب حلقة التكرار من المستخدم إدخال سلسلة من الأرقام عن طريق الدالة `prompt`، ويتوقف تنفيذ الحلقة عندما لا يُدخَل أي رقم:

```
let sum = 0;
while (true) {
  let value = +prompt("Enter a number", '');
  if (!value) break; // (*)
  sum += value;
}
alert( 'Sum: ' + sum );
```

عند النظر إلى الشيفرة، تُنفَّذ الكلمة المفتاحية `break` في السطر `(*)` إذا أدخل المستخدم سطرًا فارغًا أو ألغى عملية الإدخال وبذلك تتوقف حلقة التكرار فورًا، وينتقل تنفيذ الشيفرة إلى السطر الأول بعد حلقة التكرار أي إلى الدالة `alert`. يمكن استعمال حلقة أبدية (لانهائية) مع الكلمة المفتاحية `break` في الحالات التي لا يُعرَّف فيها متى يصبح الشرط غير محقَّق.

2.13.5 الاستمرار في التكرار التالي

التعليمة `continue` هي "نسخة أخف" من التعليمة `break`، إذ لا توقف تنفيذ حلقة التكرار بأكملها بل توقف تنفيذ التكرار الحالي فقط، وينتقل لتنفيذ التكرار التالي (إذا تحقق الشرط طبعًا). أي نستعملها في الحالات التي نرغب فيها بإيقاف تنفيذ التكرار الحالي والانتقال إلى التكرار التالي.

تستخدم حلقة التكرار أدناه التعليمة `continue` لإخراج القيم الفردية فقط من الأعداد 0 وحتى 10:

```
for (let i = 0; i < 10; i++) {
  // إذا تحقق التعبير، تخطى جسم الحلقة وانتقل للتكرار التالي
  if (i % 2 == 0) continue;
  alert(i); // 1, 3, 5, 7, 9
```

```
}
```

فيما يخص القيم الزوجية i، يوقف التعبير البرمجي continue تنفيذ حلقة التكرار وينتقل إلى التكرار التالي في الحلقة for (مع الرقم التالي)، لذلك تظهر الدالة alert فقط القيم الفردية.

ا. تقليل مستوى التداخل عبر التعليمة continue

حلقة التكرار المسؤولة عن إخراج القيم الفردية سوف تبدو كما يلي:

```
for (let i = 0; i < 10; i++) {
  if (i % 2) {
    alert( i );
  }
}
```

من الناحية التقنية، هذا مشابه للمثال أعلاه. يمكنك استخدام الشرط if بدلًا من استخدام التعليمة continue. ولكن سيؤدي ذلك لخلق مستوى إضافي من التداخل (استدعاء الدالة alert داخل الأقواس المعقوصة {}). تقل قابلية القراءة الإجمالية إذا كانت الشيفرة الموجودة داخل التعبير الشرطي if أطول من بضعة أسطر.

ب. لا يُستخدم الموجهان break/continue في المعامل الشرطي الثلاثي ?

لاحظ أنه لا يمكن استخدام البنى التي لا تشبه صياغتها التعابير البرمجية مع المعامل الثلاثي ? تحديدًا تعليمات مثل break/continue.

إليك الشيفرة التالية:

```
if (i > 5) {
  alert(i);
} else {
  continue;
}
```

أعد كتابتها باستخدام المعامل الشرطي ?:

```
(i > 5) ? alert(i) : continue; // لا يسمح باستخدام الموجه continue هنا
```

توقفت الشيفرة عن العمل بسبب خطأ في الصياغة (syntax error)، وهذا سبب آخر لعدم استخدام المعامل ? بدلًا من الشرط if.

2.13.6 تسمية حلقات التكرار

تحتاج في بعض الأحيان لإيقاف حلقات تكرار متداخلة ومتعددة في وقت واحد. على سبيل المثال، تُنفذ حلقتي تكرار في الشيفرة التالية على المتغيرين i و j ، لإخراج الإحداثيات (i, j) من (0,0) إلى (3,3):

```
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    let input = prompt(`Value at coords (${i},${j})`, '');
    // ماذا لو أردت الخروج من الحلقة والانتقال للدالة alert التي تليها؟
  }
}
alert('Done!');
```

ستحتاج إلى طريقة لإيقاف حلقة التكرار إذا ألغى المستخدم الإدخال.

سيُوقف استخدام الموجِّه break بعد input حلقة التكرار الداخلية فقط ولكن هذا ليس كافيًا، فما الحل يا ترى؟! هنا يأتي دور اللافتات (labels)!

اللافتة (label) عبارة عن مُعرِّف (وليست كلمةً محجوزةً) يتبعه نقطتين رأسيتين وتأتي قبل حلقة التكرار مباشرةً:

```
labelName: for (...) {
  ...
}
```

يوقف الموجِّه break <labelName> في المثال أدناه تنفيذه حلقة التكرار ذات المُعرِّف <labelName> (أي outer في حالتنا):

```
outer: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    let input = prompt(`Value at coords (${i},${j})`, '');
    // الخروج من كلتا الحلقتين إن ألغيت عملية الإدخال أو أعطيت قيم فارغة
    if (!input) break outer; // (*)
    // افعل شيئًا بالقيم المعطاة
  }
}
alert('Done!');
```

وظيفة الموجِّه `break outer` في المثال السابق إيقاف حلقة تكرار مسماة بالاسم `outer`؛ لذلك، ينتقل تنفيذ الشيفرة مباشرة من السطر `(*)` إلى السطر `alert('Done!')`.

يمكنك وضع اللافتة في سطر منفصل:

```
outer:
for (let i = 0; i < 3; i++) { ... }
```

يُستخدَم الموجِّه `continue` مع لافتة أيضًا وينتقل تنفيذ التعليمات البرمجية آنذاك إلى التكرار التالي ليس لحلقة التكرار الموجود فيها وإنما للحلقة الموسومة بتلك اللافتة.

لا تسمح اللافتات بالقفز إلى أي مكان في الشيفرة

لا تسمح لك اللافتات بالانتقال إلى أي سطر تريده في الشيفرة لتنفيذه إجباريًا، فمن المستحيل مثلًا تنفيذ المثال التالي:

```
break label;  // label لا يمكن القفز إلى السطر التالي المعنون بـ
label: for (...)
```

لا يمكن استدعاء `break/continue` إلا من داخل حلقة تكرارية ويمكن استعمال لافتة معهما إن سبق عنونة الحلقة بها مسبقًا.

2.13.7 الخلاصة

تعرفت إلى الآن على ثلاثة أنواع من حلقات التكرار:

- `while`: تتحقَّق من الشرط قبل كل تكرار، ويُنفذ التكرار إذا كان الشرط محققًا.
- `do..while`: تتحقَّق من الشرط بعد كل تكرار باستثناء التكرار الأول، ولا يُنفَّذ التكرار الثـاني إذا لم يكن الشرط محققًا.
- `for (;;)`: تتحقَّق من الشرط قبل كل تكرار، وهناك أيضًا عناصر تتيح التحكم أكثر بالحلقة.

لبناء حلقة أبدية (لا نهائية)، تُستخدَم حلقة التكرار `while(true)` (أي أن يكون الشرط دائمًا محققًا)، ويمكن إيقاف هذه الحلقة مثل أي حلقة أخرى، عن طريق الموجِّه `break`.

في حال عدم رغبتك بتنفيذ التكرار الحالي وتود الانتقال إلى التكرار التالي، فيمكنك استخدام الموجِّه `continue`. ويمكنك استخدام الموجهين `break/continue` مع لافتات (labels) شرط أن تكون مُعرَّفة قبل بداية حلقة التكرار. فتوفر اللافتة عندما استعمالها مع `break/continue` تحكمًا أكبر بالحلقات المتداخلة.

2.13.8 التمارين

ا. قيمة حلقة التكرار الأخيرة

الأهمية: ★★★☆☆

ما هي القيمة الأخيرة التي أظهرتها الدالة `alert` في هذه الشيفرة؟ ولماذا؟

```
let i = 3;
while (i) {
  alert( i-- );
}
```

الحل:

الإجابة: 1.

```
let i = 3;
while (i) {
  alert( i-- );
}
```

تَنقُص قيمة المتغير `i` بمقدار 1 في كل تكرار في حلقة التكرار هذه. تتوقف حلقة التكرار `while(i)` عندما يتحقَّق الشرط `i = 0`، وبالتالي تشكل خطوات حلقة التكرار هذه التسلسل التالي:

```
let i = 3;
alert(i--); // إظهار 3 ثم إنقاص قيمة i بمقدار 1 لتصبح 2
alert(i--) // إظهار 2 ثم إنقاص قيمة i بمقدار 1 لتصبح 1
alert(i--) // إظهار 1 ثم إنقاص قيمة i بمقدار 1 لتصبح 0
// توقف الحلقة لعدم تحقق الشرط
```

ب. ما هي القيم التي ستظهرها حلقة التكرار while؟

الأهمية: ★★★★☆

سجّل القيمة الناتجة من كل تكرار في الحلقة، ثم وازنها بالإجابة النهائية. هل تُظهر الدالة `alert` نفس القيم في الحلقتين أم لا؟

- النموذج السابق `++i`:

```
let i = 0;
```

```
while (++i < 5) alert( i );
```

- النموذج اللاحق `i++`:

```
let i = 0;
while (i++ < 5) alert( i );
```

الحل:

يوضح هذا التمرين كيف يمكن أن يؤدي النموذج السابق/ اللاحق (postfix/prefix) إلى نتائج مختلفة عند استخدامهما في الموازنات.

- من 1 إلى 4

```
let i = 0;
while (++i < 5) alert( i );
```

القيمة الأولى هي `i = 1`، لأن معامل الزيادة `++i` الأول يزيد `i` ثم يُرجع القيمة الجديدة. لذلك الموازنة الأولى هي `1 < 5` وتُظهر الدالة `alert` العدد `1`. ثم تتبعها الأعداد 2، 3، 4. تستخدم الموازنة دائمًا القيمة الأخيرة، نظرًا لأن معامل الزيادة `++` قبل المتغير. وأخيرًا، يُزاد `i = 4` إلى `5`، ولا يتحقق شرط حلقة التكرار `while(5 < 5)` في هذه الحالة، وتتوقف الحلقة. لذلك لا يظهر العدد `5`.

- من 1 إلى 5

```
let i = 0;
while (i++ < 5) alert( i );
```

القيمة الأولى هي `i = 1`. يزيد النموذج اللاحق `i++` المتغير `i` ثم يُرجع القيمة القديمة، لذلك ستستخدم الموازنة `i++ < 5` التعبير `i = 0` (على عكس `++i < 5`). لكن استدعاء الدالة `alert` منفصل وتُنفَّذ بعد معامل الزيادة والموازنة، لذلك تحصل على القيمة الحالية للمتغير `i = 1`. ثم تتبعها الأعداد 2، 3، 4. عند `i = 4`، يزيد النموذج السابق `++i` قيمة المتغير ويستخدم العدد `5` في الموازنة ولكن هنا لدينا النموذج اللاحق `i++`. أي أن قيمة المتغير `i` تصبح `5`، لكنه يُرجع القيمة القديمة. وبهذا تصبح الموازنة `while(4 < 5)` صحيحة، وتُنفذ الدالة `alert`. القيمة `i = 5` آنذاك هي القيمة الأخيرة، لأن الشرط في التكرار التالي `while(5 < 5)` غير مُحقق.

ج. ما القيم التي ستظهرها حلقة التكرار for؟

الأهمية: ★★★★☆

سجّل القيمة الناتجة من كل حلقة تكرار، ثم قارنها بالإجابة. هل تُظهر الدالة `alert` نفس القيم في الحلقتين أم لا؟

- النموذج اللاحق `i++`:

```
for (let i = 0; i < 5; i++) alert( i );
```

- النموذج السابق `++i`:

```
for (let i = 0; i < 5; ++i) alert( i );
```

الحل:

الإجابة في كلتا الحلقتين: من 0 إلى 4.

```
for (let i = 0; i < 5; ++i) alert( i );
for (let i = 0; i < 5; i++) alert( i );
```

يمكنك بسهولة استنتاج التالي من الخوارزمية `for`:

1. تُنفذ التعليمة `i = 0` مرة واحدة في البداية.
2. يتم يُتحقَّق من الشرط `i < 5`.
3. إذا كان الشرط محققًا `true`، يُنفذ جسم الحلقة `alert(i)` ويليه معامل الزيادة `i++`.

معامل الزيادة `i++` منفصل عن الشرط في الحالتين، فهو عبارة عن تعليمة أخرى. لا تُستخدَم القيمة التي يعيدها معامل الزيادة هنا، لذلك لا يوجد فرق بين النموذجين `i++` و `++i`.

د. إخراج الأعداد الزوجية باستخدام حلقة التكرار

الأهمية: ★★★★★

استخدم حلقة التكرار `for` لإظهار الأعداد الزوجية من 2 إلى 10.

الحل:

```
for (let i = 2; i <= 10; i++) {
  if (i % 2 == 0) {
    alert( i );
```

```
    }
  }
```

يمكنك استخدام معامل باقي القسمة (modulo) % للحصول على باقي القسمة والتحقق من التكافؤ هنا.

ه. استبدال حلقة التكرار for بحلقة التكرار while

الأهمية: ★★★★★

أعد كتابة الشيفرة باستبدال حلقة التكرار `for` بحلقة التكرار `while` دون تغيير سلوك الشيفرة (يجب أن يظل ناتج حلقة التكرار كما هو).

```
for (let i = 0; i < 3; i++) {
  alert( `number ${i}!` );
}
```

الحل:

```
let i = 0;
while (i < 3) {
  alert( `number ${i}!` );
  i++;
}
```

و. كرر حتى يكون الإدخال صحيحًا

الأهمية: ★★★★★

اكتب حلقة تكرار تطلب من المستخدم إدخال عدد أكبر من 100. إذا أدخل المستخدم عدد آخر، فاطلب منه الإدخال مرة أخرى.

يجب أن تسأل حلقة التكرار عن العدد حتى يُدخل المستخدم عدد أكبر من 100 أو يلغي الإدخال / يُدخل سطرًا فارغًا. في هذا التمرين، يدخل المستخدم الأعداد فقط أي ليس هناك حاجة لتنفيذ معالجة خاصة للتحقق من القيم المدخلة.

الحل:

```
let num;
do {
  num = prompt("Enter a number greater than 100?", 0);
} while (num <= 100 && num);
```

تتكرر حلقة التكرار `do..while` طالما أن الشرط `num <= 100 && num` محقق:

1. الجزء الأول من الشرط `num <= 100`- أي أن القيمة التي أدخلها المستخدم لا تزال أقل من `100`.
2. الجزء الثاني من الشرط `&& num`- لا يتحقق هذا الجزء إذا كان الإدخال `null` أو سلسلة نصية فارغة. تتوقف حلقة التكرار `while` في هذه الحالة أيضًا.

ملاحظة: إذا كان المتغير `num` فارغًا `null`، فسيكون الجزء الأول من الشرط `num <= 100` صحيحًا `true`، دون الجزء الثاني من الشرط لن تتوقف الحلقة إذا نقر المستخدم على زر الإلغاء (CANCEL). لذا لا يمكن الاستغناء عن أي الجزأين.

ز. إظهار الأعداد الأولية

الأهمية: ★★★☆☆

يسمى العدد عددًا أولي (prime) إذا كان عددًا صحيحًا أكبر من `1`، ولا يقبل القسمة (القسمة دون باقي قسمة) إلا على نفسه وعلى العدد `1`. بعبارة أخرى، `n > 1` المتغير `n` عبارة عن عدد أولي إذا كان لا يقبل القسمة بالتساوي على أي عدد باستثناء `1` و `n`. على سبيل المثال، العدد `5` هو عدد أولي، لأنه لا يمكن تقسيمه بالتساوي دون وجود باقي قسمة بمقدار `2`، و `3` و`4`.

اكتب الشيفرة التي تخرج الأعداد الأولية في المجال من `2` إلى `n`. إذا كانت `n = 10` مثلًا، فستكون النتيجة `2,3,5,7`.

ملاحظة: يجب أن تعمل الشيفرة من أجل أي قيمة للمتغير `n`، أي لا يكون المتغير `n` معرّف لقيمة ثابتة.

الحل:

هناك العديد من الخوارزميات لهذا التمرين:

- كتابة الشيفرة باستخدام حلقة تكرار متداخلة:

```
For each i in the interval {
  check if i has a divisor from 1..i
  if yes => the value is not a prime
  if no => the value is a prime, show it
}
```

- كتابة الشيفرة باستخدام اللافتة (label):

```
let n = 10;
nextPrime:
```

```
for (let i = 2; i <= n; i++) { // لكل قيمة من قيم i
  for (let j = 2; j < i; j++) { // ابحث عن المقسوم عليه
    if (i % j == 0) continue nextPrime; // ليس عددًا أوليًا، انتقل للتكرار التالي
  }
  alert( i ); // عددٌ أولي
}
```

هناك مساحة كبيرة لتحسين الشيفرة، على سبيل المثال يمكنك البحث عن القواسم من 2 إلى الجذر التربيعي ل `i` (مثال: القواسم الموجبة للعدد 24 هي 1، 2، 3، 4، 6، 8، 12، 24). على أي حال، إذا كنت تريد تنفيذ التمرين السابق على مجالات كبيرة، فستحتاج إلى تغيير النهج والاعتماد على الرياضيات المتقدمة والخوارزميات المعقدة مثل المنخل التربيعي (Quadratic sieve) أو منخل الأعداد العام (General number field sieve) وما إلى ذلك.

2.14 التعليمة switch

يمكن للتعليمة `switch` أن تحل محل الشرط `if` المُتعدِّد، إذ تمنحك طريقة وصفية أكثر لموازنة قيمة ما مع عدَّة متغيرات.

2.14.1 الصياغة

تحتوي التعليمة `switch` على واحدة أو أكثر من كتل `case` (حالة) وكتلة `default` (حالة افتراضية) أخيرة اختيارية.

الصياغة العامة هي:

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]
  case 'value2':  // if (x === 'value2')
    ...
    [break]
  default:
    ...
    [break]
}
```

- يُتحقَّق من المساواة الصارمة (strict equality) لقيمة المتغير `x` مع القيمة في الحالة الأولى `case` (أي `value1`) ثم الحالة الثانية (أي `value2`) وهلم جرًّا.
- في حال تحقَّقت المساواة الصارمة، يبدأ تنفيذ الشيفرة بدءًا من الكتلة البرمجية `case` المتطابقة حتى أقرب تعليمة خروج `break` (أو حتى نهاية التعليمة `switch` بأكملها).
- في حال لم تتحقق المساواة الصارمة مع أي حالة `case`، فستُنفَّذ كتلة `default` الافتراضيَّة (إن وجدت).

2.14.2 مثال تطبيقي

مثال على التعليمة `switch`:

```
let a = 2 + 2;
switch (a) {
```

```
  case 3:
    alert( 'صغير جدًا' );
    break;
  case 4:
    alert( 'بالضبط!' );
    break;
  case 5:
    alert( 'كبير جدًا' );
    break;
  default:
    alert( "لا أعرف ما هذه القيمة" );
}
```

تبدأ switch هنا في موازنة المُتغيِّر a مع قيمة الحالة case الأولى التي هي 3. لن تتحقق المطابقة في مثالنا لأنَّ قيمة a هي 4 ولكن ستتحقَّق مع قيمة الحالة الثانية، 4. بعد تَحقُّق المطابقة، يبدأ تنفيذ الشيفرة الموجودة بين case 4 وحتى أقرب break.

إذا لم يكن هناك تعليمة break (توقف وخروج)، ستُنفَّذ الحالة case 5 (والحالات اللاحقة) أيضًا دون إجراء عملية التحقُّق.

سنعيد كتابة المثال نفسه دون التعليمة break لترى الفرق:

```
let a = 2 + 2;
switch (a) {
  case 3:
    alert( 'Too small' );
  case 4:
    alert( 'Exactly!' );
  case 5:
    alert( 'Too big' );
  default:
    alert( "I don't know such values" );
}
```

في المثال أعلاه، ستُنفَّذ الدوال alert الثلاث تنفيذًا متسلسلًا وكأننا ننفِّذ الشيفرة التالية:

```
alert( 'Exactly!' );
alert( 'Too big' );
```

```
alert( "I don't know such values" );
```

أي تعبير برمجي يمكن أن يكون وسيطًا للمبدِّل switch/case

يمكنك تمرير تعابير تعسفية إلى `switch/case` مثل:

```
let a = "1";
let b = 0;
switch (+a) {
  case b + 1:
    alert("this runs, because +a is 1, exactly equals b+1");
    break;
  default:
    alert("this doesn't run");
}
```

قيمة التعبير `+a` هنا هي `1` والتي توازن مع قيمة التعبير `b + 1` في `case` لتُنفَّذ آنذاك الشيفرة المقابلة للقيمة المطابقة.

2.14.3 تجميع حالات case متعدِّدة

يمكنك تجميع العديد من الحالات `case` المختلفة لتتشارك الكتلة نفسها المراد تنفيذها عند تطابق إحداها.

على سبيل المثال، إن أردنا تنفيذ الشيفرة نفسها للحالتين `case 3` و `case 5` (أي عندما تكون قيمة `a` هي 3 و 5)، نكتبهما بالشكل التالي:

```
let a = 2 + 2;
switch (a) {
  case 4:
    alert('Right!');
    break;
  case 3:                    // (*) جمع حالتين
  case 5:
    alert('Wrong!');
    alert("Why don't you take a math class?");
    break;
  default:
    alert('The result is strange. Really.');
}
```

الآن، تُظهِر كلًا من الحالة `case 3` والحالة `case 5` الرسالة نفسها عند تطابق إحداهما.

القدرة على تجميع الحالات هي أحد الآثار الجانبية لكيفيَّة تنفيذ switch/case دون الحاجة إلى الفاصل break. يبدأ هنا تنفيذ الحالة 3 case (عند تطابقها) من السطر (*) ويمر عبر الحالة 5 case بسبب عدم وجود التعليمة break.

2.14.4 نوع القيم

دعني أركز على أمر مهم يتعلق بالتحقق من المساواة وهو أنَّ عملية المساواة تكون "صارمة" دومًا. أي يجب أن تكون القيم من النوع نفسه دائمًا لتتساوى.

لتكن لدينا الشيفرة التالية مثلًا:

```
let arg = prompt("أدخل قيمة عددية؟");
switch (arg) {
  case '0':
  case '1':
    alert( 'صفر أو واحد' );
    break;
  case '2':
    alert( 'اثنان' );
    break;
  case 3:
    alert( 'لا تُنفَّذ أبدًا' );
    break;
  default:
    alert( 'قيمة مجهولة' );
}
```

1. عند الحالة 0 والحالة 1، تُنفَّذ الدالة alert الأولى.
2. عند الحالة 2، تُنفَّذ الدالة alert الثانية.
3. لكن عند الحالة 3، ناتج الدالة prompt هي السلسلة النصية "3"، والتي لا تحقق المساواة الصارمة === مع العدد 3، لذا فإنَّ الشيفرة المكتوبة في الحالة 3 case هي شيفرة ميتة ولا تُنفَّذ أبدًا. وستُنفَّذ آنذاك شيفرة الحالة default الافتراضيَّة.

2.14.5 تمارين

أ. أعد كتابة المبدِّل switch بصيغة الشرط if

الأهمية: ★★★★★

اكتب الشرط `if..else` المقابل للمبدِّل `switch` التالي:

```
switch (browser) {
  case 'Edge':
    alert( "لديك المتصفح Edge!" );
    break;
  case 'Chrome':
  case 'Firefox':
  case 'Safari':
  case 'Opera':
    alert( 'حسنًا، نحن ندعم هذه المتصفحات أيضًا!' );
    break;
  default:
    alert( 'نرجو أن تظهر هذه الصفحة بمظهر جيد!' );
}
```

الحل:

لمحاكاة عمل التعليمة `switch` بدقَّة، يجب استخدام المساواة الصارمة `'==='` في الشرط `if`. ويمكنك استخدام المساواة `'=='` أيضًا مع السلاسل النصية المعروفة مسبقًا.

```
if(browser == 'Edge') {
  alert("لديك المتصفح Edge!");
} else if (browser == 'Chrome'
 || browser == 'Firefox'
 || browser == 'Safari'
 || browser == 'Opera') {
  alert( 'حسنًا، نحن ندعم هذه المتصفحات أيضًا!' );
} else {
  alert( 'نرجو أن تظهر هذه الصفحة بمظهر جيد!' );
}
```

ملاحظة: كُتِب التعبير التالي: … `browser == 'Chrome' || browser == 'Firefox'` في الشيفرة أعلاه على أسطر متعدِّدة لتسهيل قراءته.

على أي حال، تبقى صيغة التعليمة `switch` أكثر وضوحًا.

ب. أعد كتابة الشرط if بصيغة المبدِّل switch

الأهمية: ★★★★☆

اكتب المبدِّل switch المقابل للشيفرة التالية:

```
let a = +prompt('a?', '');
if (a == 0) {
  alert( 0 );
}
if (a == 1) {
  alert( 1 );
}
if (a == 2 || a == 3) {
  alert( '2,3' );
}
```

الحل:

تتحول عملية التحقُّق الأولى والثانية إلى حالتين منفصلتين (أي case) بينما تُجمَع عملية التحقق الثالثة في حالتين معًا:

```
let a = +prompt('a?', '');
switch (a) {
  case 0:
    alert( 0 );
    break;
  case 1:
    alert( 1 );
    break;
  case 2:
  case 3:
    alert( '2,3' );
    break;
}
```

التعليمة break في آخر الشيفرة غير مطلوبة، ولكنها تجنبنا الحصول على خطأ في الصياغة مستقبلًا. فقد تود لاحقًا إضافة حالة case أخرى، مثل case4. فإذا نسيت إضافة break قبلها (لأنك لم تكتبها الآن، أي في نهاية case3)، فستواجه خطأ حتميًّا؛ لذلك، هذا نوع من الاحتياط والحذر المسبق من ارتكاب الأخطاء.

2.15 الدوال في JavaScript

قد تحتاج أحيانًا إلى تنفيذ إجراء مماثل في أكثر من موضع في السكربت مثل عرض رسالةٍ جميلةٍ للمستخدم عند تسجيل الدخول وتسجيل الخروج وربما في مكان آخر أيضًا لذا سنحتاج إلى تنفيذ كتل تدعى بالدوال.

الدوال (Functions) عبارةٌ عن كتل برمجيَّة تُنفِّذ مجموعة من المهام وفق تسلسل مُحدَّد، فهي بذلك تُشكل "اللبنات الأساسية" للبرنامج. تسمح الدوال باستدعاء شيفرة ما عدَّة مرات دون الحاجة لكتابتها من جديد.

لقد رأيت خلال الدروس السابقة أمثلةً على دوال مبنيَّة مسبقًا (built-in functions)، مثل `alert(message)`، و `prompt(message, default)`، و `confirm(question)`، ويمكنك أيضًا إنشاء دوال خاصَّة بك.

2.15.1 تعريف الدوال

تُعرَّف الدالة بالصياغة التالية:

```
function showMessage() {
  alert( 'مرحبًا بالجميع!' );
}
```

تأتي الكلمة المفتاحية `function` أولًا، يليها اسم الدالة (`showMessage` في حالتنا)، ثم المعاملات (parameters) التي هي مجموعة من متغيرات تفصل فيما بينها بفاصلة لاتينية `,` (غير موجودة في المثال أعلاه لأنَّها اختياريَّة)، وأخيرًا جسم الدالة (function body) بين الأقواس المعقوصة وهو الإجراء المراد تنفيذه.

```
function name(parameters) {
  // جسم الدالة
}
```

تُستدعَى الدالة بكتابة اسمها متبوعًا بقوسين هلاليين () مثل `showMessage()`:

```
function showMessage() {
  alert( 'مرحبًا جميعًا!' );
}
showMessage();
showMessage();
```

يُنفِّذ الاستدعاء `showMessage()` جسم الدالة أي أنك سترى الرسالة مرتين.

يوضِّح هذا المثال أحد أسباب استخدام الدوال، وهو تجنب تكرار الجمل البرمجية ذاتها. وفي حال احتجت لتغيير الرسالة أو الطريقة التي تُعرَض بها، يكفي أن تُعدِّل الشيفرة من مكان واحد أي من الدالة فقط.

2.15.2 المتغيرات المحلية

المتغير الذي عُرِّف (صُرِّح عنه) داخل حدود دالةٍ ما مرئيٌّ فقط داخل هذه الدالة ويدعى آنذاك "متغيِّرًا محليًّا" (Local variable). إليك المثال التالي:

```
function showMessage() {
  let message = "Hello, I'm JavaScript!"; // متغيِّر محلي
  alert( message );
}
showMessage(); // Hello, I'm JavaScript!
alert( message ); // <-- خطأ! المتغيِّر محلي وموجود ضمن نطاق الدالة فقط
```

2.15.3 المتغيرات العامة

يُعرَّف المتغير العام (outer variable، ويدعى أيضًا global variable) خارج حدود الدالة ويمكنها الوصول إليه أيضًا مثل المتغيِّر userName في المثال التالي:

```
let userName = 'محمد';
function showMessage() {
  let message = 'مرحبًا، ' + userName;
  alert(message);
}
showMessage(); // مرحبًا، محمد
```

تملك الدالة الوصول الكامل إلى المتغير العام، إذ يمكنها التعديل عليه. فيمكننا مثلًا تعديل قيمة المتغيِّر userName العام من داخل الدالة قبل استعماله مثلًا:

```
let userName = 'محمد';
function showMessage() {
  userName = "أحمد"; // (1) تغيير قيمة المتغير العام
  let message = 'مرحبًا، ' + userName;
  alert(message);
}
alert( userName ); // محمد، قبل استدعاء الدالة
showMessage();
```

```
alert( userName ); // أحمد، بعد تعديل الدالة قيمته
```

تستخدم الدالة المتغير العام في حالة واحدة وهي عدم وجود متغير محلي.

إذا عُرِّف متغير محلي داخل دالة يحمل اسم المتغير العام نفسه، يغطي المتغير المحلي على العام ضمن حدود الدالة ويطغى عليه. في الشيفرة أدناه مثلًا، تستخدم الدالة `showMessage()` المتغير `userName` الذي أنشئ محليًّا وتتجاهل المتغير الخارجي العام:

```
let userName = 'محمد';
function showMessage() {
  let userName = "أحمد"; // التصريح عن متغيِّر محلي
  let message = 'مرحبًا، ' + userName; // أحمد
  alert(message);
}
// ستنشئ الدالة المتغيّر المحلي الخاص بها وتستعمله
showMessage();
alert( userName ); // محمد، لم تغير الدالة المتغير العام أو تصل إليه
```

المتغيرات العامة

المتغيرات العامة Global variables هي المتغيرات التي تُعرَّف خارج حدود الدالة وتكون مرئيةً من أي دالة (إلا إن حُجِبَت shadowed بمتغيِّر محلي يحمل الاسم نفسه). يعدُّ الحد من استخدام المتغيرات العامة سلوكًا جيدًا، إذ تحوي الشيفرة الحديثة عددًا قليلًا من المتغيرات العامة أو لا تحتوي عليها إطلاقًا. ومعظم المتغيرات مُعرَّفةٌ داخل دوالها (أي الاقتصار على المتغيرات المحلية). تفيد المتغيرات العامة أحيانًا في تخزين بيانات على مستوى المشروع ككل (project-level data).

2.15.4 المعاملات

يمكنك تمرير أية بيانات إلى الدوال باستخدام "المعاملات" (Parameters، وتسمى أيضًا وسائط الدالة [function arguments]). فتملك الدالة في المثال التالي على معاملين هما: `from` و `text`.

```
function showMessage(from, text) { // الوسائط هي: from و text
  alert(from + ': ' + text);
}
showMessage('مريم', 'مرحبًا!'); // مريم: مرحبًا (*)
showMessage('مريم', "كيف الحال؟"); // مريم: كيف الحال؟ (**)
```

عندما تُستدعَى الدالة في السطر `(*)` والسطر `(**)`، تُنسَخ القيم المُعطاة إلى متغير محلي باسم `from` ومتغير محلي آخر باسم `text` ثم تستخدمها الدالة آنذاك.

إليك مثال آخر مهم أرجو التركيز عليه؛ لدينا متغيِّرًا باسم `from` مرَّرناه إلى الدالة `showMessage()`. لاحظ أن التعديل على المتغير `from` مرئيٌّ فقط داخل الدالة ولا ينعكس خارجها، وذلك لأنَّ الدالة تحصل دائمًا على نسخة من قيمة المتغيِّر ثم تتركه وشأنه:

```
function showMessage(from, text) {
  from = '*' + from + '*'; // إظهار "from" بمظهر مختلف
  alert( from + ': ' + text );
}
let from = "مريم";
showMessage(from, "مرحبًا"); // *مريم*: مرحبًا
// لا تتغير قيمة "from" لأن الدالة عدَلت على متغير محلي مسمًى باسمه
alert( from ); // مريم
```

2.15.5 القيم الافتراضية

إن لم تُمرَّر أية قيمة لمعاملات دالةٍ، تُصبِح قيمها آنذاك غير مُعرَّفة `undefined`. فيمكن استدعاء الدالة `showMessage(from, text)` التي ذكرناها مسبقًا مع تمرير قيمةٍ واحدةٍ لها مثل:

```
showMessage("مريم");
```

لا يُعدُّ هذا خطأً بل يُظهِر القيمة `"مريم: undefined"`. لمَّا لم تُعطَ قيمةٌ للمعامل `text`، فسيُفترَّض أن `text === undefined` (أي يُعطَى القيمة `showMessage`). إذا أردت إسناد قيمة افتراضية للمعامل `text`، فيمكنك تحديدها عبر معامل الإسناد `=` أثناء تعريف الدالة بالشكل التالي:

```
function showMessage(from, text = "لم تُعطَ قيمة لـ text") {
  alert( from + ": " + text );
}
showMessage("مريم"); // مريم: لم تُعطَ قيمة لـ text
```

إن لم تُمرَّر الآن قيمةٌ للمعامل `text` عند استدعاء الدالة، فستُسنَد له القيمة "لم تُعطَ قيمة لـ text" التي هي عبارة عن سلسلة نصية. ويمكن أيضًا أن تكون القيمة الافتراضية تعبيرًا معقَّدًا يُقيَّم ثم تُسنَد القيمة الناتجة إليه إذا، وفقط إذا، لم تُعطَ قيمة لذلك المعامل؛ لذلك، الشيفرة التالية صحيحةٌ أيضًا:

```
function showMessage(from, text = anotherFunction()) {
  // تُنفَّذ الدالة anotherFunction() إذا لم تُمرَّر قيمة للمعامل text
  // وتصبح القيمة التي تعيدها هي قيمة المعامل text الحالية آنذاك
}
```

أ. تقييم المعاملات الافتراضية

يُقيّم المعامل الافتراضي في JavaScript في كل مرة تُستدعَى فيها الدالة دون المعامل المقابل له. ففي المثال أعلاه، تُستدعَى الدالة `anotherFunction()` في كل مرة تُستدعَى فيها الدالة `showMessage()` دون المعامل `text`.

ب. الطراز القديم للمعاملات الافتراضية

لم تكن الإصدارات القديمة من JavaScript تدعم المعاملات الافتراضية، ولكن كان هناك طرائق بديلة لدعمها، قد تمر معك عند مراجعتك لسكربتات قديمة مثل التحقق الصريح من كون المعامل غير مُعرَّف بالشكل التالي:

```
function showMessage(from, text) {
  if (text === undefined) {
    text = 'no text given';
  }
  alert( from + ": " + text );
}
```

أو عن طريق استعمال المعامل `||` بالشكل التالي:

```
function showMessage(from, text) {
  // إن كانت قيمة المعامل text غير معطاة (أي غير معرفة)، فستُستعمَل القيمة الافتراضية
  text = text || 'لم تُعطَ أية قيمة';
  ...
}
```

2.15.6 إرجاع قيمة

يمكن للدالة إرجاع قيمة معنيَّة إلى من استدعاها. وأبسط مثال على ذلك دالةٌ تجمع قيمتين ثم تعيد القيمة الناتجة:

```
function sum(a, b) {
  return a + b;
}
let result = sum(1, 2);
alert( result ); // 3
```

يمكن أن يقع الموجه `return` في أي مكان داخل الدالة، ولكن انتبه لأن تنفيذ الدالة يتوقف عند الوصول إليه، وتُعَاد القيمة (أو ناتج تقييم تعبير برمجي) التي تليه إلى من استدعاها (`result` أعلاه). وقد تحوي دالةٌ واحدةٌ عدَّة موجهات `return` مثل:

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('هل تملك إذنًا من والديك؟');
  }
}
let age = prompt('كم عمرك؟', 18);
if ( checkAge(age) ) {
  alert( 'سُمِح له/ا بالوصول' );
} else {
  alert( 'مُنِع/ت من الوصول' );
}
```

يمكن إنهاء تنفيذ الدالة فورًا عن طريق استخدام `return` دون قيمة مثل:

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }
  alert( "يبدأ عرض الفلم" ); // (*)
  // ...
}
```

في الشيفرة أعلاه، إذا كان التعبير `checkAge(age)` يُرجع القيمة `false`، فلن تُظهر عندئذٍ الدالة `alert` السلسلة النصية `showMovie`.

ا. إرجاع قيمة غير مُعرَّفة

تُرجع دالةٌ القيمة `undefined` إن لم يلي الموجه `return` أية قيمة أو تعبير أو لم يكن موجودًا (أي لا تعيد الدالة أي شيء):

```
function doNothing() { /*  */ }
alert( doNothing() === undefined ); // true
```

بعبارة أخرى، حالة عدم إرجاع الدالة أي شيء وحالة استعمال `return;` فقط وحالة استعمال `return undefined;` كلها سواسية وهي أنَّ الدالة تعيد القيمة `undefined`:

```
function doNothing() {
  return;
}
alert( doNothing() === undefined ); // true
```

ب. إضافة سطر جديد بين الموجه return والقيمة المعادة

لا تضف سطرًا جديدًا بين الموجه `return` والقيمة (أو التعبير) التي يفترض أن تعيدها الدالة. فعند وجود تعبير طويل يلي `return`، قد تضعه في سطر منفصل مثل السطر التالي:

```
return
 (some + long + expression + or + whatever * f(a) + f(b))
```

ولكنَّ هذه الشيفرة لا تعمل، لأنَّ JavaScript تفترض من تلقاء نفسها وجود الفاصلة المنقوطة بعد `return`. أي أن هذه الشيفرة ستُقيَّم كما يلي:

```
return;
 (some + long + expression + or + whatever * f(a) + f(b))
```

كما ترى، تصبح `return` فعليًّا فارغة ولا تُرجع الدالة أية قيمة؛ لذلك، يجب أن تكون القيمة المراد إرجاعها والموجه `return` على السطر نفسه. أي إن أردت وضع التعبير المراد إعادته على عدة أسطر، فيجب كتابة بدايته على السطر نفسه الموجود عليه الموجه `return` أو وضع قوس هلالي افتتاحي على أقل تقدير كما يلي:

```
return (
  some + long + expression
  + or +
  whatever * f(a) + f(b)
  )
```

وستعمل الشيفرة بالشكل المتوقع لها.

2.15.7 تسمية الدوال

الدوال عبارة عن إجراءات، لذلك من الأفضل تسميتها بفعل مختصر ودقيق قدر الإمكان يصف ما تقوم به ليسهل على قارئ الشيفرة الغريب فهم عملها.

أحد ممارسات التسمية الشائعة استعمال فعل مصدري ابتدائي في بداية اسم الدالة لوصف الإجراء الرئيسي الذي تُنفِّذه باختصار. ومن الأفضل أن يتفق أعضاء الفريق على معنى أسماء هذه الأفعال البادئة المستخدمة. فالدوال التي تبدأ تسميتها بالفعل "show" مثلًا عادةً ما تُظهِر شيئًا ما.

الدوال التي تبدأ عادةً بالفعل:

- "get…" تجلب قيمة.
- "calc…" تحسب شيئًا ما.
- "create…" تنشئ شيئًا ما.
- "check…" تفحص شيئًا ما وتُرجع قيمة منطقية وهلم جرًّا.

وإليك أمثلة على هذه الأسماء:

```
showMessage(..)     // إظهار رسالة
getAge(..)          // جلب العمر (بطريقة ما)
calcSum(..)         // حساب المجموع وإعادة الناتج
createForm(..)      // إنشاء إستمارة (وإعادتها غالبًا)
checkPermission(..) // التحقق من إذنٍ وإعادة قيمة منطقية
```

يتبادر لذهنك سريعًا ماهية العمل الذي تنفِّذه الدالة ونوع القيمة التي تُرجعها عند استخدام البادئات في تسمية الدوال استخدامًا ملائمًا وصحيحًا.

ا. دالة واحدة مقابل إجراء واحد

يجب أن تُنفذ الدالة ما أشار لها اسمها بالضبط، لا أكثر ولا أقل. فعندما تُريد تنفيذ إجراءين مستقلين، يجب عليك تعريف دالتين مستقلتين، حتى لو جرى استدعاؤهما معًا (يمكنك في هذه الحالة تعريف دالة ثالثة تستدعي هاتين الدالتين).

بعض الأمثلة لكسر هذه القاعدة:

- getAge سيكون سيئًا إذا أظهرت الدالة العمر بتنبيه (يجب أن تجلب العمر وتعيده فقط).
- createForm سيكون سيئًا إذا عدَّلت الصفحة وأضافت الاستمارة إليها (يجب أن تنشئه وأن تعيده فقط).
- checkPermission سيكون سيئًا إذا عرضت رسالةً تشرح حالة الوصول (يجب أن تفحص الإذن وأن تعيد النتيجة فقط).

تعتمد هذه الأمثلة على المعاني الشائعة لتلك البادئات. ولك مطلـق الحريـة أنت وفريقـك في الاتفـاق على معانٍ أخرى، لكن عادةً لا تختلف كثي7رًا عما هو شائع لها. على أي حال، يجب أن يترسـخ لـديك فهمٌ لمـا تعنيـه كل بادئة وما الأشياء التي يمكن أو لا يمكن للدوال الملحق اسمها ببادئة القيام بها. وأخيرًا، ينبغي للفريـق تبـادل هذا العُرف الموضوع.

ب. أسماء دوال قصيرة جدًا

الدوال التي تُستخدَم بكثرة تملـك أحيانًـا أسـماءً قصـيرة جـدًا مثـل الدالـة $ المُعرَّفـة في المكتبـة jQuery والدالة _ الأساسية الخاصة بالمكتبة Lodash. تعد تلك الحالات استثنائية، ولكن يجب عمومًـا أن تكـون أسـماء الدوال وصفية وموجزة.

2.15.8 الدوال == التعليقات

يجب أن تكون الدوال قصيرة وتُنفذ إجراءً واحدًا فقط. إذا كان هذا الإجراء معقد، فيفضَّـل تقسـيم الدالـة إلى عدد من الدوال البسيطة. قد لا يكون اتباع هذه القاعدة في بعض الأحيان سهل، لكنه بالتأكيد أمر جيد.

الدالة المنفصلة ليست سهلة الاختبار والتنقيح فقط بل تُعدُّ تعليقًـا عظيم الشـأن! فمثلًا وازن بين الـدالتين showPrimes(n) أدناه، إذ تخرج كل واحدة منهما الأعداد الأولية حتى العدد n المعطى.

الدالة في المثال الأول تستخدم لافتة:

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {
    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }
    alert( i ); // عدد أولي
  }
}
```

الدالة في المثال الثاني تستخدم الدالة الإضافية isPrime(n) لاختبار الأعداد إذا كانت أولية أم لا:

```
function showPrimes(n) {
  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;
    alert(i);  // a prime
  }
}
function isPrime(n) {
```

```
    for (let i = 2; i < n; i++) {
      if ( n % i == 0) return false;
    }
    return true;
  }
```

لاحظ أن المثال الثاني سهل الفهم، أليس كذلك؟ فنرى بدلًا من قطعة مبعثرة من الشيفرة اسم إجراءٍ (الذي هو `isPrime` في مثالنا). يشير الأشخاص أحيانًا إلى هذه الشيفرة بأنها "ذاتية الوصف" أي تصف نفسها بنفسها. بناءً على ذلك، يمكنك إنشاء الدوال حتى إذا لم ترد إعادة استخدامها وذلك لتنظيم الشيفرة وتسهيل قراءتها.

2.15.9 الخلاصة

صياغة تعريف دالة ما يبدو بالشكل التالي:

```
function name(parameters, delimited, by, comma) {
  /* الإجراء الذي تنفذه الدالة */
}
```

- تُنسَخ القيم التي تُمرَّر إلى الدالة على أنها معاملات إلى متغيرات محلية تستخدمها الدالة.
- تستطيع أي دالة الوصول إلى المتغيرات العامة الخارجية، لكن أولوية الاستخدام تكون للمتغيرات المحلية ثم المتغيرات العامة (إن لم تتوفر المحلية). لا تملك الشيفرة التي تقع خارج حدود الدالة وصولاً إلى متغيراتها المحلية.
- يمكن للدالة إرجاع أي قيمة، أو القيمة `undefined` إن لم تُرجع أي شيء.

يوصى باستخدام المتغيرات المحلية والمعاملات بشكل أساسي في الدالة وذلك لجعل الشيفرة واضحة وسهلة الفهم، وتجنب استخدام المتغيرات العامة قدر الإمكان.

فهم عمل دالة تحتوي على معاملات وتستعملها ثم تُرجع قيمةً أسهل من فهم دالة لا تحتوي على أي معاملات، ولكنها تستعمل المتغيرات العامة وتعدل عليها.

تسمية الدالة:

- يجب أن يصف اسم الدالة الإجراء الذي تُنفِّذه بوضوح. عندما ترى استدعاءً لدالة في شيفرةٍ ما، فإن الاسم الجيد يساعدك على فهم ما ستُنفذه وما ستُرجعه على الفور.
- الدالة عبارة عن إجراء (فعل)، لذلك عادة ما تستعمل الأفعال المصدرية في تسمية الدوال.

- يوجد العديد من البادئات المشهورة في إلحاقها بأسماء الـدوال مثـل …create، و …show، و …get، و …check تُستخدَم للإشارة لما ستُنفذه هذه الدوال.

الدوال هي اللبنات الأساسية في بناء الشيفرة. أصبحت الآن تملك فهمًا جيـدًا للأساسـيات ويمكنـك البـدء في إنشائها واستخدامها. سنطبق في الدروس القادمة كل ما تعملناه ونعود لـه مـرارًا وتكـرارًا والغـوص أكـثر في مواضيع متقدمة.

2.15.10 التمارين

ا. هل وجود التعبير البرمجي else مهم في الشيفرة؟

الأهمية: ★★★★☆

ترجـع الدالـة التاليـة القيمـة true إذا كـانت قيمـة المعامـل age أكـبر من 18 وإلا فإنهـا تطلب تأكيـدًا ثم ترجع نتيجته:

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    // ...
    return confirm('هل أخذت إذن والديك؟');
  }
}
```

هل سيتغير عمل الدالة إذا حُذف التعبير else؟

```
function checkAge(age) {
  if (age > 18) {
    return true;
  }
  // ...
  return confirm('Did parents allow you?');
}
```

هل يوجد اختلاف في سلوك هذين المثالين؟

الحل:

لا يوجد أي اختلاف.

ب. أعد كتابة الدالة باستخدام المعامل ? أو المعامل ||

الأهمية: ★★★★☆

ترجع الدالة التالية القيمة `true` إذا كانت قيمة المعامل `age` أكبر من `18` وإلا فإنها تطلب تأكيدًا وتُرجع نتيجته:

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('هل يأذن لك والديك؟');
  }
}
```

أعد كتابة الدالة السابقة بدون استخدام التعبير الشرطي `if` لتنفيذ الإجراء نفسه الموضح أعلاه في سطر واحد.

اكتب نموذجين مختلفين للدالة `checkAge`، وليكن النموذج الأول باستخدام المعامل الشرطي علامة الاستفهام `?`، والثاني باستخدام المعامل `||`.

الحل:

النموذج الأول باستخدام المعامل الشرطي علامة الاستفهام `?`:

```
function checkAge(age) {
  return (age > 18) ? true : confirm('هل يأذن لك والديك؟');
}
```

النموذج الثاني باستخدام المعامل `||` (النموذج الأبسط):

```
function checkAge(age) {
  return (age > 18) || confirm('هل يأذن لك والديك؟');
}
```

لاحظ أن الأقواس حول التعبير `age > 18` ليست مطلوبة هنا لكن وجودها يُسهِّل قراءة الشيفرة.

ج. الدالة min(a,b)

الأهمية: ★☆☆☆☆

اكتب الدالة `min(a,b)` التي ترجع قيمة العدد الأصغر من العددين `a` و `b` المعطيين. أي دالة مثل:

```
min(2, 5) == 2
min(3, -1) == -1
min(1, 1) == 1
```

الحل:

النموذج الأول باستخدام التعبير الشرطي `if`:

```
function min(a, b) {
  if (a < b) {
    return a;
  } else {
    return b;
  }
}
```

النموذج الثاني باستخدام المعامل الشرطي علامة الاستفهام `?`:

```
function min(a, b) {
  return a < b ? a : b;
}
```

في حال كان العددان متساويين `a == b`، لا فرق إذا أُرجِع الأول أم الثاني.

د. الدالة pow(x,n)

الأهمية: ★★★★☆

اكتب الدالة `pow(x,n)` التي تُرجع قيمة العدد `x` مرفوع للقوة `n`. بعبارة أخرى، تُرجع ناتج ضرب العدد `x` في نفسه عدد `n` من المرات.

```
pow(3, 2) = 3 * 3 = 9
pow(3, 3) = 3 * 3 * 3 = 27
pow(1, 100) = 1 * 1 * ...* 1 = 1
```

أنشئ صفحة ويب باستخدام الدالة `prompt` تسمح للمستخدم إدخال العدد `x` والعدد `n`، ثم تُظهر نتيجة الدالة `pow(x,n)`.

ملاحظة: في هذا التمرين، يجب أن يكون العدد `n` من الأعداد الطبيعية الصحيحة وأكبر من العدد `1` أيضًا.

الحل:

```
function pow(x, n) {
  let result = x;
  for (let i = 1; i < n; i++) {
    result *= x;
  }
  return result;
}
let x = prompt("x?", '');
let n = prompt("n?", '');
if (n < 1) {
  alert(`القوة ${n} غير مدعومة، لذا أدخل عددًا صحيحًا أكبر من الصفر`);
} else {
  alert( pow(x, n) );
}
```

2.16 تعابير الدوال

الدالة ليست بنية سحرية تُبنَى بها الشيفرة في JavaScript، وإنما هي نوع خاص من القيم. يُطلق على الصياغة التي تُستخدم في بناء الدوال "التصريح عن دالة" (Function Declaration):

```
function sayHi() {
  alert( "مرحبًا" );
}
```

هناك صياغة أخرى لبناء دالة تسمى "تعبير دالة" (function expression) وتبدو بالشكل التالي:

```
let sayHi = function() {
  alert( "مرحبًا" );
};
```

تُعامل الدالة في هذه الصياغة مثل أي قيمة، إذ تُنشَأ ثم تُسنَد للمتغير `sayHi` كما توضح الشيفرة. بغض النظر عن الصياغة التي تُعرَّف بها الدالة، فهي مُجرَّد قيمة مُخزَّنة في المتغير `sayHi`. المراد من هذه الشيفرة هو نفسه في الشيفرة التي تسبقها: "إنشاء دالة ووضعها في المتغير `sayHi`".

يمكن عرض الدالة ككل باستخدام `alert` دون الأقواس:

```
function sayHi() {
  alert( "Hello" );
}
alert( sayHi ); // إظهار شيفرة الدالة
```

لاحظ أنَّ السطر الأخير في الشيفرة لا يستدعي الدالة لتنفيذها، وذلك بسبب عدم وجود الأقواس بعد اسمها `sayHi`. العديد من لغات البرمجة لا تتأثر بذلك (أي لا تهتم بوجود هذه الأقواس)، ولكن avaScript ليست منها.

الدالة في JavaScript عبارة عن قيمة، لذا يمكن التعامل معها على أنها قيمة. يُظهِر الاستدعاء `alert( sayHi );` سلسلة نصية تمثِّل شيفرة الدالة المُعرَّفة بأكملها، أي الشيفرة المصدرية للدالة.

الدالة عبارة عن قيمة خاصة، وأحد الأمور الخاصَّة التي تنفرد بها عن القيم العادية هي إمكانية استدعائها وتنفيذها وذلك بوضع الأقواس بعد اسمها بالشكل `sayHi()` في أي مكان داخل الشيفرة. على أي حال، الدالة تبقى قيمة ولا مشكلة في معاملتها مثل أنواع القيم الأخرى. فيمكنك مثلًا إسناد دالة إلى متغير آخر:

```
function sayHi() {   // (1) إنشاء
  alert( "مرحبًا" );
```

```
}
let func = sayHi;     // (2) نسخ
func(); // مرحبًا      // (3) تنفيذ النسخة المنسوخة عن الدالة
sayHi(); // مرحبًا     //      وهذا يعمل أيضًا ولِمَ لا!
```

إليك تفصيلٌ لما تنفِّذه الشيفرة السابقة:

1. ينشئ التصريح عن الدالة في السطر `(1)`الدالة ثمَّ يضعها في المتغير `sayHi`.
2. تُنسَخ الدالة في السطر `(2)` إلى المتغير `func`. لاحظ هنا عدم وجود أقواس استدعاء الدالة بعد `sayHi`. إذا أُضيفَت هذه الأقواس فإنَّ الاستدعاء `func = sayHi()` سوف يُسنِد الناتج الذي تعيده الدالة إلى المتغير `func` وليس الدالة `sayHi` نفسها.
3. يمكنك الآن استدعاء الدالة عبر استدعاء `sayHi()` أو `func()`.

يمكنك أيضًا استخدم تعبير الدالة للتصريح عن الدالة `sayHi` في السطر الأول:

```
let sayHi = function() { ... };
let func = sayHi;
// ...
```

يولد كلا التعبيرين النتيجة نفسها.

ا. وجود الفاصلة المنقوطة في نهاية تعبير الدالة

قد تتساءل عن سبب وجود الفاصلة المنقوطة في نهاية "تعبير الدالة"، وعدم وجودها في نهاية "تعريف الدالة":

```
function sayHi() {
  // ...
}
let sayHi = function() {
  // ...
};
```

الجواب بسيط:

- ليس هناك حاجة للفاصلة المنقوطة `;` في نهاية كتل التعليمات البرمجية والصِيغ التي تستخدمها مثل `if { ... }`، و `function f { }`، و `for { }`...إلخ.

- يُعدُّ تعبير الدالة، `let sayHi = ...;`، تعليمة برمجية (statement) ويُستخدَم على أنَّه قيمة أي هو ليس كتلة برمجية (code block) ولكنه عملية إسنادٍ لقيمة. تكتب الفاصلة المنقوطة `;` في نهاية التعليمات البرمجية، بغض النظر عن ماهية هذه التعليمات؛ لذلك، لا ترتبط الفاصلة المنقوطة بتعبير الدالة نفسه إطلاقًا، وإنها فقط تُنهِي التعليمة البرمجية.

2.16.2 دوال رد النداء

إليك مزيدًا من الأمثلة حول تمرير الدوال على أنَّها قيم واستخدام تعابير الدوال.

سنكتب الدالة `ask(question, yes, no)` التي تقبل تمرير ثلاثة معاملات إليها:

- المعامل `question`: يمثِّل نص السؤال.
- المعامل `yes`: يُمثِّل دالة يراد تنفيذها إذا كانت إجابة المعامل `question` هي `yes`.
- المعامل `no`: يمثِّل دالة يراد تنفيذها إذا كانت إجابة المعامل `question` هي `no`.

تطرح الدالة السؤال `question`، وبناءً على إجابة المستخدم، تُستدعى الدالة `yes()` أو الدالة `no()`:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
function showOk() {
  alert( "لقد وافقت!" );
}
function showCancel() {
  alert( "لقد ألغيت التنفيذ." );
}
// ask تمرر الدالتين showCancel و showOk على أنها وسائط إلى الدالة
ask("Do you agree?", showOk, showCancel);
```

مثل هذه الدوال مفيدةٌ للغاية في الحياة العملية. ويتمثَّل الاختلاف الرئيسي بين التنفيذ الواقعي العملي والمثال النظري أعلاه في أنَّ الأولى تستخدم طرائق أكثر تعقيدًا للتفاعل مع المستخدم من مجرد دالة `confirm` بسيطة. تُظهِر مثل هذه الدالة نافذة أسئلة مُعدَّلة جميلة المظهر في المتصفح لكن هذا موضوع آخر متقدم.

يسمى الوسيطين `showOk` و `showCancel` للدالة `ask` "بدوال ردود النداء" (callback functions) أو "ردود النداء" (callbacks) فقط.

الفكرة القابعة خلف "رد النداء" هي تمرير دالة ثم توقُّع إعادة استدعائها لاحقًا إذا لزم الأمر. أي مثلها كَمَثِل ارتداد صدى الصوت وعودته للمنادي أو مناداة أحدهم بطلبِ تنفيذ أمرٍ وتوقع إجابة النداء (رد النداء) بنتيجة الفعل وغيرها من الأمثلة المستقاة من الواقع.

في المثال السابق، تصبح الدالة showOk رد نداء للإجابة "نعم" (yes)، والدالة showCancel رد نداء للإجابة "لا" (no). يمكن استخدام تعبير دالة لكتابة الدالة نفسها بشكل أقصر:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
ask(
  "هل تقبل؟",
  function() { alert("لقد قبلت."); },
  function() { alert("لقد ألغيت التنفيذ."); }
);
```

تُعرَّف الدوال هنا مباشرةً داخل الدالة ask(...) ولا تملك اسمًا في هذه الحالة، لذا يُطلَّق عليها "دوال مجهولة" (anonymous). لا يمكن الوصول إلى مثل هذه الدوال من خارج الدالة ask(...) (لعدم إسنادها إلى متغيرات) ولا يهمنا هذا الأمر هنا.

الدالة عبارة عن قيمة تمثل "إجراء"

القيم العادية مثل السلاسل النصية أو الأعداد تمثِّل البيانات ولكن يمكن اعتبار الدالة -بما أنها قيمة خاصة- على أنَّها إجراء، لذا يمكنك تمريرها بين المتغيرات وتنفيذها عند الحاجة.

2.16.3 تعبير الدوال مقابل التصريح عن الدوال

يسلط هذا القسم الضوء على الاختلافات الرئيسية بين تعبير الدوال (Function Expressions) والتصريح عن الدوال (Function Declarations).

- الصياغة: كيف نفرِّق بينهما في الشيفرة.
- التصريح عن دالة: يُصرَّح عن الدالة في سياق الشيفرة الرئيسية بشكل منفصل عن بقية التعليمات.

```
// التصريح عن دالة
function sum(a, b) {
  return a + b;
}
```

- تعبير دالة: تُنشَأ الدالة داخل تعبير برمجي (expression) أو داخل صياغة بانية أخرى. هنا، أُنشئَت الدالة في الجانب الأيمن من "تعبير الإسناد" (الإشارة =):

```
// تعبير دالة
let sum = function(a, b) {
  return a + b;
};
```

يظهر الفرق جليًّا عندما يُنشِئ محرك JavaScript كلتا الدالتين. يُنشَأ تعبير الدالة عندما يصل تنفيذ الشيفرة إليه ويصبح جاهزًا للاستخدام من تلك اللحظة فقط.

بمجرد أن ينتقل تنفيذ الشيفرة إلى الجانب الأيمن من معامل الإسناد. `let sum = function…`، تُنشَأ الدالة وتستطيع استخدامها بدءًا من تلك اللحظة (إسناد، استدعاء، …إلخ).

التصريح عند الدوال مختلف بعض الشيء. يمكن استدعاء الدالة المُصرَّح عنها قبل أن يصل تنفيذ الشيفرة إليها. فمثلًا تبقى الدالة المُصرَّح عنها في المجال العام مرئيَّةً لجميع أجزاء السكربت كله (أي يمكن أي يمكن استدعاؤها من أي مكان)، بغض النظر عن مكان وجودها في الشيفرة.

ذلك الأمر عائدٌ لخوارزميات داخلية تنفِّذها JavaScript قبل تنفيذ الشيفرة. فعندما تستعد JavaScript لتنفيذ سكربت ما، فإنَّها تبحث أولًا عن الدوال المصرَّح عنها في النطاق العام فيه وتنشئها. الأمر مشابه "لمرحلة التهيئة" (initialization stage). وبعد أن تعالج جميع تلك الدوال، تبدأ عملية تنفيذ الشيفرة، لذلك يمكن الوصول إليها من أي جزء من السكربت وتنفيذها.

إليك المثال التالي:

```
sayHi("جعفر"); // مرحبًا، جعفر
function sayHi(name) {
  alert( `مرحبًا، ${name}` );
}
```

تُنشَأ الدالة `sayHi` المُصرَّح عنها في المجال العام عندما تستعد JavaScript لتنفيذ السكربت، وتصبح مرئية في كامل أرجائه. لاحظ هنا أن السكربت لن يعمل كما سبق في حال استخدمت تعبير دالة بدلًا من عملية التصريح:

```
sayHi("جعفر"); // خطأ!
let sayHi = function(name) {  // (*) ;-) بَطَل مفعول السحر هنا
  alert( `Hello, ${name}` );
};
```

تُنشَأ الدالة هنا عند وصول التنفيذ إليها، يحدث ذلك في السطر (*)، لتعريفها بتعبيرٍ ويكون ذلك أي بعد فوات الأوان.

ميزة أخرى خاصة بعملية التصريح عن الدوال هي نطاق الكتلة الخاصة بها (block scope). فإن طُبِّق الوضع الصارم (strict mode) في السكربت، تُرَى الدالة المُعرَّفة داخل كتلة برمجية ضمنها فقط ولا يمكن الوصول إليها من خارجها.

على سبيل المثال، لنفترض أننا نريد تعريف الدالة `welcome()` بناءً على قيمة المتغير `age` التي نحصل عليها أثناء تنفيذ السكربت من المستخدم. نلاحظ أن التنفيذ لن يسير على ما هو مخطط له إن صرحنا عن الدالة بالشكل التالي:

```
let age = prompt("كم عمرك؟", 18);
// welcome التصريح الشرطي عن الدالة
if (age < 18) {
  function welcome() {
    alert("مرحبًا!");
  }
} else {
  function welcome() {
    alert("السلام عليكم!");
  }
}
// ثم استعمالها لاحقًا
welcome(); // خطأ: الدالة welcome غير معرّفة
```

حصلنا على ذلك الخطأ لأن الدالة المُصرَّح عنها داخل كتلة تبقى مرئية فقط ضمن نطاقها.دعنا نعدِّل على المثال السابق في محاولة لحل المشكلة:

```
let age = 16; // لنعتمد العمر 16 مثلًا
if (age < 18) {
  welcome();               // \   (تنفيذ)
                           //  |
  function welcome() {     //  |
    alert("مرحبًا!");       //  |  تعريف الدالة موجود داخل النطاق حيث استدعيت
  }                        //  |
                           //  |
  welcome();               // /   (تنفيذ)
```

```
} else {
  function welcome() {     // لمّا كان العمر 16، لن تُنشأ هذه الدالة
    alert("السلام عليكم!");
  }
}
// لم يعد هنالك نطاق (مجال) لأي كتلة هنا لعدم وجود أقواس معقوصة
// لذا، لن نستطيع رؤية الدالتين المنشأتين داخل الكتلتين السابقتين
welcome(); // خطأ: الدالة welcome غير معرّفة
```

ما الذي يمكن فعله لجعل الدالة welcome مرئية خارج نطاق الكتلة if الشرطية؟ تتمثَّل الطريقة الصحيحة في استخدام تعبير الدوال وذلك بإسناد تنفيذ الدالة welcome لمتغير يُعرَّف في النطاق العام لتصبح مرئية خارج الشرط if. لاحظ أنَّ الشيفرة تعمل على النحو المطلوب بهذه الطريقة:

```
let age = prompt("كم عمرك؟", 18);
let welcome;
if (age < 18) {
  welcome = function() {
    alert("مرحبًا!");
  };
} else {
  welcome = function() {
    alert("السلام عليكم!");
  };
}

welcome(); // تمام، لا يوجد أي خطأ
```

بإمكانك ببساطة استخدام معامل علامة الاستفهام ? بالشكل التالي:

```
let age = prompt("كم عمرك؟", 18);

let welcome = (age < 18) ?
  function() { alert("مرحبًا!"); } :
  function() { alert("السلام عليكم!"); };

welcome(); // تمام، لا يوجد أي خطأ
```

متى عليك التصريح عن الدالة ومتى تستعمل تعبير الدالة؟

تمنح عملية التصريح عن الدوال حريةً أكبر في تنظيم الشيفرة، لتوفير إمكانية استدعاء هذه الدوال قبل تعريفها وهذه أهم نقطة تُؤخذ في الحسبان عند المفاضلة بين التصريح والتعبير. أضف إلى ذلك أنَّ صياغة التصريح تُسهِّل قراءة الشيفرة أيضًا، إذ من الأسهل البحث عن `function f(…) {…}` في الشيفرة بدلًا من `let f = function(…) {…}`. فالتصريح ملفت للنظر أكثر من التعبير.

وإذا كان التصريح غير مناسب لسبب ما، أو احتجت إلى تعريف مشروط لدالة (مثل المثال السابق)، فيجب عليك استخدام التعبير عوضًا عن التصريح.

2.16.4 الخلاصة

- الدالة عبارة عن قيمة يمكن إسنادها أو نسخها أو تعريفها في أي مكان في الشيفرة.
- إن صُرِّح عن دالة في تعليمة برمجية (statement) منفصلة في سياق الشيفرة الرئيسية (النطاق العام)، فذلك يدعى "التصريح عن دالة".
- إن أُنشئت دالة في تعبير برمجي (expression)، فذلك يدعى "تعبير دالة".
- تعالج الدوال المُصرَّح عنها قبل تنفيذ الكتلة البرمجية (السكربت) الحاوية لها، وتصبح - نتيجةً لذاك - مرئيةً في أي مكان داخل الكتلة.
- تُنشَأ الدالة المُعرَّفة بوساطة تعبير عندما يحين دورها في التنفيذ بحسب مكان وجودها في السكربت.

التصريح هو الخيار المفضل والشائع في إنشاء الدوال، لتوفير إمكانية رؤية الدالة قبل أن يحين دورها في التنفيذ حيثما كان موضعها في السكربت. أضف إلى ذلك أنه يساعد على تنظيم الشيفرة ويُسهِّل من قراءتها.

من جهة أخرى، يُفضَّل تجنب استخدام تعبير الدوال إلا في الحالات التي لا يكون الصريح فيها ملائمًا. لقد تعرفت على مثالين في هذا الفصل يشرحان هذه النقطة، وسترى المزيد من الأمثلة في الدروس القادمة.

2.17 أساسيات الدوال السهمية

هناك صياغة بسيطة وموجزة لإنشاء الدوال تسمى "الدوال السهمية" (Arrow functions)، وغالبًا ما تكون أفضل من تعبير الدوال. سُمي هذا النوع من الدوال بالدوال السهمية لأنها تشبه السهم ببساطة، وإليك صياغتها:

```
let func = (arg1, arg2, ...argN) => expression
```

يُنشِئ هذا دالة `func` تملك الوسائط `arg1..argN`، وتُقيِّم التعبير `expression` على الجانب الأيمن ثم تُرجِع قيمته.

لاحظ أنَّ السطر السابق يقابل الشيفرة التالية بالضبط:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

ولكنه أكثر إيجازًا. إليك مثال آخر:

```
let sum = (a, b) => a + b;
/* صياغة الدالة السهمية أصغر من الصياغة العادية التالية
let sum = function(a, b) {
  return a + b;
};
*/
alert( sum(1, 2) ); // 3
```

كما ترى، تعني `(a, b) => a + b` أن الدالة تقبل وسيطين مسميين، `a` و `b`. تقيِّم الدالة التعبير `a + b` أثناء تنفيذها وتعيد الناتج.

- إن كان لدينا وسيطًا واحدًا فقط، يمكن إزالة قوسي المعامل:

```
let double = n => n * 2;
// let double = function(n) { return n * 2 } يماثل

alert( double(3) ); // 6
```

- إذا لم يكن هناك أي وسيط، توضع أقواس فارغة (ضرورية) بالشكل التالي:

```
let sayHi = () => alert("مرحبًا!");
```

```
sayHi();
```

يمكن استخدام الدوال السهمية كما يُستخدَم تعبير الدوال. فإليك مثال الدالة `welcome()` السابق باستعمال الدوال السهمية:

```
let age = prompt("كم عمرك؟", 18);
let welcome = (age < 18) ?
  () => alert('مرحبًا!') :
  () => alert("السلام عليكم!");

welcome(); // تمام، الدالة تعمل
```

تبدو الدوال السهمية للوهلة الأولى غير مألوفة وصعبة القراءة، لكن هذا سرعان ما يتغير عند اعتياد العينين على صيغة وهيكل تلك الدوال.

الدوال السهمية مناسبة جدًا لكتابة إجراءات بسيطة بسطر واحد فقط وستحبها كثيرًا إن كنت كسولًا (ميزة الكسل في المبرمج إيجابية جدًا :-D) وتستثقل كتابة كلمات كثيرة.

2.17.1 الدوال السهمية متعددة الأسطر

الأمثلة أعلاه أخذت الوسائط الموجودة على يسار المعامل `=>`، ومررتها إلى التعبير `expression` الموجود على جانبها الأيمن لتقييمه.

نحتاج في بعض الأحيان إلى شيء أكثر تعقيدًا، مثل التعابير والجمل البرمجية المتعددة، إذ في هذه الحالة استخدام الأقواس المعقوصة ثم استخدام الموجه `return` داخلها. إليك المثال التالي:

```
let sum = (a, b) => {  // يفتتح القوس المعقوص دالة متعددة الأسطر
  let result = a + b;
  return result; // إن استعملت الأقواس المعقوصة، فاستعمل return لإعادة نتائج
};
alert( sum(1, 2) ); // 3
```

ما زال هنالك المزيد!

تناول هذا الفصل الدوال السهمية بإيجاز، لكن هذا ليس كل شيء! الدوال السهمية لها ميزات أخرى مثيرة للاهتمام، لذا سنعود إليها لاحقًا في الفصل "زيارة الدوال السهمية مجدَّدًا". يمكننا في الوقت الحالي استخدامها مع إجراءات السطر الواحد وردود النداء.

2.17.2 الخلاصة

وجدنا أن الدوال السهمية مفيدةٌ جدًا خصوصًا في كتابة إجراءات بسطر واحد، وتأتي بشكلين:

1. بدون أقواس معقوصة: `(...args) => expression`، الجانب الأيمن عبارة عن تعبير، إذ تقيِّمه وترجع الناتج.
2. مع أقواس معقوصة: `(...args) => { body }`، تساعدك الأقواس في كتابة تعليمات برمجية متعددة داخل الدالة الواحدة، لكننا بحاجة إلى الموجه `return` لإرجاع شيء ما.

2.17.3 تمارين

ا. اكتب الشيفرة التالية مجدَّدًا باستعمال الدوال السهمية

ضع دوال سهميةً مقابلة لتعابير الدوال الموجودة في الشيفرة التالية:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
ask(
  "هل تقبل؟",
  function() { alert("لقد قبلت."); },
  function() { alert("لقد ألغيت التنفيذ."); }
);
```

الحل:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
ask(
  "هل قبلت؟",
  () => alert("لقد قبلت."),
  () => alert("لقد ألغيت التنفيذ.")
);
```

تبدو الشيفرة واضحة وقصيرة، أليس كذلك؟

2.18 مراجعة لما سبق

يلخِّص هذا الفصل مُميِّزات JavaScript التي تعرَّفنا عليها باختصار في الدروس السابقة مع إيلاء اهتمام خاص لأدق المواضيع.

2.18.1 صياغة الشيفرة

تنتهي التعليمات البرمجية في JavaScript بفاصلة منقوطة ; :

```
alert('مرحبًا'); alert('بالعالم');
```

يمكنك إنهاء تعليمة برمجية ببدء سطر جديد، إذ يعد محرف السطر الجديد على أنَّه فاصل للتعليمات:

```
alert('مرحبًا')
alert('بالعالم')
```

هذا ما يسمى "الإدراج التلقائي للفاصلة المنقوطة" (automatic semicolon insertion)، لكن يعمل كما هو متوقع في بعض الأحيان مثل الحالة التالية:

```
alert("سيحدث خطأ هنا بعد الرسالة")
[1, 2].forEach(alert)
```

تخبرك أغلب أدلة الصياغة بضرورة وضع فاصلة منقوطة بعد كل تعليمة برمجية (statement). انتبه إلى أنَّ الفاصلة المنقوطة غير مطلوبة بعد الكتل البرمجية `{...}` مثل كل الدوال وحلقات التكرار وغيرها:

```
function f() {
  // لا حاجة لفاصلة منقوطة بعد التصريح عن دالة
}
for(;;) {
  // لا حاجة لفاصلة منقوطة بعد حلقة تكرار
}
```

ولكن إذا وضعت فاصلة منقوطة "زائدة" في مكان ما داخل الشيفرة، لن تعدها JavaScript خطأ وستتجاهلها ببساطة.

تجد المزيد في فصل "بنية الشيفرة".

2.18.2 الوضع الصارم

إن أردت تمكين جميع ميزات JavaScript الحديثة كاملًا، يجب أن تستفتح السكربت بالموجه `"use strict"` مثل:

```
'use strict';
...
```

يجب أن يُكتَب هذا الموجِّه في بداية السكربت أو في بداية جسم الدالة.

تعمل الشيفرة بدون الموجه `"use strict"`، لكن هناك بعض الميزات تبقى على صورتها القديمة "المتوافقة" (compatible way). عمومًا، يُفضَّل العمل بالوضع الحديث دومًا. تُفعِّل بعض ميزات JavaScript الحديثة، مثل الأصناف التي ستتعرف عليها لاحقًا، الوضع الصارم ضمنيًّا.

تجد المزيد في قسم "الوضع الصارم: النمط الحديث لكتابة الشيفرات".

2.18.3 المتغيرات

يمكنك تعريف المتغيرات عبر:

- `let`
- `const` (متغير ثابت لا يمكن تغيير قيمته)
- `var` (نمط قديم سوف تتعرف عليه لاحقًا)

يمكن أن يحتوي اسم المتغير على:

- حروف وأعداد، لكن الحرف الأول لا يجب أن يكون عددًا.
- الرمزان `_`، `$`.
- يُسمَح أيضًا باستخدام الحروف الهجائية غير اللاتينية والحروف الهيروغليفية، ولكنها غير شائعة الاستخدام.

يمكنك تخزين أي قيمة داخل المتغير مثل:

```
let x = 5;
x = "عبد الله";
```

إليك سبعة أنواع من البيانات:

- `number`: يمثِّل الأعداد العشرية والأعداد الصحيحة.

- `string`: يمثِّل السلاسل النصية.
- `boolean`: يمثِّل القيم المنطقية: `true/false`.
- `null`: هذا النوع يعني أن القيمة "فارغة" أو "غير موجودة".
- `undefined`: هذا النوع يعني أن القيمة "غير مُعرَّفة".
- `object` و `symbol`: يمثِّل الأول بنية معقدة من البيانات والثاني مُعرِّفًا فريدًا، لم نسلط الضوء عليهما بعد.

يرجع المعامل `typeof` نوع القيمة، مع استثناءين:

```
typeof null == "object" // خطأ من أصل اللغة
typeof function(){} == "function" // تعامل الدوال معاملة خاصة
```

تجد المزيد في قسم "المتغيرات" وقسم "أنواع البيانات".

2.18.4 الدوال التفاعلية

لمَّا كنا نستعمل المتصفح بيئة عمل لنا، فستكون دوال واجهة المستخدم الأساسية هي:

- `prompt(question, [default])`: هي دالة تطرح السؤال `question`، ثم ترجع ما أدخله المستخدم أو تُرجع القيمة `null` في حال ألغى المستخدم عملية الإدخال (بالضغط على الزر "cancel").
- confirm(question): هي دالة تطرح السؤال `question` ثم تُتيح لك الاختيار بين "موافق" (Ok) أو "إلغاء" (Cancel) ثم تعاد قيمة منطقية، `true/false`، تمثِّل هذا الاختيار.
- alert(message): دالةٌ تُظهِر لك الرسالة `message` المُمرَّرة إليها فقط.

تظهر جميع هذه الدوال نافذة صغيرة تدعى "النافذة المنبثقة الشرطية" (modal window، وهي عنصر تحكم رسومي)، فهي توقف تنفيذ الشيفرة وتمنع المستخدم من التفاعل مع صفحة الويب حتى يتفاعل معها.

جرب تنفيذ المثال التالي:

```
let userName = prompt("ما اسمك؟", "محمد");
let isTeaWanted = confirm("هل تريد كوبًا من الشاي؟");
alert( "الزائر: " + userName ); // محمد
alert( "أيريد كوبًا من الشاي؟" + isTeaWanted ); // true
```

تجد المزيد في فصل "الدوال التفاعلية".

2.18.5 المعاملات

تدعم JavaScript المعاملات التالية:

ا. المعاملات الحسابية

معاملات الحساب الأساسية وهي `* + - /` بالإضافة إلى المعامل `%` لإيجاد باقي القسمة، وأيضًا معامل القوة `**`.

يدمج معامل الجمع الثنائي `+` السلاسل النصية. إذا كان أحد العاملين عبارة عن سلسلة نصية، فسيُحوَّل الآخر إلى سلسلة نصية أيضًا مثل:

```
alert( '1' + 2 ); // '12', سلسلة نصية
alert( 1 + '2' ); // '12', سلسلة نصية
```

ب. معاملات الإسناد

هنالك معامل إسناد بسيط وهو `a = b` ومعامل إسناد مركَّب مثل `a *= 2`.

ج. المعاملات الثنائية

تعمل المعاملات الثنائية في المستوى المنخفض للبيانات أي في مستوى البتات، لذا ارجع إلى توثيقها في موسوعة حسوب إن لزمتك.

د. المعاملات الشرطية

المعامل الوحيد الذي يحوي ثلاث عوامل هو معامل علامة استفهام `?` أو كما يسمى "المعامل الثلاثي":

```
cond ? resultA : resultB
```

إذا تحقق الشرط `cond`، يُرجع المعامل القيمة `resultA`؛ خلا ذلك، يُرجع القيمة `resultB`.

ه. المعاملات المنطقية

ينفِّذ المعاملان المنطقيان AND `&&` و OR `||` "دارة تقييم قصيرة" (short-circuit evaluation) ثمَّ يرجعان القيمة المُقيَّمة الناتجة. يُحوِّل المعامل المنطقي NOT `!` العامل الذي استدعي معه لقيمة منطقية ثم يُرجع القيمة المعاكسة له.

و. عامل الاستبدال اللاغي

يوفر العامل ?? وسيلة لاختيار قيمة معرَّفة من مجموعة متغيرات، فمثلًا نتيجة `a ?? b` هي a إلا إن كانت قيمة هذا المتغير null أو undefined، ففي تلك الحالة، تؤخذ قيمة b.

ز. معاملات الموازنة

يُحوّل معامل المساواة == القيم المختلفة إلى أعداد ثم يتحقق من المساواة (ما عدا القيمة الفارغة null والقيمة غير المُعرَّفة undefined التي تساوي بعضها بعضًا). إليك المثال التالي:

```
alert( 0 == false ); // true
alert( 0 == '' ); // true
```

وكذلك تفعل معاملات الموازنة الأخرى.

فيما يخص معامل المساواة الصارمة === لا يُوحَّد نوع القيم المراد التحقق من تساويها، إذ تعد عملية التحقق من نوعين مختلفين بالنسبة لهذا المعامل عملية غير محققة دومًا (أي النوعان غير متساويين).

تعدُّ القيمة الفارغة null والقيمة غير المحددة undefined حالة خاصة، إذ تساوي إحداهما الأخرى (عبر المعامل ==) ولا تساويان أي شيء آخر.

توازن إشارة الأكبر > وإشارة الأصغر < بين كل محرفين متقابلين من السلسلتين النصيَّتين المراد موازنتها مع بعضهما، وأما بالنسبة لموازنة أنواع البيانات الأخرى، فتُحوَّل أولًا إلى أعداد ثم توازن.

ح. معاملات أخرى

هنالك معاملات أخرى غير شائعة الاستخدام مثل معامل الفاصلة.

تجد المزيد في الفصول التالية: فصل "المعاملات في JavaScript" وفصل "معاملات الموازنة" وفصل "المعاملات المنطقية".

2.18.6 حلقات التكرار

لقد غطينا ثلاثة أنواع من حلقات التكرار هي:

```
// 1
while (condition) {
  ...
}
// 2
do {
  ...
} while (condition);
// 3
for(let i = 0; i < 10; i++) {
```

```
    ...
  }
```

المتغير الذي يُعرَّف داخل حلقة التكرار `for(let...)` مرئيٌّ داخلها فقط، ولكن يمكنك حذف `let` وإعادة استخدام المتغير.

يأمر الموجهان `break/continue` بإنهاء التكرار الحالي والعودة للحلقة (`continue`) أو الخروج من الحلقة بأكملها (`break`). يمكن استخدام اللافتات (labels) للتحكم بالحلقات المتداخلة وإنهائها مثلًا.

تجد المزيد في فصل "حلقتي التكرار `while` و `for`". وسوف نتعرف لاحقًا على المزيد من حلقات التكرار.

2.18.7 التعبير switch

يمكن للتعبير `switch` أن يحل محل التعبير الشرطي `if`، إذ يمنحك طريقة وصفية أكثر لموازنة قيمة ما مع عدَّة قيم. ويستخدم معامل المساواة الصارمة `===` في عملية الموازنة. إليك المثال التالي:

```
let age = prompt('كم عمرك؟', 18);
switch (age) {
  case 18:
    alert("لن يعمل"); // الناتج سلسلة نصية وليس عدد
  case "18":
    alert("يعمل");
    break;
  default:
    alert("أي قيمة غير مساوية للقيمتين السابقتين");
}
```

تجد المزيد في فصل "التعبير switch".

2.18.8 الدوال

إليك ثلاث طرائق لإنشاء دالة في JavaScript:

الطريقة الأولى: التصريح عن دالة: تُعَرَّف الدالة في سياق الشيفرة الرئيسية (ضمن النطاق العام) بشكل منفصل عن بقية التعليمات:

```
function sum(a, b) {
  let result = a + b;
  return result;
```

```
}
```

الطريقة الثانية: تعبير دالة: تُنشَأ الدالة داخل تعبير برمجي أو داخل كتلة برمجية أخرى:

```
let sum = function(a, b) {
  let result = a + b;
  return result;
};
```

الطريقة الثالثة: الدوال السهمية:

```
// التعبير في الطرف الأيمن
let sum = (a, b) => a + b;
// يمكن أن تمتد على عدة أسطر باستعمال الأقواس المعقوصة شرط إعادة شيء
let sum = (a, b) => {
  // ...
  return a + b;
}
// دون معاملات
let sayHi = () => alert("مرحبًا");
// مع معامل وحيد
let double = n => n * 2;
```

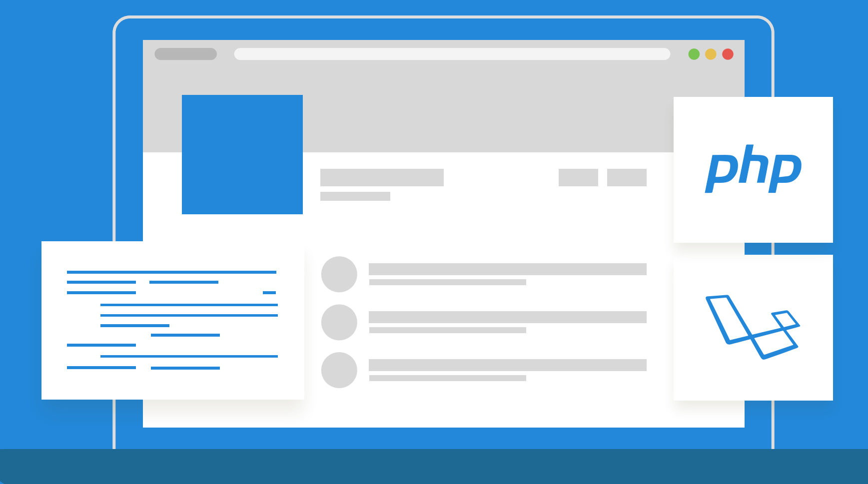
- قد تحتوي الدوال على متغيرات محلية، إذ تُعرَّف تلك المتغيرات داخل جسم الدالة وتبقى مرئيَّةً داخل الدالة فقط.
- يمكن أن تملك المعاملات قيمًا افتراضية مثل: `function sum(a = 1, b = 2) {...}`.
- تُرجع الدوال قيمة ما دائمًا. وإذا لم يكن هناك الموجه `return`، فستكون تلك القيمة المعادة القيمة `undefined`.

تجد المزيد في فصل "الدوال".

2.18.9 المزيد قادم

كان ما سبق قائمةً مختصرةً بميزات JavaScript. لقد تعرَّفت إلى الآن على الأساسيات في JavaScript. أمَّا في الفصول القادمة، فستتعرف على المزيد من المميزات والمواضيع المتقدمة، لذا خذ استراحة وتهيَّأ جيدًا لما هو قادم.الاعتناء بجودة الشيفرة

دورة تطوير تطبيقات الويب باستخدام لغة PHP



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

3. الاعتناء بجودة الشيفرة

يتضمن هذا الفصل الأقسام التالية:

1. تنقيح الأخطاء في chrome
2. نمط كتابة الشيفرة
3. التعليقات
4. شيفرة النينجا البرمجية
5. الاختبار الآلي باستخدام mocha
6. تعويض نقص دعم المتصفحات

3.1 تنقيح الأخطاء في Chrome

قبل كتابة شيفرات برمجية أكثر تعقيدا، لنتطرَّق إلى موضوع مهم ألا وهو تنقيح الأخطاء.

تنقيح الأخطاء هي عملية إيجاد الأخطاء في السكربت وإصلاحها وتدعم جميع المتصفحات الحديثة وبعض البيئات الأخرى عملية "تنقيح الأخطاء" في أدوات المطور (developer tools وتختصر إلى DevTools) وهي واجهة مستخدم خاصة والتي تجعل العثور على الأخطاء وتصحيحها أمرًا سهلًا. تُتيح هذه الواجهة أيضا تَتَبُّع الشيفرة خطوة بخطوة لمعرفة ما يحدث فيها بالتفصيل.

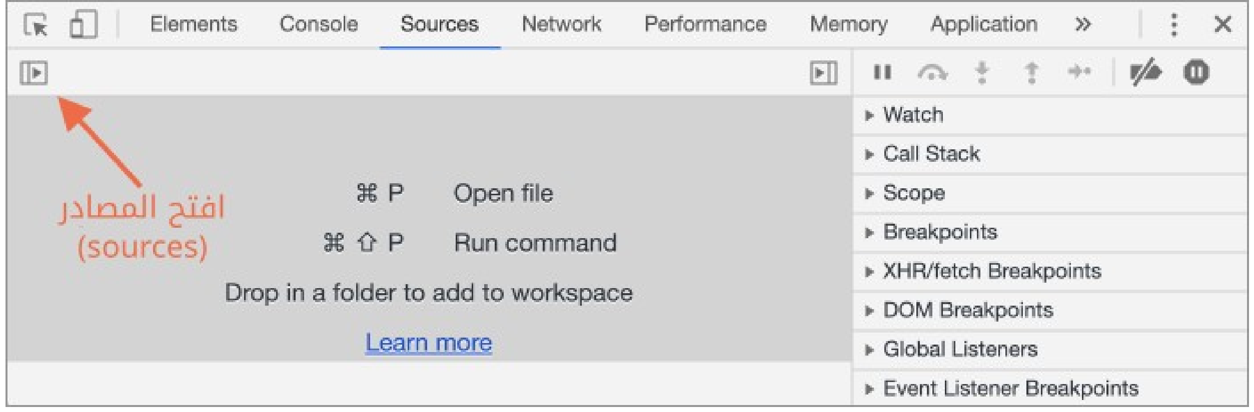
سنستخدم المتصفح Chrome لأن لديه ميزات كافية لذلك، كما تتوفر هذه الميزات في معظم المتصفحات الأخرى.

3.1.1 جزء الموارد Sources

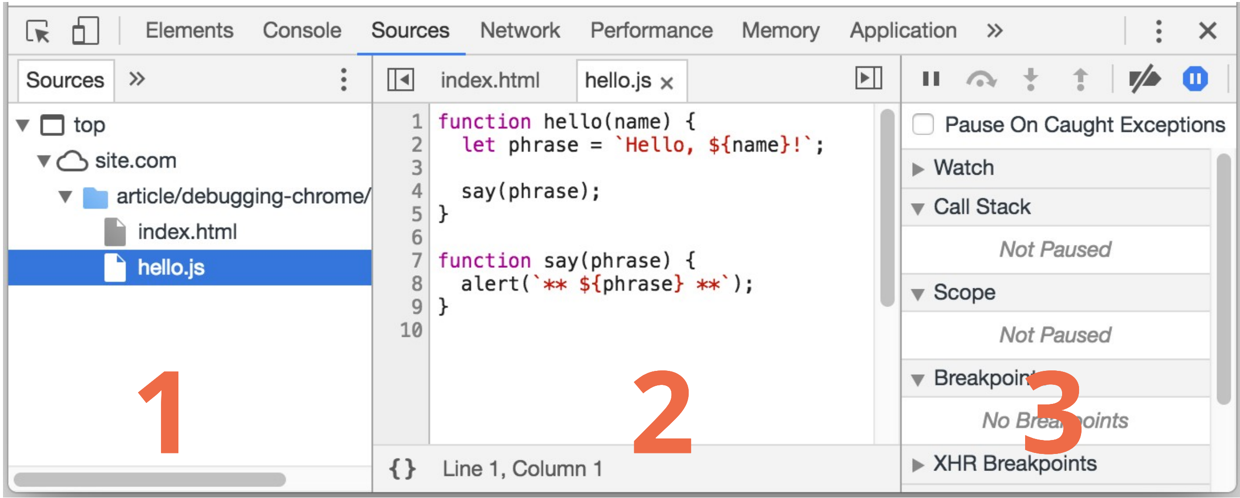
قد يبدو إصدار Chrome لديك مختلفًا بعض الشيء، إلا أن المحتوى هو ذاته.

- افتح صفحة example في Chrome.
- شغِّل أدوات المطور بالضغط على `F12` (على أجهزة Mac، استعمل الاختصار `Cmd+Opt+I`).
- اختر الجزء `sources`.

إن كانت هذه هي مرتك الأولى للدخول إلى جزء `sources`، فهذا ما ستراه:



يفتح هذا الزر علامة تبويب تحوي الملفات. اضغط عليها واختر `hello.js` من العرض الشجري "tree view". يجب أن ترى التالي:



هنا يمكننا رؤية ثلاث مناطق:

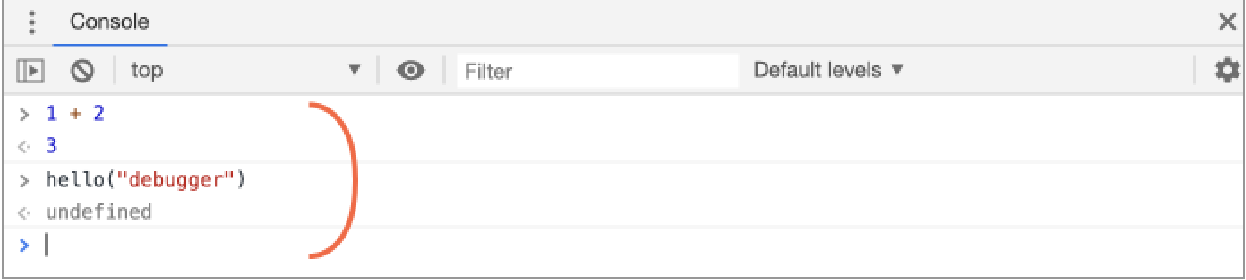
1. منطقة الموارد "Resources zone" والتي تعرض ملفات HTML، و JavaScript، و CSS، وغيرها من الملفات بما في ذلك الصور المُلحقة بالصفحة. قد تظهر إضافات Chrome هنا أيضًا.
2. منطقة المصدر "Source zone" والتي تعرض الشيفرة البرمجية المصدرية.
3. منطقة المعلومات والتحكم "Information and control zone" لتنقيح الأخطاء، سَنكتشفها أكثر فيما يلي.

يمكنك الضغط على الزر مُجددًا لإخفاء قائمة الموارد وإعطاء الشيفرة البرمجية مساحة كافية.

3.1.2 طرفية التحكم (Console)

تظهر الطرفية عند الضغط على `Esc` ويمكن كتابة الأوامر فيها ثم الضغط على `Enter` لتنفيذها ثم تظهر مخرجات الأمر أسفله مباشرة.

لننفذ مثلًا الأمر `2+1` والذي ينتج عنه القيمة 3، بينما `hello("debugger")` لا يُظهِر أي نتائج، لذلك فإننا نرى `undefined`:



3.1.3 نقاط التوقف (Breakpoints)

لنختبر ما يحدُث أثناء تنفيذ الشيفرة البرمجية في صفحة example. في الصفحة hello.js، اضغط على السطر رقم 4؛ نضغط على الرقم ذاته وليس السطر.

هكذا تكون قد أنشأت نقطة توقف. اضغط على الرقم 8 أيضًا. يجب أن يبدو الشكل كما في الصورة التالية:

نقطة التوقف هي نقطة يتوقف فيها مصحح الأخطاء عن تنفيذ JavaScript تلقائيًا.

يمكننا فحص المتغيرات الحالية وتنفيذ الأوامر أو أي شيء آخر في الطرفية أثناء توقف عمل الشيفرة البرمجية. أي أنه يمكننا تتبع الشيفرة البرمجية عند نقطة معينة عبر إيقافها والتأكد من أي شيء فيها كما نريد.

يمكننا رؤية قائمة بالعديد من نقاط التوقف في الجزء الأيمن من الشاشة. يكون الأمر مفيدًا عند وجود عدة نقاط توقف في أكثر من ملف، وذلك يتيح لنا:

- التنقل بسرعة إلى نقاط التوقف في الشيفرة البرمجية (بالضغط عليها من الجزء الأيمن).
- إلغاء تفعيل نقاط التوقف مؤقتا بإلغاء تحديدها.
- حذف نقطة التوقف بالضغط عليها باليمين واختيار حذف "remove".
- وهلم جرًا …

نقاط التوقف الشرطية

يتيح لك الضغط يمينًا على رقم السطر إنشاء نقطة توقف شرطية تُنَفَّذ عند تحقق الشرط المحدد فقط. يكون ذلك مفيدًا عندما تريد إيقاف التنفيذ لمعرفة قيمة أي متغير أو قيمة أي معامل في دالة

3.1.4 تعليمة Debugger

يمكن أيضا إيقاف تنفيذ الشيفرة البرمجية بوضع الأمر `debugger` فيه كما يلي:

```
function hello(name) {
  let phrase = `Hello, ${name}!`;

  debugger;  // <-- يتوقف المنقح هنا

  say(phrase);
}
```

هذه الطريقة سهلة عندما نُعدِّل الشيفرة البرمجية باستخدام محرر الشيفرات البرمجية ولا نريد الانتقال إلى المتصفح وتشغيل السكربت في وضع أدوات المطور لإنشاء نقاط توقف.

3.1.5 توقف وتحقق

في المثال، يتم استدعاء الدالة `hello()` أثناء تحميل الصفحة، لذلك فإن أسهل طريقة لتفعيل مُنقِّح الأخطاء (بعد إعداد نقاط التوقف) هي إعادة تحميل الصفحة. اضغط على `F5` (لمستخدمي ويندوز أو لينكس)، أو اضغط على `Cmd+R` (لمستخدمي Mac)، فسيتوقف تنفيذ الشيفرة البرمجية في السطر الرابع حيث تم إنشاء نقطة التوقف:

افتح قوائم المعلومات المنسدلة على اليمين (موضحة بأسهم) إذ تتيح هذه القوائم التحقق من حالة السكربت الحالية:

1. `Watch` - تعرض القيم الحالية لأية تعابير: يمكنك الضغط على + وإدخال أي تعبير تريده. سيعرض المعالج قيمته في أي وقت ويحسب قيمته تلقائيا أثناء التنفيذ.

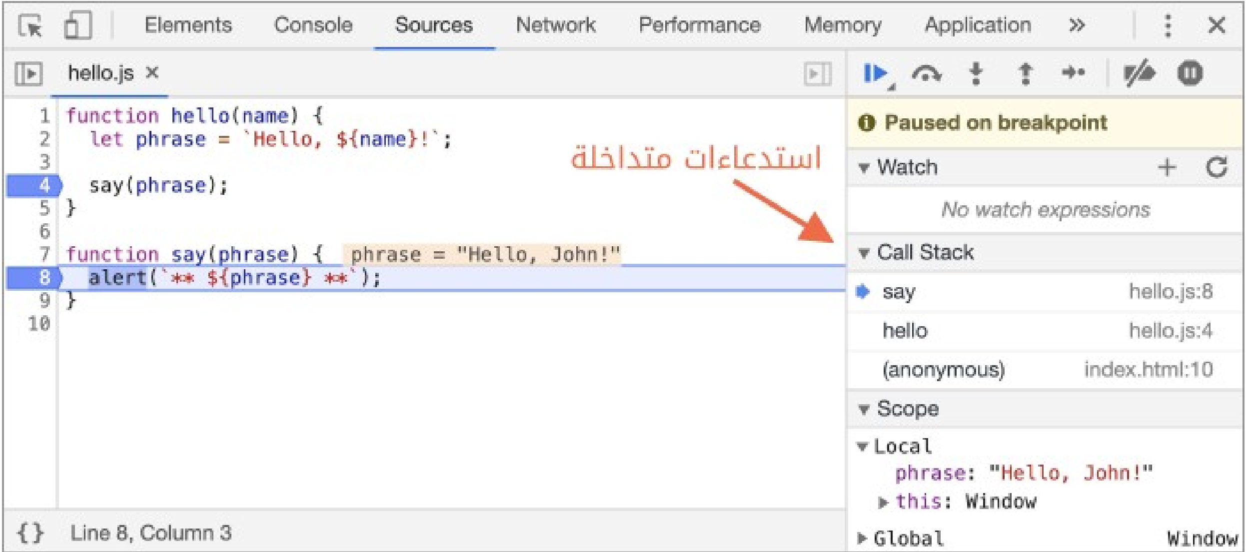
2. `Call Stack` - تعرض سلسلة الاستدعاءات المتداخلة: في الوقت الحالي، المعالج وصل حتى استدعاء الدالة `hello()`، المُستدعاة من خلال السكربت `index.html` (لا يوجد دوال أخرى لذلك سُمِّيَت "anonymous"). إن ضَغَطت على عنصر من المُكدِّس (stack) مثلا "anonymous"، فسينتقل المعالج مباشرة إلى الشيفرة البرمجية المُمَثِّلة لهذا العنصر وستتمكن من فحص جميع متغيراته أيضا.
3. `Scope` - تعرض المتغيرات الحالية: `Local` تعرض متغيرات الدالة المحلية إذ يمكنك أيضا رؤية قيم هذه المتغيرات موضحة على اليمين بينما تعرض `Global` المتغيرات العامة (خارج نطاق أي دالة). يوجد أيضا الكلمة المفتاحية `this` والتي لم تُشرح بعد، لكن سيتم شرحها قريبا.

3.1.6 تتبع التنفيذ

يوجد بعض الأزرار لِتَتبُّع التنفيذ أعلى يمين اللوحة، لِنتطرق إليها.

استأنف التنفيذ `(F8)`

يستأنف التنفيذ، وإن لم توجد أي نقاط توقف، فإن التنفيذ سيستمر بعد الضغط على هذا الزر وسَيفقد منقِّح الأخطاء السيطرة على السكربت. هذا ما سنراه بعد الضغط عليه:



تم استئناف التنفيذ، ووصل لنقطة توقف أخرى داخل الدالة `say()` وتوقف هناك. انظر في قسم `Call stack` على اليمين فقد تم تنفيذ استدعاء آخر إذ وصل التنفيذ الآن حتى الدالة `say()`.

أخطو خطوة أخرى ونفِّذ الأمر التالي (F9)

الضغط على عليه سيؤدي إلى تنفيذ التعليمة التالية، وسـيُنفَّذ في حالتنـا هـذه الأمـر alert. عمومـًا، يـؤدي الضغط على هذا الزر المرة تلو الأخرى إلى تنفيذ الشيفرة تعليمة بعد تعليمة.

- أخطو خطوة أخرى ونفِّذ الأمر التالي دون الدخول ضمن أي دالة (`F10`)

يشبه هذا الزر الزر السـابق من ناحيـة تنفيـذ التعليمـة التاليـة تحديـدًا إن كـانت التعليمـة التاليـة عبـارة عن استدعاء دالة. فالزر السابق يدخل ضمن الدالة (الدالة المعرَّفة ضمن الشيفرة وليس الدالة المبنيـة مسـبقًا مثـل alert) وينتقل إلى السطر الأول فيها، بينما يسلك هذا الأمر سلوكًا مختلفًـا ويتخطى الـدخول إلى الدالـة وينتقـل إلى التعليمـة الـتي بعـد تلـك الدالـة مباشـرةً. هـذا الأمـر مفيـد في حـال لم تكن مهتمًـا في معرفـة مـا يحصل داخل الدالة.

- أخطو خطوة أخرى للداخل (`F11`)

يشبه عمل هذا الزر زر "أخطو خطوة أخرى" ما قبل السابق في جميـع النـواحي باسـتثناء حالـة التعامـل مـع اسـتدعاء دالـة غـير متزامنـة (asynchronous function)، وإن كنت جديـدًا على عـالم JavaScript ولم تفهم المصطلح السابق، فتجاهل هذا الزر حاليًا ولا تشغل بالك بالفرق بينهما.

سأذكر الفرق في حال عدت إلى هذا القسم لاحقًا أو كنت تعرف ما هي الدوال غير المتزامنـة. أمَّا الـزر فيتجاهل المهام غير المتزامنة (async actions) مثل استدعاء الدالة setTimeout (الـتي تجـدول للاسـتدعاء لاحقًا)، بينما يدخل هذا الزر ضمن الدوال غير المتزامنة منتظرًا تنفيذًا أي حدث إن كان ذلك ضروريًا.

- استمر بالتنفيذ حتى نهاية الدالة الحالية (`Shift+F11`)

سيتوقف التنفيذ عند آخر سطر للدالة الحالية. يكون هذا الأمر مفيدا عند الدخول إلى استدعاء دالة مصـادفةَ باستخدام الزر ، لكن تتبع هذه الدالة ليس أمرا مهما، لذلك نقوم بتخطي تتبعها.

- تفعيل/تعطيل جميع نقاط التوقف

لا يقوم هذا الزر بمتابعة التنفيذ، إنما يُفَعِّل/يُعطِّل نقاط التوقف.

- تفعيل/تعطيل التوقف التلقائي في حال حدوث خطأ

عند تفعيله في وضع أدوات المطور، فـإن الشـيفرة البرمجيـة تتوقـف عن التنفيـذ تلقائيـا عنـد أي خطـأ في السكربت ثم يمكننا تحليل المتغيرات لمعرفة سبب الخطـأ؛ لـذلك، إن أوقـف خطـأ مَـا متابعـة تنفيـذ الشـيفرة

البرمجية، يمكننا فتح مصحح الأخطاء وتفعيل هذا الخيار وإعادة تحميل الصفحة لرؤية مكان توقف الشيفرة البرمجية وما هو المحتوى عند تلك النقطة.

> **الاستمرار حتى هنا "Continue to here"**
>
> الضغط يمينًا بالفأرة على أي سطر من الشيفرة البرمجية يفتح قائمة السياق المحتوية على خيار مفيد يُدعَى "Continue to here". يكون هذا الخيار مُفيدًا عندما نريد التقدم بضع خطوات للأمام بدون وضع نقطة توقف.

3.1.7 التسجيل Logging

يمكن استخدام الدالة `console.log` لِعرض شيء على الشاشة تمثِّل مخرجات من الشيفرة البرمجية. فيعرض الأمر التالي مثلًا القيم من 0 حتى 4 إلى الشاشة:

```
// open console to see
for (let i = 0; i < 5; i++) {
  console.log("значение", i);
}
```

لا يرى المستخدم العادي هذه المخرجات. لرؤيتها، افتح قائمة الطرفية `Console` مباشرةً في لوحة أدوات المطور أو اضغط على `Esc` إن كنت في قائمة أخرى من تلك اللوحة هكذا لتُفتَح الطرفية في أسفل تلك القائمة.

إن كانت الشيفرة البرمجية تحتوي على أوامر `console.log`، فسنرى ما يحدث من خلال سجلات التتبع بدون الدخول إلى المُصحح.

3.1.8 الخلاصة

كما رأينا، فإن هناك ثلاث طرائق رئيسية لإيقاف تنفيذ السكربت:

1. باستخدام نقطة توقف.
2. الأمر `debugger`.
3. وجود خطأ (في حال كانت أدوات المطور مفتوحة وكان الزر مُفَعَّلًا).

يمكننا عند توقف السكربت فحص المتغيرات وتتبع الشيفرة البرمجية لرؤية أي أخطاء. يوجد العديد من الخيارات الأخرى في أدوات المطور أكثر مما تم شرحه إلى الآن. إن أردت مزيدًا من التفاصيل، يمكنك الرجوع إلى توثيق أدوات المطور الرسمي في هذا الرابط: developers.google.com/web/tools/chrome-devtools.

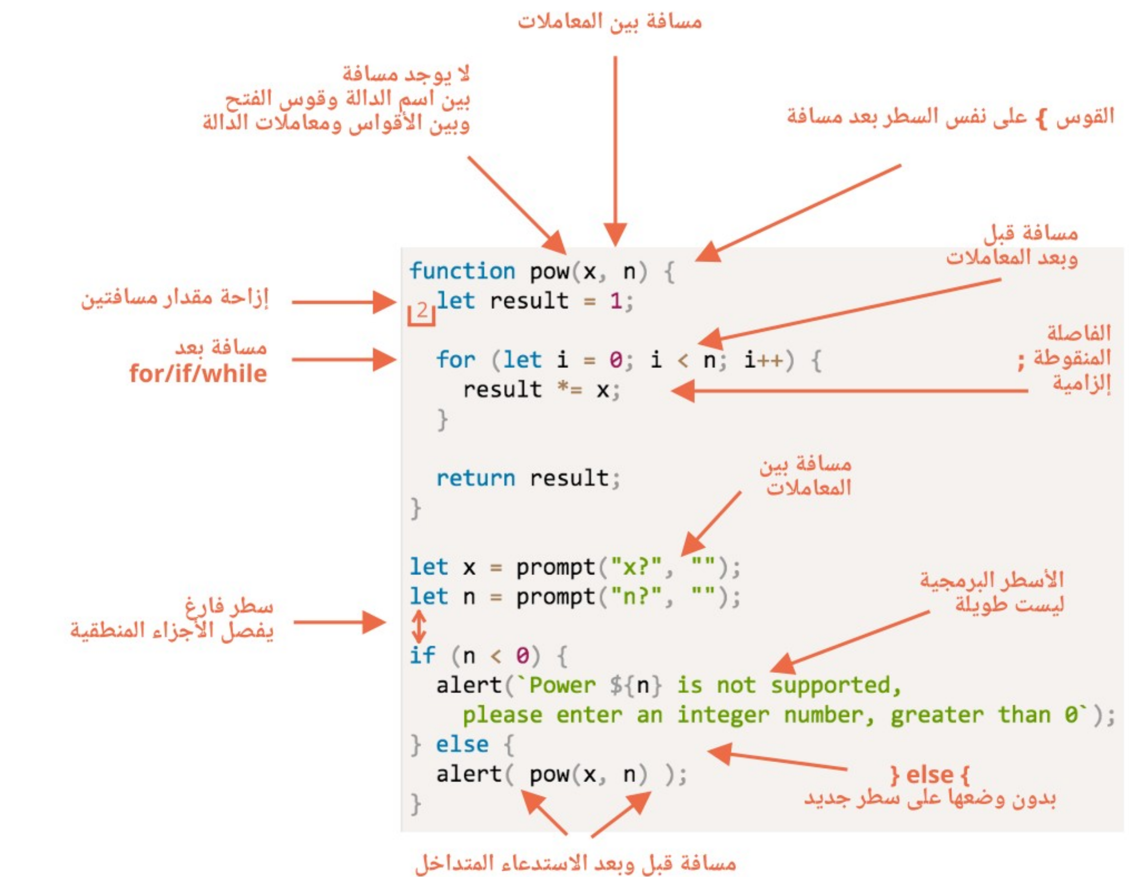
تُعَد المعلومات التي وُضِعَت كافيـة لبـدء تتبـع أي شـيفرة برمجيـة وتنقيحهـا، لكنـك سـتحتاج للاطلاع على الدليل لاحقًا لتعلم ميزات متقدمة في أدوات المطور، خاصة إن كنت تتعامل كثيرا مع المتصفح. يمكنك في أي وقت اللعب والتجريب في أدوات المطور ورؤيـة مـا يحـدث. تُعـد هـذه أفضـل طريقـة لتعلم العمـل مـع أدوات المطور. لا تنسَ الضغط يمينا وتجريب قوائم السياق.

3.2 نمط كتابة الشيفرة

يجب أن تكون الشيفرة البرمجية مرتبة ونظيفة وسهلة القراءة قدر الإمكان. هذا ما يسمى بفن البرمجة وهو أخذ مهمة معقَّدة وبرمجتها بطريقة صحيحة ومقروءة في الوقت ذاته. يساعد نمط كتابة الشيفرة كثيرًا في ذلك.

3.2.1 الصياغة

في الصورة أدناه يوجد صفحة تحوي بعض القواعد المهمة في تركيب الشيفرات البرمجية.



سنناقش هذه القواعد وأسبابها بالتفصيل.

> **لا يوجد قواعد إلزامية**
>
> لا يوجد قاعدة إلزامية لنمط كتابة الشيفرات وهذه القواعد تعد تفضيلات وأعراف وليست أساسيات ثابتة.

ا. الأقواس المعقوصة {}

تُكتب الأقواس المعقوصة Curly Braces في معظم مشاريع JavaScript بالطريقة "المصرية" وذلك بوضع قوس الفتح في نفس السطر الذي يحوي الكلمة المفتاحية - ليس في سطر جديد. يجب وضع مسافة قبل القوس الافتتاحي كما يلي:

```
if (condition) {
  // افعل هذا
  //.... وذاك
  //.... وذاك
}
```

التعليمة المكونة من سطر واحد مثل `if (condition) doSomething()` تعد حالة متطرِّفة مهمة، فهل يجب استخدام الأقواس فيها؟

فيما يلي بعض البدائل المشروحة. يمكنك قراءتها والحكم على درجة سهولة قراءتها بنفسك:

- يضع المبتدئون أحيانًا أقواسًا في الموضع الذي لا حاجة فيه إليها - ما يعد ممارسة خاطئة :-

```
if (n < 0) {alert(`Power ${n} is not supported`);}
```

- الانتقال إلى سطر جديد بدون استخدام أقواس. تجنب هذا الأمر لأنه يسبب بعض الأخطاء:

```
if (n < 0)
  alert(`Power ${n} is not supported`);
```

- سطر واحد بدون أقواس يُعد مقبولا في حال كان السطر قصيرًا:

```
if (n < 0) alert(`Power ${n} is not supported`);
```

- أفضل الطرائق:

```
if (n < 0) {
  alert(`Power ${n} is not supported`);
}
```

يمكن استخدام سطر واحد في حالة الشيفرات البرمجية المختصرة مثل: `if (cond) return null`. لكن استخدام شيفرة برمجية كتلية (كما في رقم 4) هو الأفضل من ناحية سهولة القراءة.

ب. طول السطر

لا يحب أحدٌ قراءة سطر برمجي طويل إذ أصبح فصل الأسطر الطويلة ممارسة عامة لدى الجميع.

إليك المثال التالي:

```
// تسمح الفاصلة العليا المائلة ` بتقسيم النص إلى عدة أسطر
let str = `
  Ecma International's TC39 is a group of JavaScript developers,
```

```
  implementers, academics, and more, collaborating with the community
  to maintain and evolve the definition of JavaScript.
`;
```

بالنسبة للتعليمة البرمجية `if`:

```
if (
  id === 123 &&
  moonPhase === 'Waning Gibbous' &&
  zodiacSign === 'Libra'
) {
  letTheSorceryBegin();
}
```

يتم الاتفاق على الحد الأقصى لطول السطر البرمجي على مستوى فريق العمل، ويكون طول السطر البرمجي غالبا بين 80 إلى 120 محرفًا.

ج. المسافة البادئة

يوجد نوعان من المسافات الفارغة البادئة (indents):

- البادئة الأفقية: 2 أو 4 مسافات.

تتكون المسافة البادئة الأفقية من 2 أو 4 فراغات أو تكون عبارة عن مسافة جدولة أفقية (الزر `Tab`). وُجدَت اختلافات قديمة حول أيهما أفضل، لكن المسافات هي الأكثر شيوعا هذه الأيام.

تتميز الفراغات عن الجدولة `Tab` بكونها أكثر مرونة أثناء التعديل. مثلًا، يمكننا إزاحة المتغيرات داخل قوس مفتوح كالتالي:

```
show(parameters,
     aligned, // خمسة فراغات
     one,
     after,
     another
  ) {
  // ...
}
```

- البادئة العمودية: الأسطر الفارغة لفصل الشيفرات البرمجية إلى أجزاء منطقية

يمكن تقسيم كل شيء - حتى الدالة الوحيدة - إلى أجزاء منطقية لتسهل قراءتها. في المثال أدناه، تم تقسيم تعريف المتغيرات وحلقة التكرار الرئيسية والنتيجة عموديًا:

```
function pow(x, n) {
  let result = 1;
  //              <--
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  //              <--
  return result;
}
```

يمكنك وضع سطر فار غ حيثما تريد لجعل الشيفرة البرمجية مقروءة بسهولة. لا يجب أن يوجد أكثر من 9 أسطر بدون بادئة عمودية.

د. الفواصل المنقوطة ";"

يجب أن توضع فاصلة منقوطة بعد كل تعليمة برمجية، حتى وإن كان من الممكن عدم إضافتها. يوجد بعض اللغات حيث يكون استخدام الفاصلة المنقوطة اختياريا وتُستخدَم نادرًا آنذاك ولكن في بعض الحالات في JavaScript لا يحل السطر الجديد محل الفاصلة المنقوطة مما يجعل الشيفرة البرمجية عرضةً للخطأ. يمكنك الاطلاع أكثر عن ذلك في جزء بنية الشيفرة البرمجية.

إن كنت مبرمجًا متمرسًا في JavaScript، يمكنك اختيار نمط كتابة بدون فاصلة منقوطة مثل StandardJS. أو يُفَضَّل استخدام فواصل منقوطة لتجنب الأخطاء، فأغلب المبرمجين يضعون فواصل منقوطة.

ه. مستويات التداخل

تجنب تداخل الشيفرة البرمجية للعديد من المستويات، ففي الحلقة المتكررة مثلًا يُفَضَّل استخدام التعليمة `continue` لتجنب التداخل العميق. على سبيل المثال، بدلًا من إضافة `if` شرطية داخلية كالتالي:

```
for (let i = 0; i < 10; i++) {
  if (cond) {
    ... // <- مستوى تشعيب إضافي
  }
}
```

يمكننا كتابة:

```
for (let i = 0; i < 10; i++) {
  if (!cond) continue;
  ...  // <- لا مزيد من التشعيبات
}
```

يمكن استخدام الأسلوب ذاته مع `if/else` و `return`. مثلًا، نحتاج لجزأين في المثال أدناه.

- خيار 1:

```
function pow(x, n) {
  if (n < 0) {
    alert("Negative 'n' not supported");
  } else {
    let result = 1;

    for (let i = 0; i < n; i++) {
      result *= x;
    }

    return result;
  }
}
```

- خيار 2:

```
function pow(x, n) {
  if (n < 0) {
    alert("Negative 'n' not supported");
    return;
  }
  let result = 1;
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
```

يُعد الخيار 2 أسهل قراءةً من الخيار 1 لأن الحالة الخاصة `n < 0` فُحصَت مسبقًا ويمكننا بعد فحص قيمتها الانتقال إلى جزء الشيفرة البرمجية الرئيسية دون الحاجة لتداخل أكثر.

3.2.2 موضع الدوال

في حال كنت تكتب العديد من الدوال المساعدة والتي تستخدمها الشيفرة البرمجية، فإن هناك ثلاث طرائق لتنظيم هذه الدوال.

- تعريف الدوال أعلى الشيفرة البرمجية التي تستخدمها:

```
// تعريفات الدوال
function createElement() {
  ...
}

function setHandler(elem) {
  ...
}

function walkAround() {
  ...
}

// الشيفرة التي تستخدمها
let elem = createElement();
setHandler(elem);
walkAround();
```

- الشيفرة البرمجية أولًا ثم الدوال:

```
// الشيفرة التي تستخدم الدوال
let elem = createElement();
setHandler(elem);
walkAround();

// --- دوال مساعدة ---
function createElement() {
  ...
```

```
}

function setHandler(elem) {
  ...
}

function walkAround() {
  ...
}
```

- الطريقة المختلطة: تُعرَّف الدالة في أول مكان تستخدم فيه.

الخيار الثاني هو الأفضل غالبًا ذلك لأنه عند قراءة شيفرة برمجية، فإننا نريد معرفة ما تقوم به أولًا قبل معرفة الكيفية، فإن كانت الشيفرة البرمجية في البداية فإن غرضـها يكـون واضـحًا مباشـرة. قـد لا نحتـاج أيضًـا لقـراءة الدوال، خاصةً إن كانت وظائفها واضحة من مسمياتها.

3.2.3 دليل نمط كتابة الشيفرة

يحتوي دليل أنماط كتابة الشيفرة (التكويد) قواعد عامة حـول كيفيـة كتابـة شـيفرة برمجيـة. مثلا، الأقـواس المستخدمة، وعـدد مسـافات البادئـة، وأقصـى طـول للسـطر، ...إلخ. والعديـد من التفاصـيل الدقيقـة. فعنـدما يستخدم جميع أعضاء الفريق الدليل نفسه لنمط الكتابة، فستبدو الشيفرة البرمجية موحدة بغض النظر عمَّن قام بكتابتها.

يمكن لأي فريق الكتابة بالنمط الخاص به، لكن لا حاجة لذلك لوجود العديـد من المعـايير العالميـة للاختيـار منها بل يجب اتباع نمط موحَّد بين كامل الفريق.

بعض الخيارات الشهيرة:

- Google JavaScript Style Guide
- Airbnb JavaScript Style Guide
- Idiomatic.JS
- StandardJS
- وغيرها

في حال كنت مطورًا مبتدئًا، ابدأ بالشرح المزود هنا. ثم يمكنك الانتقال إلى دليل آخـر ممـا ذكرنـاه لتتعـرف على المزيد من التفاصيل وتختار الأسلوب الأنسب لك.

3.2.4 منقحات الصياغة التلقائية Automated Linters

منقحات الصياغة (Linters): هي عبارة عن أدوات يمكنها فحص نمط الشيفرة البرمجية تلقائيًا واقتراح تعديلات لتحسينها. الأمر الذي يجعلها مفيدة أكثر هو قدرتها على العثور على بعض الأخطاء، مثل الخطأ باسم متغير أو دالة ما، لهذا فإنه من المستحسن استخدام منقح صياغة (Linter) حتى لو لم تُرد اتباع نمط كتابة معين.

هنا بعض أدوات تنقيح الصياغة المعروفة مثل:

- JSLint – تُعد من أدوات تنقيح الصياغة الأولى.
- JSHint – تحوي إعدادات أكثر من JSLint.
- ESLint – الأحدث تقريبا.

كلها تؤدي الغرض ذاته، والكاتب هنا يستخدم ESLint.

معظم هذه الأدوات تكون مدمجة مع العديد من المحررات الشهيرة: يجب عليك أن تُفَعِّل الإضافة في المحرر وتحدد نمط الكتابة الذي تريده.

مثلا، لاستخدام ESLint اتَّبع ما يلي:

1. ثبت Node.js.
2. ثبت ESLint باستخدام الأمر `npm install -g eslint` (يُعد npm مُثَبِّت حزم JavaScript).
3. أنشئ ملف إعداد وسمِّه "`.eslintrc`" في ملف مشروع JavaScript الرئيسي (الملف الذي يحتوي على جميع الملفات).
4. ثبت/فعِّل الإضافة لمحررك الذي يدعم ESLint، فمعظم المحررات تدعم ESLint.

هنا مثال على ملف "`.eslintrc`":

```
{
  "extends": "eslint:recommended",
  "env": {
    "browser": true,
    "node": true,
    "es6": true
  },
  "rules": {
    "no-console": 0,
```

```
      "indent": ["warning", 2]
    }
  }
```

التعليمـــة `"extends"` هنــا تعــني أن الإعــداد يعتمــد على مجموعــة إعــدادات ESLint الافتراضــية "eslint:recommended". يُمكننا تعديل الإعدادات التي نريدها لاحقًا. يمكن أيضا تنزيل مجموعة قواعد نمـط الكتابة أو التكويد وتوسيعها بدلًا من ذلك. انظـر في الرابـط eslint.org/docs/user-guide/getting-started للمزيد من التفاصيل حول كيفية التثبيت. تحتوي بعض المحررات على منقح صياغة مدمج فيها إلا أنَّهـا ليسـت بدقة ESLint.

3.2.5 الخلاصة

تهدف جميع قواعد بناء التعليمات والتعابير في هذا الفصل (وفي باقي مراجع أنماط التكويد) لرفع مسـتوى سهولة قراءة الشيفرات وجميع القواعد قابلة للنقاش طبعًا.

عند التفكير في كتابـة شـيفرة برمجيـة بشـكل أفضـل، يجب أن نسـأل أنفسـنا: "مـا الـذي يجعـل الشـيفرة البرمجية أسهل للقراءة والفهم؟" و "ما الذي قد يساعدنا لتجنب الأخطاء؟" يوجد العديـد من الأشـياء الـتي يجب الانتباه لها أثناء اختيار نمط تكويد معين.

سـيتيح لـك قـراءة العديـد من أنمـاط الكتابـة والتكويـد معرفـة أحـدث الأفكـار عن أنمـاط التكويـد وأفضـل الممارسات الشائعة.

3.2.6 تمارين

ا. نمط تكويد سيئ

الأهمية: ★★★★☆

ما الخطأ في نمط التكويد أدناه؟

```
function pow(x,n)
{
  let result=1;
  for(let i=0;i<n;i++) {result*=x;}
  return result;
}

let x=prompt("x?",''), n=prompt("n?",'')
if (n<=0)
```

```
{
  alert(`Power ${n} is not supported, please enter an integer number
greater than zero`);
}
else
{
  alert(pow(x,n))
}
```

قم بإصلاحه.

الحل:

يمكنك ملاحظة ما يلي:

```
function pow(x,n)  // <- لا يوجد مسافات بين المُعاملات
{  // <- قوس الفتح في سطر مستقل
  let result=1;   // <- = عدم وجود مسافات قبل أو بعد
  for(let i=0;i<n;i++) {result*=x;}   // <- لا يوجد مسافات
  // يجب أن يكون محتوى { ... } في سطر جديد
  return result;
}

let x=prompt("x?",''), n=prompt("n?",'') // <-- ممكنة تقنيا،
// لكن يُفَضَّل جعلها في سطرين، بالإضافة إلى عدم وجود مسافات وعدم وجود ;
//
if (n<0)  // <-  لا يوجد مسافات بين (n < 0) ، ويجب وجود سطر فارغ قبلها
{   // <- قوس الفتح في سطر مستقل
  // يمكن فصل الأسطر الطويلة في الأسفل حتى تصبح سهلة القراءة
  alert(`Power ${n} is not supported, please enter an integer number
greater than zero`);
}
else // <- "} else {" يمكن كتابتها في سطر واحد هكذا:
{
  alert(pow(x,n))  //  ; لا يوجد مسافات ولا يوجد
}
```

بعد تصحيح الأخطاء تصبح الشيفرة البرمجية كما يلي:

```
function pow(x, n) {
  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

let x = prompt("x?", "");
let n = prompt("n?", "");

if (n < 0) {
  alert(`Power ${n} is not supported,
    please enter an integer number greater than zero`);
} else {
  alert( pow(x, n) );
}
```

3.3 التعليقات

رأينا في فصل بنية الشيفرة البرمجية أنه يمكن أن تكون التعليقات ذات سطر واحد: وتبدأ ب `//` أو أن تمتد على عدَّة أسطر: `/* ... */`.

تُستخدَم التعليقات لوصف الغرض من الشيفرة البرمجية وآلية عملها. في أول مرة، قد تبدو التعليقات مهمة، لكن المبرمجين المبتدئين يستخدمونها بطريقة غير صحيحة.

3.3.1 التعليقات السيئة

يستخدم المبتدئون التعليقات لشرح "ماذا يحدث في الشيفرة البرمجية". كما يلي:

```
// ستقوم هذه الشيفرة البرمجية بهذا الأمر (...) وذلك الأمر
// وربما أشياء أخرى
very;
complex;
code;
```

يجب أن يكون عدد التعليقات التوضيحية أقل في الشيفرات البرمجية الجيدة، لأن الشيفرة البرمجية يجب أن تكون مفهومة بدون تعليقات.

يوجد قاعدة مهمة حول هذا الأمر:

"إن كانت الشيفرة البرمجية غير مفهومة لدرجة أنها تحتاج إلى التعليقات لشرحها، يجب أن تُكتَب من جديد بدلًا من التعليقات التوضيحية الكثيرة".

ا. طريقة: أخرج الدوال

أحيانا، يكون استبدال جزء من الشيفرة البرمجية أمرًا مهمًا كما يلي:

```
function showPrimes(n) {
  nextPrime:
  for (let i = 2; i < n; i++) {

    // check if i is a prime number
    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

```

```
    alert(i);
  }
}
```

البديل الأفضل هو إخراج دالة `isPrime`:

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;

    alert(i);
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if (n % i == 0) return false;
  }

  return true;
}
```

بهذه الطريقة يمكننا فهم الشيفرة البرمجية بشكل أفضل لأن الدالة أصبحت بديلة عن التعليق. هذه الشيفرة البرمجية تسمى شيفرة برمجية موصوفة بحد ذاتها.

ب. طريقة: أنشئ دوال

إن كان هناك شيفرة برمجية طويلة كما يلي:

```
// here we add whiskey
for(let i = 0; i < 10; i++) {
  let drop = getWhiskey();
  smell(drop);
  add(drop, glass);
}

// here we add juice
```

```
for(let t = 0; t < 3; t++) {
  let tomato = getTomato();
  examine(tomato);
  let juice = press(tomato);
  add(juice, glass);
}

// ...
```

فيُفضل إخراجه إلى دالة كما يلي:

```
addWhiskey(glass);
addJuice(glass);

function addWhiskey(container) {
  for(let i = 0; i < 10; i++) {
    let drop = getWhiskey();
    //...
  }
}

function addJuice(container) {
  for(let t = 0; t < 3; t++) {
    let tomato = getTomato();
    //...
  }
}
```

مـرة أخـرى، يجب أن يكـون عمـل الدالـة واضـحًا من اسـمها وموضـعها وهكـذا لن نحتـاج إلى الكثـير من التعليقات. وستكون بنية الشيفرة البرمجية أفضل عند تجزئتها. وسيصبح من الواضـح مـاهي آليـة عمـل الدالـة، وماذا تستقبل وماذا تعيد.

في الواقـع لا يمكننـا تجنب التعليقـات التوضـيحية فهنـاك بعض الخوارزميـات المعقـدة وأدوات التعـديل والتحسين الذكية (smart tweaks) التي تحتاج توضيحًا لكن يجب أن نحاول أن نجعل الشيفرة البرمجية سهلة القراءة وأن تشرح نفسه بنفسها.

3.3.2 التعليقات الجيدة

إن كانت التعليقات التوضيحية سيئة، فأي التعليقات هو الجيد؟

وصف الهيكلة

يُعطي نظرة عامة عن المكونات، وطريقة تفاعلها، وطريقة تدفق التحكم في مختلف الحالات، وما إلى ذلك وهو باختصار ما يُسمَى بالنظرة الشاملة للشيفرة (bird's eye view of the code). يوجد لغة خاصة تدعى UML لبناء رسوم هيكلية عالية المستوى لشرح الشيفرة البرمجية. هذه اللغة تستحق التعلم فعلا.

توثيق مُعامِلات الدوال واستخدامها

يوجد تركيب خاص بالجمل البرمجية يُسمَّى JSDoc لتوثيق الدوال: استخدامها، مُعامِلاتها، والقيم التي تُرجِعُها. مثال:

```
/**
 * Returns x raised to the n-th power.
 *
 * @param {number} x The number to raise.
 * @param {number} n The power, must be a natural number.
 * @return {number} x raised to the n-th power.
 */
function pow(x, n) {
  ...
}
```

تتيح لنا هذه التعليقات فهم الغرض من هذه الدالة واستخدامها مباشرة دون الحاجة للنظر إلى تفاصيل الشيفرة البرمجية فيها. يمكن للعديد من المُحررات فهم هذه التعليقات أيضا واستخدامها في الإكمال التلقائي وبعض حالات فحص الشيفرة التلقائي.

يوجد أيضا بعض الأدوات مثل JSDoc 3 والتي يمكنها توليد توثيق HTML من هذه التعليقات. للمزيد من المعلومات حول JSDoc اقرأ هنا http://usejsdoc.org.

لِمَ حُلَّت هذه المهمة بهذه الطريقة؟

ما هو مكتوب هو مهم، لكن قد يكون ما هو غير مكتوب أكثر أهمية لفهم ما يجري. لا تستطيع الشيفرة البرمجية الإجابة عن السؤال "لم حُلَّت هذه المهمة بهذه الطريقة؟".

إن كان هناك العديد من الطرائق لحل هذه المهمة. لم اختيرَت هذه الطريقة؟ خاصة عندما لا تكون هذه الطريقة هي الأفضل.

يمكن حدوث التالي دون تعليقات توضح السبب:

1. ستفتح الشيفرة البرمجية المكتوبة منذ وقت طويل (أنت أو أي شخص آخر) وسترى أنها "غير مناسبة".
2. ستفكر: "كم كنت غبيا حينها، وكم أنا ذكي الآن"، ثم ستُعيد كتابة الشيفرة البرمجية باستخدام البديل الذي تراه أكثر صحة وملائمة.
3. فكرة إعادة الكتابة تكون سهلة، لكن عند القيام بها تكتشف أن الحل "الأمثل" لا يتناسب مع الغرض المطلوب. حينها قد تتذكر السبب لأنَّك جربت الأمر ذاته مسبقًا بالفعل ولم يفلح ثم ستعود إلى الخيار السابق لكن بعد ضياع الوقت.

التعليقات التي توضح سبب استخدام طريقة ما مهمةٌ جدًا فهي تساعد على الاستمرار بالتطوير مباشرة.

أي ميزات غير ملحوظة في الشيفرة البرمجية ومكان استخدامها

إن احتوت الشيفرة البرمجية على شيء غير ملحوظ أو شيء يُدرَك حدسيا، فيجب تعليقه.

3.3.3 الخلاصة

التعليقات هي علامة مهمة عن المطوِّر الجيد: وجودها أو عدم وجودها، إذ تتيح التعليقات الجيدة صيانة الشيفرة البرمجية جيدًا، عند العودة لها بعد مدة من الوقت واستخدامها بكفاءة.

علِّق ما يلي:

- الهيكلة العامة، النظرة عالية المستوى.
- استخدام الدوال.
- الحلول المهمة، خاصة تلك الغير ملحوظة مباشرة.

تجنب التعليقات:

- التي تشرح آلية عمل الشيفرة البرمجية والغرض منها.
- ضع مثل هذه التعليقات عندما يكون من المستحيل جعل الشيفرة البرمجية سهلة القراءة وتصف نفسها وتحتاج لتوضيح فقط

تستخدم التعليقات أيضا للأدوات التي تقوم بالتوثيق تلقائيًا مثل JSDoc3: تقرأ هذه الأدوات التعليقات وتولِّد ملفات HTML أو بأي صيغة أخرى.

3.4 شيفرة النينجا البرمجية

"التعلم بدون تفكير يضيع الجهد، والتفكير بدون تعلُّم مجرد مخاطرة"

استخدم مـبرمجو النينجا القـدماء Programmer ninjas هـذه الحيـل لشـحذ عقـول مراجعـو الشـيفرات البرمجيـة. يبحث معلمـو مراجعـة الشـيفرات البرمجيـة عن هـذه الحيـل في مهـام الاختبـار. أحيانًـا، يسـتخدم المبرمجون المبتدئون هذه الحيـل بصـورة أفضـل من مـبرمجي النينجـا. اقرأهـا جيـدًا ثم حـدِّد من أنت - نينجـا، مبتدئ، أو مُراجِع شيفرات برمجية؟

> **موازنة تم اكتشافها**
>
> يحاول الكثيرون اتباع طرائق النينجا ولكن لا يفلح إلا القليل.

3.4.1 البلاغة في الإيجاز

اختصر الشيفرة البرمجية قدر الإمكان وأظهر مدى ذكائك ولتَقُدكَ ميزات اللغة الخفية.

مثال: ألق نظرة على العامل الثلاثي التالي `'?'`:

```
// مأخوذ من مكتبة جافاسكربت شهيرة
i = i ? i < 0 ? Math.max(0, len + i) : i : 0;
```

إن كتبت بهذه الطريقة، فإن أي مطور آخر يقرأ هذا السطر ويحاول فهم ما هي قيمة `i` سيأتي إليك باحثًا عن الإجابة. أخبرهم في ذلك الحين أن الاختصار أفضل دومًا ودلهم على طريق النينجا.

3.4.2 المتغيرات ذات الحرف الواحد

استخدام المتغيرات ذات الحرف الواحد هي طريقة أخرى لكتابة الشيفرة البرمجية بصـورة أسـرع مثـل a، أو b، أو c. يختبئ المتغير ذو الحرف الواحـد داخـل الشـيفرة البرمجيـة كمـا يختـبئ النينجـا الحقيقي في الغابـة. لن يتمكن أي شخص من العثور عليه بسهولة باستخدام "البحث" المُدمَج مع المحرر. حتى إن عثر شخص مـا عليـه فلن يفهم ما معنى المتغير a أو b.

حديثنا عن المتغيرات المكونة من حرف واحد يذكِّرنا بالحلقات وهنا الجزء الممتع، فمـبرمج النينجـا الحقيقي لن يستخدم المتغير `i` كعداد للحلقة `"for"` بل سيستخدمه في أي مكان عدا هنـا إذ يوجـد العديـد من الأحـرف الغريبة لاستخدامها كعداد الحلقة بدلا منه مثل x أو y. متغير بحرف غريب عدادًا للحلقـة يُعـد مناسـبًا إن كـانت الحلقة تمتد إلى صفحة أو صفحتين. فإن أتى شخص ما للتعمق في هـذه الحلقـة فلن يعـرف أن المتغيـر x هـو عدَّاد هذه الحلقة.

3.4.3 استخدم الاختصارات

إن كانت قوانين الفريق تمنع استخدام المتغيرات ذات الحرف الواحد أو المتغيرات الغامضة، اختصر أسماء المتغيرات.

- `list` ← `lst`
- `userAgent` ← `ua`
- `browser` ← `brsr`
- وهكذا...

سيفهم شخص فطن فقط الأسماء المستخدمة. حاول اختصار كل شيء إذ يجب أن يُكمل تطوير شيفرتك البرمجية شخص مؤهل فقط .

3.4.4 حلق عاليًا واستخدم التجريد

> "المربع العظيم لا يحوي أي زوايا
> السفينة الأعظم تكتمل أخيرا
> النغمة العظيمة نادرة الصوت
> الصورة العظيمة لا تحوي أي شكل"
>
> — تاو تي تشينغ

حاول اختيار الكلمة الأكثر تجريدًا عند اختيار اسم ما مثل `obj`، و `data`، و `value`، و `item`، و `elem` وهكذا.

- الاسم الأمثل لمتغير هو `data`. استخدمه حيثما استطعت، فبالحقيقة، كل متغير يحتوي على بيانات "data". لكن، ماذا إن كان هناك متغير بالاسم `data` بالفعل؟ جرب الاسم `value`، فهو عالمي أيضًا، فكل متغير يحتوي على قيمة.
- سمِّ المتغير حسب نوعه: `str`، `num`...: جرب هذه الطريقة. قد يتساءل أي شخص مبتدئ حول إن كانت مثل هذه الأسماء مفيدة لمبرمج النينجا؟ بالفعل هي مفيدة إذ اسم المتغير يُخبر عن شيء ما. فيدل اسم المتغير في هذه الطريقة عن نوع البيانات التي يحويها: نص، رقم أو أي نوع آخر لكن عندما يحاول شخص آخر فهم الشيفرة البرمجية، سيُصدَم بعدم توفر معلومات كافية لفهمها! وسيفشل تمامًا في تعديل شيفرتِك البرمجية. يمكن معرفة قيمة المتغير عبر تتبع الشيفرة البرمجية. لكن ما معنى المتغير؟ وماهو النص أو الرقم الذي يخزنه؟ لا يوجد طريقة لمعرفة ذلك بدون تأمل جيد!

- لكن ماذا إن لم يكن هناك المزيد من هـذه الأسـماء؟ فقـط أضـف رقمًـا إليهـا مثـل: `data1`، `item2`، `elem5`...

3.4.5 اختبار الملاحظة

يمكن لِمبرمج قـوي الملاحظـة فقـط فهم شـيفرتُك البرمجيـة. لكن، كيـف تختـبر ذلـك؟ إحـدى الطـرق هي استخدام متغيرات بأسماء متشابهة مثل `date` و `data`. اخلُـط مـا بين المتغـيرين حيث يمكن ذلـك فستصـبح قراءة شيفرة برمجية بهذه الطريقة مستحيلة. وعند وجود خطأ كتابي، فسيَعلق القارئ لوقت طويل.

3.4.6 مرادفات ذكية

"أصعب شيء هو العثور على قطة سوداء في غرفة مظلمة، خاصةً عند عدم وجود قطة!"

استخدام أسماء متشابهة لنفس الأشياء يجعل الحياة أكثر تشويقا ويُظهِر مدى إبداعك للجمهور. على سـبيل المثال، لنأخذ بالحسبان بوادِئ الدوال فإن كانت الدالة تعرض رسـالة على الشاشـة، ابـدأها بالكلمـة `display`… مثل `displayMessage`. وإن كانت دالة أخرى تعـرض شـيئًا آخـر على الشاشـة مثـل اسـم المسـتخدم ابـدأها بالكلمة `show`… مثل `showName`. اجعل من يقـرأ الشـيفرة يظن أن هنـاك اختلاف خفي بين مثـل هـذه الـدوال بينما لا يوجد أي اختلاف. اتفق مع فريق مبرمجي النينجا: إن بدأ أحمد دالـة العـرض بالكلمـة `display...` في شـيفرته البرمجيـة، فيمكن لمحمـد اسـتخدام `render...` ويمكن لِأمـل اسـتخدام `paint...`. لاحـظ كيـف ستصبح الشيفرة البرمجية أكثر اختلافًا وتشويقًا.

الآن إلى خدعة القبعة!

اسـتخدم نفس البادئـةِ لـدالتين مهمـتين بوظيفـتين مختلفـتين! مثلا، الدالـة "`printPage(page)`" ستستخدم طابعة، بينما الدالة "`printText(text)`" ستضع النص على الشاشـة. هكـذا تجعـل القـارئ يفكـر جيدًا بالدالة المُسماة `printMessage`: "أين تضع هذه الدالـة الرسـالة؟ إلى الطابعـة أو على الشاشـة؟". لجعـل الأمر أوضح، يجب أن تضع الدالة "`printMessage(message)`" الرسالة في النافذة الجديدة! (وظيفة مهمـة بطريقة مختلفة).

3.4.7 أعد استخدام الأسماء

عند تقسيم الكل، تحتاج الأجزاء إلى أسماء.

يوجد بالفعل أسماء كافية للجميع.

يجب أن تعرف متى تتوقف.

— تاو تي تشينغ

ضع متغيرًا جديدًا عند الحاجة فقط، وأعد استخدام الأسماء الموجودة عوضًا عن ذلك. اكتب قيمًا جديدة إلى هذه المتغيرات. وفي الدوال، حاول استخدام المتغيرات المُمَرَّرة إلى الدالة كمُعاملات. ستُصعِّب بهذه الطريقة معرفة ما في المتغير الآن ومن أين أتى هذا المتغير أيضا. الغرض من هذه الطريقة هو تطوير حدس وذاكرة قارِئ الشيفرة البرمجية إذ سيحتاج الشخص ذو الحدس الضعيف إلى تحليل الشيفرة البرمجية سطرًا تلو الآخر وتَعقَّب التغييرات في كل جزء من أجزاء الشيفرة البرمجية.

إحدى الطرق المتقدمة هي بتبديل قيمة متغير ما مكان محتوى آخر مختلف تمامًا عن السابق خفيةً داخل حلقة أو دالة.

على سبيل المثال:

```
function ninjaFunction(elem) {
  // elem عشرون سطرًا من الشيفرة يتعامل مع المتغير

  elem = clone(elem);

  // elem عشرون سطرًا إضافيًا يتعامل الآن مع نسخة من المتغير
}
```

سيتفاجأ المبرمج الآخر الذي يريد التعامل مع `elem` في الجزء الثاني من الدالة لأنه سيكتشف فقط أثناء التنفيذ والتعقب وبعد تفحص الشيفرة بأنه يتعامل مع `elem` بعد استنساخه!

3.4.8 الشرطة السفلية للمتعة

ضع الشرطات السفلية _ و __ قبل أسماء المتغيرات. مثل "`_name`" أو "`__value`". سيكون جيدا إن كنت أنت فقط من تعرف معنى ذلك، أو استخدمها للمتعة فقط دون معنى معين أو استخدم معانٍ مختلفة في أماكن مختلفة. بهذا ترمي عصفورين بحجر. أولا، تصبح الشيفرة البرمجية أطول وأصعب للقراءة. ثانيا، سيأخذ القارئ وقتا لمعرفة معنى الشرطات السفلية.

يضع مبرمج النينجا الذكي الشرطات السفلية في مكان معين من الشيفرة البرمجية ويتجنبها في أماكن أخرى. هذا يجعل الشيفرة البرمجية أكثر هشاشة ويزيد من احتمالية الخطأ المستقبلي.

3.4.9 أظهر حبك

أظهر للجميع مدى روعة المكونات التي تستخدمها! ستُبهِر الأسماء مثل `superElement`، `megaFrame`، و `niceItem` القارئ. يمكن كتابة: "`super..`"، و "`mega..`"، و"`nice..`" لكن بطريقة لا تُظهِر تفصيلاً حول المعنى. قد يبحث القارئ عن معانٍ خفية ويتأمل لساعة أو اثنتين من وقته.

3.4.10 داخِل المتغيرات الخارجية

"عندما تكون في الضوء، لا يمكنك رؤية شيء في الظلام
عندما تكون في الظلام، يمكنك رؤية كل شيء في الضوء"

— غوان ين زي

استخدم أسماء المتغيرات نفسها داخل وخارج الدوال. ببساطة، لا حاجة لاختراع أسماء جديدة.

```
let user = authenticateUser();

function render() {
  let user = anotherValue();
  ...
  ...many lines...
  ...
  ... // <-- يريد مبرمجٌ ما التعامل مع المتغير user هنا
  ...
}
```

سيفشل المبرمج الذي يقفز بِداخل الدالة `render` في ملاحظة أن هناك متغير `user` محلي يغطي على المتغير الخارجي.

سيتعامل مع `user` مُعتبِرًا أنه المتغير الخارجي نفسه، وبذلك ستكون نتائج الدالة "`authenticateUser()`" غير متوقعة وسينجح الفخ.

3.4.11 آثار جانبية في كل مكان

يوجد بعض الدوال التي تبدو كأنها لا تقوم بشيء مثل، "`isReady()`"، و "`checkPermission()`"، و "`findTags()`" ...إلخ. يُتَوَقَّع أنها تقوم بعمليات حسابية، تُوجِد وتُرجِع قيم البيانات بدون تغيير شيء خارج نطاقها. بمعنى آخر، بدون آثار جانبية.

إضافة حدث مفيد إليها سيكون خدعة جيدة، بجانب مهمتها الرئيسية.

بالتأكيد، ستوسّع نظرة الدهشة التي على وجه قارئ الشيفرة -عندما يرى دالة بالاسم "`is..`"، أو "`check..`"، أو "`find..`" وتُغَيِّر شيئا ما - آفاق المنطق لديك.

طريقة أخرى هي إرجاع نتيجة غير قياسية.

أظهر ذكائك! اجعل الدالة `checkPermission` تُرجِع كائنًا معقدًا يحتوي على نتيجة الفحص المُراد من الدالة. سَيتساءل المطورون الذين سيكتبون "`if (checkPermission(..))`" عن سبب عدم عملها. حينها أخبرهم أن يقرؤوا التوثيق ووجههم لهذا الفصل :).

3.4.12 دوال قوية!

لا تجعل الدالة محدودة بما يتضمنه اسمها. توسَّع أكثر. على سبيل المثال، يمكن للدالة "`validateEmail(email)`" (بالإضافة إلى فحص صحة البريد الإلكتروني) عرض رسالة خطأ وطلب إعادة إدخال البريد الإلكتروني.

لا يجب أن تكون باقي الوظائف واضحة من مسمى الدالة. مبرمج النينجا الحقيقي لن يجعل هذه الوظائف واضحة حتى في الشيفرة البرمجية.

3.4.13 الخلاصة

جميع الملاحظات السابقة هي من شيفرات برمجية حقيقية مكتوبة بواسطة مبرمجين محترفين، ربما أكثر احترافية منك ;)

- اتبع بعض الملاحظات وستكون شيفرتك البرمجية مليئة بالمفاجآت.
- اتَّبِع اغلبها وستكون شيفرتك البرمجية حقا ملك لك أنت فقط، لن يريد أي شخص تغييرها.
- اتَّبِع جميعها وستصبح شيفرتك البرمجية درسًا قيمًا للمطورين المبتدئين الباحثين عن إرشاد.

3.5 الاختبار الآلي باستخدام mocha

يُستخدَم الاختبار الآلي في الكثير من المهام، كما يستخدم بكثرة في المشاريع الحقيقية.

3.5.1 لم نحتاج الاختبارات؟

عند كتابة دالة، يمكننا تخيل ما يجب أن تقوم به: ما هي المعاملات التي تعطي نتائج معينة. يمكننا فحص الدالة أثناء التطوير من خلال تشغيلها وموازنة مخرجاتها مع ما هو متوقع. مثلا يمكننا القيام بذلك في الطرفية.

إن كان هناك خطأ، فإننا نُصلِح الشيفرة البرمجية، ونُعيد تشغيلها، ونفحص النتائج. وهكذا حتى تصبح صحيحة. لكن هذه الطريقة "إعادة التشغيل" غير مثالية.

عند اختبار شيفرة برمجية عن طريق إعادة التشغيل اليدوية، فمن السهل نسيان شيءٍ ما. على سبيل المثال، عند إنشاء الدالة `f` نكتب فيها بعض الشيفرات البرمجية، ثم نَفحصها: "`f(1)`" تعمل لكن "`f(2)`" لا تعمل. أصلِح الشيفرة حتى تعمل "`f(2)`". ثم تبدو الدالة كأنها مكتملة، لكننا ننسى إعادة اختبار "`f(1)`" مما قد يؤدي إلى خطأ.

هذا الأمر وارد بكثرة. فعند تطوير أي شيء، نُبقِي العديد من الاحتمالات في الحسبان لكنه من الصعب توقع أن يختبر المبرمج جميع هذه الحالات يدويا بعد كل تغيير، فيصبح من السهل إصلاح شيء ما وإفساد شيء آخر.

يعني الاختبار الآلي أن الاختبارات تُكتب مستقلة، بالإضافة إلى الشيفرة البرمجية. تشغِّل هذه الاختبارات الدوال بعدة طرائق وتوازنها مع النتائج المتوقعة.

3.5.2 التطوير المستند إلى السلوك

لنبدأ بتقنية تسمى التطوير المستند إلى السلوك (Behavior Driven Development أو كاختصار BDD).

هذه التقنية BDD هي 3 في 1: اختبارات وتوثيق وأمثلة. سنجرب حالة تطوير عملية لفهم BDD.

3.5.3 تطوير الدالة "pow": الوصف

لنفترض أننا نريد إنشاء الدالة `pow(x, n)` التي ترفع الأساس `x` إلى القوة `n`. مع الأخذ بالحسبان أن $n \geq 0$.

هذه المهمة هي مجرد مثال: المعامل `**` يقوم بهذه العملية في JavaScript، لكننا نركز هنا على تدفق التطوير الذي يمكن تطبيقه على مهام أكثر تعقيدًا.

يمكننا تخيل ووصف ما يجب أن تقوم به الدالة `pow` قبل إنشاء شيفرتها البرمجية. هذا الوصف يُسمى "specification" أو باختصار "spec" ويحتوي على وصف حالات الاستخدام بالإضافة إلى اختبارات لهذه الحالات كالتالي:

```
describe("pow", function() {

  it("raises to n-th power", function() {
    assert.equal(pow(2, 3), 8);
  });

});
```

تحتوي المواصفات على 3 أجزاء رئيسية كما ترى في الأعلى:

```
describe("title", function() { ... })
```

ماهي الوظيفة التي نصفها، في هذه الحالة، نحن نصف الدالة pow. وتُستخدَم لضم مجموعة من الاختبارات، أي أجزاء `it`.

```
it("use case description", function() { ... })
```

نصف (نحن بطريقة مقروءة للبشر) حالة الاستخدام المخصصة في عنوان `it`، والمعامل الآخر عبارة عن دالة تفحص هذه الدالة.

```
assert.equal(value1, value2)
```

الشيفرة البرمجية بداخل `it` يجب أن تُنَفَّذ بدون أخطاء في حال كان التنفيذ صحيحًا.

تستخدم الدوال `assert.*` لِفحص ما إن كانت الدالة pow تعمل بالشكل المتوقع أم لا. تستخدم إحدى هذه الدوال في هذا المثال، `assert.equal`، والتي توازن معاملَين وتُرجِع خطأ في حال عدم تساويهما. في المثال تفحص هذه الدالة إن كانت نتيجة تنفيذ الدالة `pow(2, 3)` تساوي 8. كما يوجد العديد من أنواع التحقق والموازنة والتي سنُضيفها لاحقًا. يمكن تنفيذ الوصف، وسينفِّذ الفحص الموجود بداخل `it` كما سنرى لاحقا.

3.5.4 تدفق التطوير

يبدو تدفق التطوير غالبًا كما يلي:

1. يُكتَب الوصف الأولي مع فحص للوظيفة الرئيسية.
2. يُنشَأ تنفيذ أولي.
3. للتأكد من صحة عمل التنفيذ، نُشَغِّل إطار التقييم Mocha الذي يُشَغِّل الوصف. ستظهر أخطاء في حال عدم اكتمال الوظائف. نُصحِّح الأخطاء حتى يصبح كل شيء صحيحًا.

4. هكذا أصبح لدينا تنفيذ مبدئي يعمل كالمطلوب بالإضافة إلى فحصه.
5. نضيف المزيد من حالات الاستخدام للوصف، ربما بعض هذه الميزات ليس مضمنًا في التنفيذ بعد. حينها يبدأ الاختبار بالفشل.
6. عُد للخطوة 3 وحدِّث التنفيذ إلى أن تختفي كل الأخطاء.
7. كرِّر الخطوات 3-6 حتى تجهز كل الوظائف.

إذًا، تُعدُ عملية التطوير تكرارية. نكتب الوصف، ننفذه، نتأكد من اجتياز التنفيذ للفحص، ثم نكتب المزيد من الاختبارات، نتأكد من صحة عملها. حتى نحصل على تنفيذ صحيح مع اختباراته في الأخير.

لنُجرب تدفق التطوير هذا على حالتنا العملية.

الخطوة 1 أصبحت جاهزة: لدينا وصفًا مبدئيًّا للدالة pow. الآن وقبل التنفيذ، لِنستخدم بعض مكتبات JavaScript لتشغيل الاختبار حتى نتأكد من إن كانت تعمل (لن تعمل).

3.5.5 المواصفات أثناء التنفيذ

سنستخدم في هذا الشرح مكتبات JavaScript التالية للاختبار:

- Mocha - الإطار الرئيسي الذي يوفر دوال الفحص الأكثر استخدامًا ما يشمل `describe` و `it` بالإضافة إلى الدوال الرئيسية التي تُشَغِّل الاختبار.
- Chai - المكتبة المحتوية على دوال توكيدية تتيح لنا استخدام العديد من هذه الدوال، ونحتاج الآن `assert.equal` فقط.
- Sinon - مكتبة للتجسس على الدوال، ومحاكاة الدوال المدمجة، والمزيد؛ سنحتاج هذه المكتبة لاحقًا.

تُعد هذه المكاتب مفيدة للاختبار في كل من المتصفح والخادم. سنأخذ بعين الاعتبار هنا جهة المتصفح. صفحة HTML كاملة مع هذه المكاتب ووصف الدالة pow:

```
<!DOCTYPE html>
<html>
<head>
  <!-- add mocha css, to show results -->
  <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.css">
  <!-- add mocha framework code -->
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.js"></
script>
```

```
  <script>
mocha.setup('bdd'); // minimal setup
  </script>
  <!-- add chai -->
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js"></
script>
  <script>
// chai has a lot of stuff, let's make assert global
let assert = chai.assert;
  </script>
</head>
<body>
<script>
function pow(x, n) {
/* function code is to be written, empty now */
}
</script>

<!-- the script with tests (describe, it...) -->
<script src="test.js"></script>

<!-- the element with id="mocha" will contain test results -->
<div id="mocha"></div>

<!-- run tests! -->
<script>
mocha.run();
</script>
</body>

</html>
```

يمكن تقسيم الصفحة إلى 5 أجزاء:

1. <head> - لإضافة مكاتب وأنماط للاختبارات.
2. <script> يحتوي على الدالة التي سيتم اختبارها، في هذا المثال الشيفرة البرمجية للدالة pow.

3. الاختبـــــار - عبــــارة عن ســـــكربت خـــــارجي في هـــــذا المثــــال test.js يحتـــــوي على describe("pow", ...) الموضح سابقا.
4. عنصر HTML <div id="mocha"> والذي سيُستخدم بواسطة Mocha لعرض النتائج.
5. يتم بدء الاختبارات باستخدام الأمر mocha.run().

النتائج (تجربة حية):

passes: 0 failures: 1 duration: 0.03s 100%

pow

✖ raises to n-th power

```
AssertionError: expected undefined to equal 8
    at Context.<anonymous> (test.js:4:12)
```

يفشـل الاختبـار ويظهـر خطـأ في الـوقت الحـالي. يُعـد هـذا منطقيًـا: فالدالـة pow مـا زالت فارغـة، فـإن pow(2,3) تُرجع undefined بدلا من 8. لاحظ أن بإمكانك تشغيل أكثر من اختبار مختلف مسـتقبلًا، فهنـاك مُشغلات عالية المستوى مثل karma وغيرها.

3.5.6 التنفيذ الأولي

لنقم بتنفيذ مبسط للدالة pow حتى تعمل الاختبارات:

```
function pow(x, n) {
  return 8; // :) we cheat!
}
```

الآن ستعمل (اطلع على التجربة الحية)!

passes: 1 failures: 0 duration: 0.02s 100%

pow

✓ raises to n-th power

3.5.7 تطوير الوصف

ما قمنا به هو غش فقط، لا تعمل الدالة كالمطلوب: إن قمنا بحساب `pow(3,4)` فسنحصل على قيمة غير صحيحة، لكنها ستجتاز الاختبارات.

هذه الحالة غير عملية، وتحدث بكثرة. الدالة تتجاوز الاختبارات لكن آلية عملها خاطئة. هذا يعني أن الوصف غير مثالي. ونحتاج لإضافة المزيد من حالات استخدام الدالة إليه. لنضف اختبارًا آخر للتأكد ما إن كان `pow(3, 4) = 81`.

يمكنُنا اختيار إحدى الطريقتين لتنظيم الاختبارات:

- الخيار الأول - إضافة `assert` إلى `it`:

```
describe("pow", function() {

  it("raises to n-th power", function() {
    assert.equal(pow(2, 3), 8);
    assert.equal(pow(3, 4), 81);
  });

});
```

- الخيار الآخر - عمل اختبارين:

```
describe("pow", function() {

  it("2 raised to power 3 is 8", function() {
    assert.equal(pow(2, 3), 8);
  });

  it("3 raised to power 3 is 27", function() {
    assert.equal(pow(3, 3), 27);
  });
```

```
});
```

يختلف المبدأ في أنه عند وجود خطأ في `assert`، فإن `it` تتوقف عن العمـل. لـذا ففي الخيـار الأول عنـد فشل `assert` الأولى فلن نرى مخرجات `assert` الأخرى. يُعد الخيـار الثـاني أفضـل للحصـول على المزيـد من المعلومات حول ما يحدث بعمل اختبارين منفصلين. بالإضافة إلى وجود قاعدة أخرى من الجيد اتباعها.

كل اختبار يفحص شيئًا واحدًا فقط. إن وجدنا اختبـارًا يفحص شـيئين مختلفين فمن الأفضـل فصـلُهما إلى اختبارين. لنكمل باستخدام الخيار الثاني.

النتائج (تجربة حية):

passes: 1 failures: 1 duration: 0.09s 100%

pow

✓ 2 raised to power 3 is 8

✖ 3 raised to power 4 is 81

```
AssertionError: expected 8 to equal 81
    at Context.<anonymous> (test.js:8:12)
```

سيفشل الاختبار الثاني كَالمتوقع. فَالدالة تُرجع دوما 8، بينما تتوقع الدالة `assert` النتيجة 27.

3.5.8 تطوير التنفيذ

لنكتب شيئا أكثر واقعية لاجتياز الاختبارات:

```
function pow(x, n) {
  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

للتأكد من صحة عمل الدالة K نختبرها لأكثر من قيمة. يمكننا القيام بـذلك باسـتخدام الحلقـة `for` بـدلًا من تكرار `it` يدويًا:

```
describe("pow", function() {

  function makeTest(x) {
    let expected = x * x * x;
    it(`${x} in the power 3 is ${expected}`, function() {
      assert.equal(pow(x, 3), expected);
    });
  }

  for (let x = 1; x <= 5; x++) {
    makeTest(x);
  }

});
```

النتيجة (تجربة حية):

passes: 5 failures: 0 duration: 0.01s 100%

pow
- ✓ 1 in the power 3 is 1
- ✓ 2 in the power 3 is 8
- ✓ 3 in the power 3 is 27
- ✓ 4 in the power 3 is 64
- ✓ 5 in the power 3 is 125

3.5.9 دالة describe متداخلة

سنضيف المزيد من الاختبارات، لكن، قبل ذلك لاحظ أنه يجب جمع الدالة المساعدة `makeTest` والدالة `for`. لن نَحتاج لاستخدام `makeTest` في باقي الاختبارات، نحتاج إليها فقط في `for`: لأن وظيفتهُما العامة هي فحص كيف ترفع الدالة `pow` قيمة ما إلى قوة معينة.

يتم جمع الدالتين باستخدام الدالة `describe` المتداخلة:

```
describe("pow", function() {

  describe("raises x to power 3", function() {
```

```
    function makeTest(x) {
      let expected = x * x * x;
      it(`${x} in the power 3 is ${expected}`, function() {
        assert.equal(pow(x, 3), expected);
      });
    }

    for (let x = 1; x <= 5; x++) {
      makeTest(x);
    }

  });

  // ... more tests to follow here, both describe and it can be added
});
```

تُعرِّف describe الداخلية مجموعة فرعية جديدة من الاختبارات. يمكن ملاحظة الإزاحة في المخرجات (اطلع على تجربة حية للمثال):

pow

raises x to power 3

✓ 1 in the power 3 is 1
✓ 2 in the power 3 is 8
✓ 3 in the power 3 is 27
✓ 4 in the power 3 is 64
✓ 5 in the power 3 is 125

يمكننا إضافة المزيد من دوال it و describe في الطبقة العلوية مع دوال مساعدة لها، هذه الدوال لن ترى الدالة makeTest.

انتبه إلى أنَّه يمكننا إعداد دوال before/after التي تُنَفَّذ قبل/بعد تنفيذ الاختبارات، كما يمكننا استخدام beforeEach/afterEach قبل/بعد كل it.

على سبيل المثال:

```
describe("test", function() {

  before(() => alert("Testing started – before all tests"));
  after(() => alert("Testing finished – after all tests"));

  beforeEach(() => alert("Before a test – enter a test"));
  afterEach(() => alert("After a test – exit a test"));

  it('test 1', () => alert(1));
  it('test 2', () => alert(2));

});
```

تسلسل التنفيذ سيكون كالتالي:

```
Testing started – before all tests (before)
Before a test – enter a test (beforeEach)
1
After a test – exit a test   (afterEach)
Before a test – enter a test (beforeEach)
2
After a test – exit a test   (afterEach)
Testing finished – after all tests (after)
```

تستخدم `beforeEach/afterEach` و `before/after` غالبا لتنفيذ الخطوات الأولية، العدادات التي تُخرج 0 أو للقيام بشيء ما بين الاختبارات أو مجموعة اختبارات.

3.5.10 توسيع الوصف

أصبحت الوظيفة الرئيسية للدالة `pow` مكتملة. تم تجهيز الدورة الأولى من التطوير. الآن يمكننا الاحتفال بالانتهاء من ذلك وبدء تطوير الدالة.

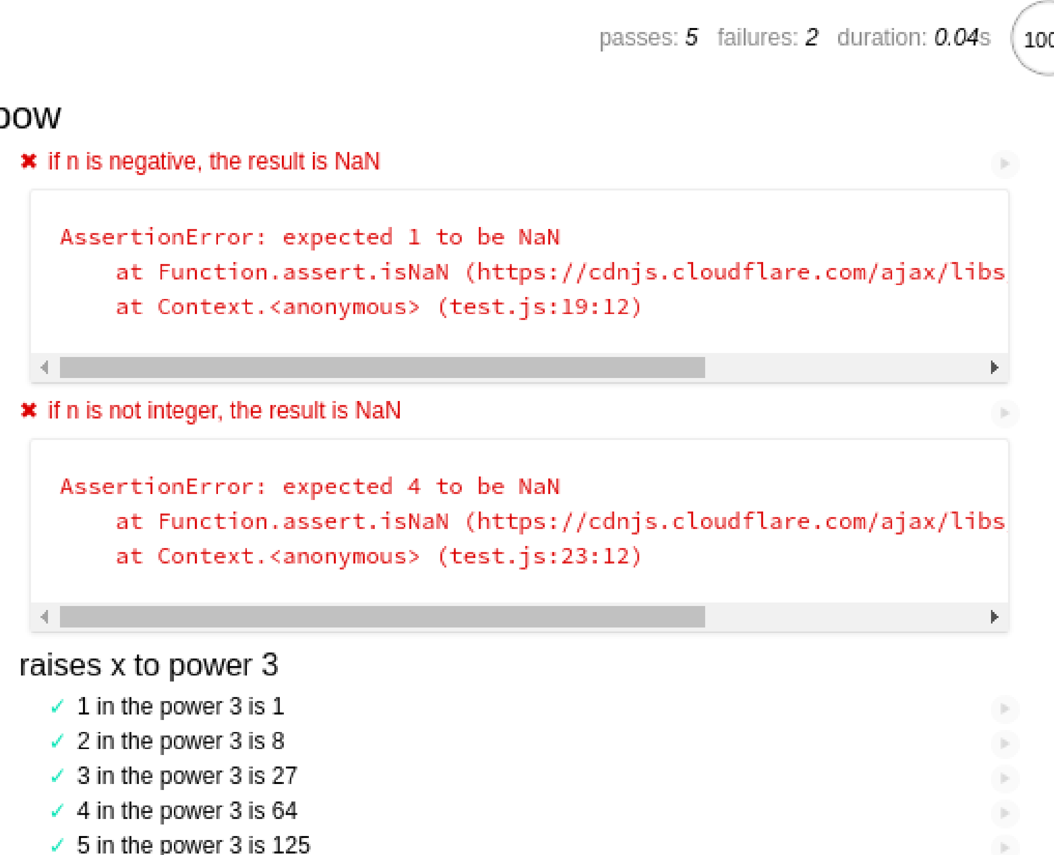
كما ذكرنا مسبقا، فإن الدالة `pow(x, n)` ستتعامل مع أرقام موجبة فقط `n`. تُرجع دوال JavaScript دائما `NaN` عند وجود خطأ حسابي. لنقم بهذا الأمر مع قيم `n`.

أولا، نضيف هذا إلى الوصف:

```
describe("pow", function() {

```

```
  // ...

  it("for negative n the result is NaN", function() {
    assert.isNaN(pow(2, -1));
  });

  it("for non-integer n the result is NaN", function() {
    assert.isNaN(pow(2, 1.5));
  });

});
```

النتائج مع الاختبارات الجديدة (تجربة حية):



تفشل الاختبارات المُضافة مؤخرا وذلك لأن التنفيذ لا يدعمها. هكذا هي الطريقة التي يعمل بها BDD: نبدأ بكتابة الاختبارات التي نعلم بأنها ستفشل ثم نكتب التنفيذ الخاص بها.

دوال تأكيد أخرى

لاحظ أن الدالة `assert.isNaN`: تفحص وجود القيمة NaN.

يوجد المزيد من دوال التأكيد في Chai، مثلا:

- `assert.equal(value1, value2)`: تفحص التساوي `value1 == value2`.
- `assert.strictEqual(value1, value2)`: تفحص التساوي التام `value1 === value2`.
- `assert.notEqual`، `assert.notStrictEqual`: تفحص عكس الدالتين أعلاه.
- `assert.isTrue(value)`: تفحص أن `value === true`.
- `assert.isFalse(value)`: تفحص أن `value === false`.

يمكنك قراءة باقي الدوال في التوثيق.

لذا، يجب أن نضيف بعض الأسطر للدالة `pow`:

```
function pow(x, n) {
  if (n < 0) return NaN;
  if (Math.round(n) != n) return NaN;

  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

الآن أصبحت تعمل وتَجتاز جميع الاختبارات (تجربة حية):

passes: 7 failures: 0 duration: 0.02s 100%

pow

✓ if n is negative, the result is NaN

✓ if n is not integer, the result is NaN

raises x to power 3

✓ 1 in the power 3 is 1

✓ 2 in the power 3 is 8

✓ 3 in the power 3 is 27

✓ 4 in the power 3 is 64

✓ 5 in the power 3 is 125

3.5.11 الخلاصة

يكون الوصف في البداية في BDD، ثم يأتي التنفيذ. فنحصل في الأخير على كل من الوصف والشيفرة البرمجية.

يمكن استخدام الوصف بثلاث طرق:

1. اختبارات - تضمن صحة عمل الشيفرة البرمجية.
2. توثيق - توضح العناوين في `describe` و`it` وظيفة الدالة.
3. أمثلة - تُعد الاختبارات أمثلة فعالة تعرض كيف يمكن استخدام الدالة.

يمكننا تطوير، تغيير، أو إعادة كتابة دالة من الصفر من خلال الوصف مع ضمان صحة عملها. يعد هذا الأمر مهمًا خاصة في المشاريع الكبيرة عند استخدام دالة في عدة أماكن. فعند تغيير هذه الدالة، يكون من الصعب التحقق من صحة عملها يدويًا في كل مكان.

يوجد خيارين بدون اختبارات:

1. تنفيذ التغيير بغض النظر عن النتائج. هكذا قد نواجه الكثير من الأخطاء عند الفحص اليدوي.
2. يتجنب المطورون تحديث الشيفرة عند توقع حدوث أخطاء فادحة وعدم وجود اختبارات لفحص هذه الأخطاء فتبقى الشيفرة البرمجية بدون تحديث.

يساعد الفحص الآلي على تجنب هذه المشاكل!

إن كان المشروع مليئا بالاختبارات، فلن يكون هناك أي مشكلة إطلاقا. فبعد أي تغيير يمكننا تنفيذ الاختبارات لنرى العديد من التحقيقات أجريت في غضون ثوانٍ.

علاوة على ذلك، الشيفرة البرمجية المختبرة جيدا تكون بهيكل أفضل.

منطقيا، لأن الشيفرة البرمجية المختبرة سهلة التعديل والتطوير. مع وجود سبب آخر أيضا. يجب أن تكون الشيفرة البرمجية منظمة من أجل كتابة اختبار بحيث يكون لدى كل دالة وظيفة رئيسية موصوفة، ومدخلات ومخرجات معروفة جيدا. ذلك يعني 1 هيكلة جيدة منذ البداية.

لا يكون الأمر بهذه السهولة في الواقع. ففي بعض الأحيان يكون من الصعب كتابة وصف للدالة قبل شيفرتها البرمجية لأن سلوكها لا يكون واضحًا بعد. لكن عموما، كتابة الاختبارات يجعل التطوير أسرع وأكثر استقرارًا.

ستواجه الكثير من المهام التي تتضمن ذلك لاحقا في الشرح. فسترى المزيد من الأمثلة العملية. يتطلب كتابة الاختبارات معرفة جيدة بِجافاسكربت. لكننا في بداية تعلمها. فلترتيب كل شيء، لست مطالبًا بكتابة الاختبارات في الوقت الحالي، لكن يجب أن تكون قادرًا على قرائتها حتى إن كانت معقدة أكثر قليلا مما تم شرحه هنا.

3.5.12 تمارين

ا. ما الخطأ في الاختبار التالي؟

الأهمية: ★★★★★

ما الخطأ في الاختبار للدالة pow أدناه؟

```
it("Raises x to the power n", function() {
  let x = 5;

  let result = x;
  assert.equal(pow(x, 1), result);

  result *= x;
  assert.equal(pow(x, 2), result);

  result *= x;
  assert.equal(pow(x, 3), result);
});
```

الحل:

يوضح الاختبار أحد الإغراءات التي يواجهها المطور أثناء كتابة اختبار. ما لدينا الآن هو ثلاثة اختبارات، لكنها مرتبة كدالة واحدة تتضمن ثلاثة تأكيدات.

تكون هذه الطريقة أسهل في بعض الأحيان، لكن إن وُجِد أي خطأ، فلن يكون من السهل معرفة سبب ذلك الخطأ. إن حدث خطأ ما أثناء تنفيذ شيفرة معقدة، فَسنحتاج لمعرفة قيم البيانات عند تلك النقطة. أي أننا سنحتاج لتنقيح الاختبار.

من الأفضل تجزئة الاختبار إلى أجزاء `it` متعددة مع مدخلات ومخرجات محددة بوضوح كما يلي:

```
describe("Raises x to power n", function() {
  it("5 in the power of 1 equals 5", function() {
    assert.equal(pow(5, 1), 5);
  });

  it("5 in the power of 2 equals 25", function() {
    assert.equal(pow(5, 2), 25);
  });

  it("5 in the power of 3 equals 125", function() {
    assert.equal(pow(5, 3), 125);
  });
});
```

بدلنا `it` مكان `describe` وبعدد من `it`. إن فشل شيء ما الآن فَسَنتمكن من رؤية البيانات. يمكننا أيضًا عزل اختبار واحد وتشغيلة في وضع مستقل باستخدام `it.only` بدلًا من `it`:

```
describe("Raises x to power n", function() {
  it("5 in the power of 1 equals 5", function() {
    assert.equal(pow(5, 1), 5);
  });

  // Mocha will run only this block
  it.only("5 in the power of 2 equals 25", function() {
    assert.equal(pow(5, 2), 25);
  });

  it("5 in the power of 3 equals 125", function() {
    assert.equal(pow(5, 3), 125);
  });
});
```

3.6 تعويض نقص دعم المتصفحات

تتطور لغة JavaScript باستمرار نتيجةً لوجود العديد من المقترحات المبنية على الحاجة إلى تطويرها دوريًا. تُحلَّل هذه المقترحات وتُضاف إن كانت جيدة إلى القائمة https://tc39.github.io/ecma262/ ثم النظر في وصفها.

لدى الفِرَق العاملة على محركات جافاسكربت أفكارها الخاصة حول ما يجب تنفيذه أولًا. قد تقرر هذه الفِرَق تنفيذ مقترح ما زال مسودة وتأجيل أشياء أصبح وصفها جاهزا لأنَّها قد تكون أقل أهمية أو صعبة التنفيذ؛ لذلك، نجد أن هذه المحركات تتضمن المعايير المتعارفة فقط.

لترى الميزات التي تدعمها اللغة، انظر في هذه الصفحة kangax.github.io/compat-table/es6 (يوجد الكثير من الميزات، فما زال لدينا الكثير لنتعلمه).

3.6.1 Babel

تفشل بعض المحركات في دعم بعض الشيفرات البرمجية عند استخدام ميزات اللغة الحديثة. فكما ذكرنا سابقًا، ليست جميع الميزات مُضَمَّنة في كل مكان. هنا يأتي دور Babel لإصلاح الوضع.

يُعد Babel مُفَسِّرًا تحويليًا (transpiler)، إذ يعيد كتابة شيفرة JavaScript حديثة بإصدار سابق.

في الواقع، يوجد جزأين في Babel:

1. **برنامج المُفَسِّر التحويلي (transpiler)** يعيد كتابة الشيفرة ويُشَغله المطورون على أجهزتهم. يعيد البرنامج كتابة الشيفرة البرمجية كتابةً متوافقةً مع إصدار سابق للغة ثم يتم توصيل الشيفرة الناتجة إلى الموقع لاستخدامها. توفر مشاريع بناء الأنظمة الحديثة مثل webpack الوسائل المناسبة لتشغيل المُفَسِّر التحويلي (transpiler) تلقائيًا مع كل تغيير للشيفرة البرمجية ما يجعل الاندماج في عملية التطوير أسهل.
2. **برنامج لتعويض نقص دعم المتصفحات (Polyfills)** قد تتضمن ميزات اللغة الجديدة بعض الوظائف المدمجة وهياكل الجُمَل. يعيد المُفَسِّر التحويلي كتابة الشيفرة البرمجية مُحولًا هياكل الجمل إلى إصدارات أقدم، لكن يجب تضمين الوظائف المُدمجة الجديدة. تُعد JavaScript لغة ديناميكية للغاية، وقد تضيف/تعدل السكربتات أي دالة حتى تتعامل وفقا للمعايير الحديثة.

السكربت الذي يضيف/يحدث دالة جديدة يسمى "polyfill"، أي معوِّض نقص الدعم لتلك الدالة أو الميزة الحديثة فهو يغطي الفجوة ويضيف المحتوى المفقود.

برامج تعويض نقص دعم المتصفحات التي قد تثير اهتمامك:

- core js تدعم الكثير وتسمح بتضمين الميزات المُرادة فقط.

- polyfill.io هي خدمــة تــزود الســكربت بوســائل تعــويض نقص دعم المتصــفحات وفقًــا للمــيزات والمتصفح.

لذلك، إن احتجت استخدام ميزات اللغة الحديثة، فمن المهم اسـتخدام مُفَسِّـر تحـويلي وبرنـامج دعم نقص المتصفحات.

3.6.2 أمثلة من الشرح

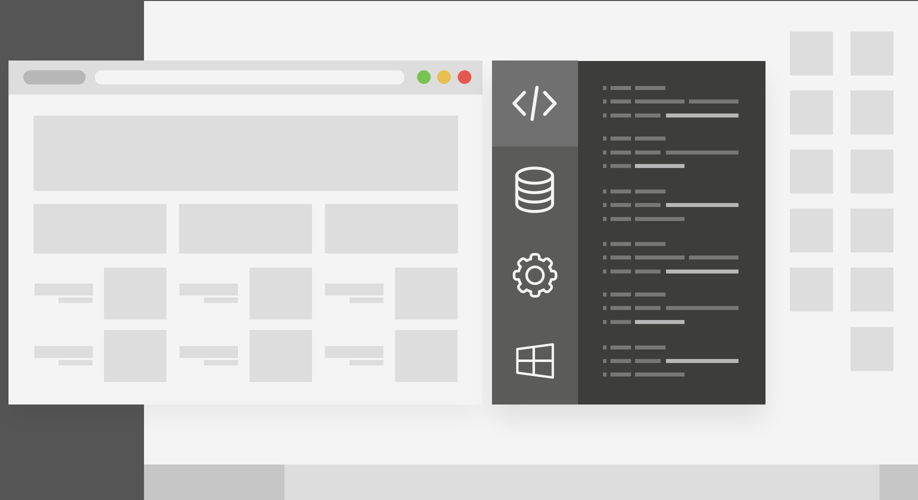
معظم الأمثلة قابلة للتشغيل في مكانها مثل:

```
alert('Press the "Play" button in the upper-right corner to run');
```

لن تعمل الأمثلة التي تستخدم JavaScript حديثة إلا في المتصفحات التي تدعمها.

يُعد متصفح جوجل كروم الأكثر حداثة دوما مع ميزات اللغة الحديثـة، من الجيـد تشـغيل العـروض الحديثـة بدون أي مُفَسِّر تحويلي. تعمل العديد من المتصفحات الأخرى المُحدَّثة جيدًا.

دورة علوم الحاسوب



دورة تدريبية متكاملة تضعك على بوابة الاحتراف
في تعلم أساسيات البرمجة وعلوم الحاسوب

التحق بالدورة الآن

4. الكائنات: تأسيس المفاهيم

يتضمن هذا الفصل الأقسام التالية:

1. الكائناتَ Objects
2. نسخ الكائن: الفرق بين القيمة والمرجع
3. كنس البيانات المهملة
4. الدول في الكائنات واستعمالها this
5. الباني والعامل new
6. التسلسل الاختياري ?.
7. النوع الرمزي Symbol
8. التحويل بين نوع كائن إلى نوع أولي

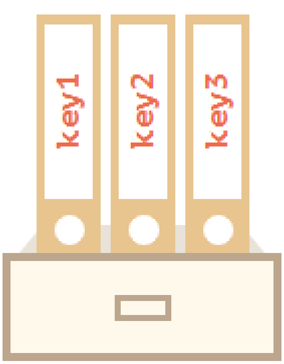
4.1 الكائنات Objects

يوجد سبعة أنواع للبيانات في JavaScript كما رأينا في فصل أنواع البيانات. ستة من هذه الأنواع تُدعى "أساسية" (primitive) لأنها تحوي قيمة شيء واحد فقط (سواء كان نصًا أو رقمًا أو أي شيء آخر).

في المُقابل، تُستخدم الكائنات لتخزين مجموعة من البيانات المتنوعة والوحدات المعقدة المُرَمَّزة بمفاتيح. تُضمَّن الكائنات في ما يقارب جميع نواحي JavaScript، لذا يجب علينا أن نفهمها قبل التعمق في أي شيء آخر.

يمكن إنشاء أي كائن باستخدام الأقواس المعقوصة `{…}` مع قائمة اختيارية بالخاصيات. الخاصية هي زوج من "مفتاح: قيمة" (`key: value`) إذ يكون المفتاح عبارة عن نص (يُدعى "اسم الخاصية")، والقيمة يمكن أن تكون أي شيء.

يمكننا تخيل الكائن كخزانة تحوي ملفات. يُخزن كل جزء من هذه البيانات في الملف الخاص به باستخدام المفتاح. يمكن إيجاد، أو إضافة، أو حذف ملف باستخدام اسمه.



يمكن إنشاء كائن فارغ (خزانة فارغة) باستخدام إحدى الصيغتين التاليتين:

```
let user = new Object(); // (object constructor) صياغة باني كائن
let user = {};  // (object literal) صياغة مختصرة لكائن عبر الأقواس
```

تُستخدم الأقواس المعقوصة `{...}` عادة، وهذا النوع من التصريح يُسمى "الصياغة المختصرة لتعريف كائن" (object literal).

4.1.1 القيم المُجرَّدة والخاصيات

يمكننا إضافة بعض الخاصيات (properties) إلى الكائن المعرَّف بـالأقواس `{...}` مباشـرة بشـكل أزواج "مفتاح: قيمة":

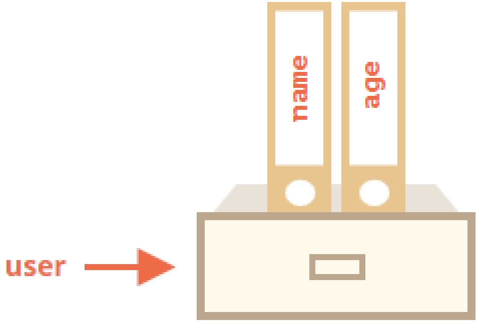
```
let user = {     // كائن
  name: "Ahmad",  // name عبر المفتاح Ahmad خزِّن القيمة
  age: 30         // age عبر المفتاح 30 خزِّن القيمة
};
```

لدى كل خاصية مفتاح (يُدعى أيضًا "اسم " أو "مُعَرِّف") قبـل النقطـتين " `:` " وقيمـة لهـذه الخاصـية بعـد النقطتين.

يوجد خاصيتين في الكائن `user`:

1. اسم الخاصية الأولى هو `"name"` وقيمتها هي `"Ahmad"`.
2. اسم الخاصية الثانية هو `"age"` وقيمتها هي `"30"`.

يمكن تخيل الكائن السابق `user` كخزانة بملفين مُسَمَّيان "name" و "age".

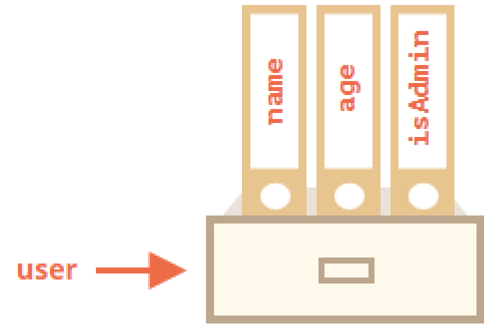


يمكننـا إضـافة، وحـذف، وقـراءة الملفـات من الخزانـة في أي وقت. يمكن الوصـول إلى قيم الخاصـيات باستخدام الصيغة النُقَطية (dot notation):

```
// الحصول على قيم خاصيات الكائن:
alert( user.name ); // Ahmad
alert( user.age ); // 30
```

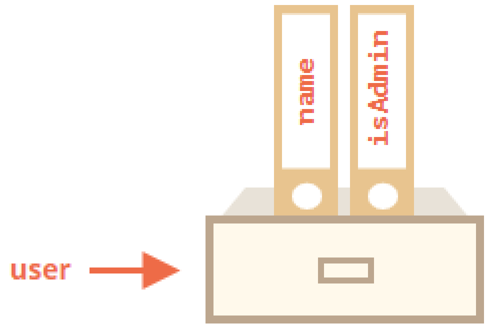
يمكن للقيمة أن تكون من أي نوع، لِنُضِف قيمة من نوع بيانات منطقية (boolean):

```
user.isAdmin = true;
```



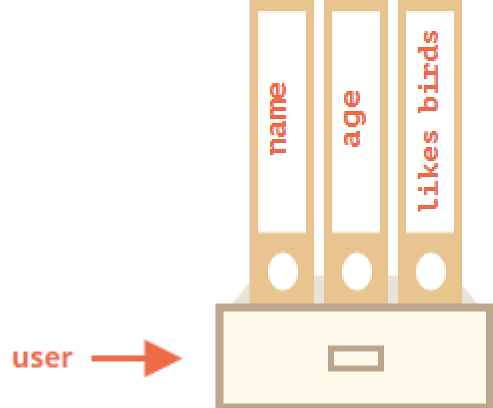
يمكننا استخدام المُعامِل `delete` لحذف خاصية:

```
delete user.age;
```



يمكننا أيضا استخدام خاصيات بأسماء تحوي أكثر من كلمة، لكن يجب وضعها بين علامات الاقتباس "":

```
let user = {
  name: "Ahmad",
  age: 30,
  "likes birds": true  // يجب أن تكون الخاصية ذات الاسم المُحتوي على أكثر من كلمة بين علامتي اقتباس
};
```



يمكن إضافة فاصلة بعد آخر خاصية في القائمة:

```
let user = {
  name: "Ahmad",
  age: 30,
}
```

وتُسمى بفاصلة "مُعَلِّقَة" أو "زائدة" فهي تجعل إضافة أو حذف أو التنقل بين الخاصيات أسهل لأن جميع الأسطر تُصبِح متشابهة.

4.1.2 الأقواس المعقوفة

لا تعمل طريقة الوصول إلى الخاصيات ذات الأسماء المحتوية على أكثر من كلمة باستخدام الصيغة النُقَطية:

```
// تعرض هذه التعليمة وجود خطأ في الصياغة
user.likes birds = true
```

ذلك لأن الصيغة النُقطية تحتاج لاسم متغير صحيح. لا يحوي مسافات أو حدود أخرى. يوجد بديل يعمل مع أي نص "صيغة الأقواس المعقوفة" `[]`:

```
let user = {};
// تخزين
user["likes birds"] = true;
// استرجاع
alert(user["likes birds"]); // true
// حذف
delete user["likes birds"];
```

يعمل كل شيء وفق المطلوب الآن. يُرجى ملاحظة أنَّ النص بداخل الأقواس مُحاط بعلامتي اقتباس (تعمل علامات الاقتباس الأخرى بطريقة صحيحة أيضًا).

تتيح لنا الأقواس المعقوفة أيضًا جلب اسم خاصية ناتجة عن قيمة أي تعبير - بدلًا من اسم الخاصية الفعلي - مثل استعمال اسم من متغير كما يلي:

```
let key = "likes birds";

// user["likes birds"] = true; يماثل قول
user[key] = true;
```

يمكن حساب قيمة المتغير key أثناء التنفيذ أو قد تعتمد قيمته على مدخلات المستخدمين ثم نستخدمه للوصول إلى الخاصية مما يعطي مرونة كبيرة في التعامل. إليك المثال التالي:

```
let user = {
  name: "Ahmad",
  age: 30
};

let key = prompt("What do you want to know about the user?", "name");

// الوصول باستخدام متغير
alert( user[key] ); // Ahmad (if enter "name")
```

لا يمكن استخدام الصيغة النُقطية بالطريقة نفسها:

```
let user = {
  name: "Ahmad",
  age: 30
};

let key = "name";
alert( user.key ) // غير معروف
```

ا. الخاصيات المحسوبة

يمكن استخدام الأقواس المعقوفة في كائن معرَّف بالأقواس، وهذا ما يسمى بالخاصيات المحسوبة. إليك المثال التالي:

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {
  [fruit]: 5, // fruit يؤخذ اسم الخاصية من المتغير
};
alert( bag.apple ); // fruit="apple" قيمتها 5 إذا كانت
```

معنى الخاصية المحسوبة سهل: تعني `[fruit]` أنَّ اسم الخاصية يجب أن يُؤخذ من `fruit`؛ لذا، إن أدخل الزائر `"apple"`، ستصبح قيمة `bag` هي `{apple: 5}`.

يعمل الأمر السابق بالطريقة التالية ذاتها:

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {};

// fruit خُذ اسم الخاصية من المتغير
bag[fruit] = 5;
```

لكن شكله يبدو أفضل، أليس كذلك؟! يمكن استخدام تعابير أكثر تعقيدًا داخل الأقواس المعقوفة:

```
let fruit = 'apple';
let bag = {
  [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

الأقواس المعقوفة أكثر قوة من الصيغة النُقطية. فهي تسمح باستخدام أي اسم للخاصيات والمتغيرات. لكنها أكثر تعقيدا في الكتابة. لذا، إن كانت أسماء المتغيرات سهلة ومعروفة، تُستخدَم الصيغة النُقَطية غالبًا. وإن أردنا بعض التعقيد، نستخدم إلى الأقواس المعقوفة.

يمكن استخدام الأسماء المحجوزة مع أسماء الخاصيات

لا يمكن لمتغير أن يحمل اسم إحدى الكلمات المحجوزة في اللغة مثل "for"، أو "let"، أو "return"، …الخ. لكن لا تُطبَّق هذه القاعدة على أسماء خاصيات الكائنات. فيمكن استخدام أي اسم معها:

```
let obj = {
  for: 1,
  let: 2,
  return: 3
};
alert( obj.for + obj.let + obj.return );  // 6
```

عمومًا، يمكن استخدام أي اسم، لكن هناك استثناء: "__proto__" لهذا الاسم معاملة خاصة لأسباب تاريخية. مثلًا، لا يمكننا استخدام الاسم على أنَّه قيمة لغير كائن:

```
let obj = {};
obj.__proto__ = 5;
alert(obj.__proto__); // [object Object] لا تعمل وفق المطلوب
```

كما نرى في الشيفرة، تم تجاهل تخزين القيمة الأولية 5.

تخزين أزواج الخاصيات التحكمية في كائن وإتاحة تحديد مفاتيح هذه الأزواج للزائر قد يجعل الشيفرة مصدرًا للأخطاء ومليئًا بالثغرات.

في تلك الحالة، قد يختار الزائر - الخبير والماكر - الاسم __proto__ ليكون مفتاحًا ويخرِّب البنية المنطقية للشيفرة (كما في المثال أعلاه). يوجد طريقة لجعل الكائنات تتعامل مع __proto__ بِعدِّها خاصية عادية وسنتطرق لها لاحقًا بعد فهم الكائنات بشكل أعمق.

يوجد أيضا هيكل بيانات آخر يدعى Map، والذي ستتعلمه في القسم الذي يتحدث عن نوعي البيانات Map و Set، اللذين يدعمان استعمال أي نوع مع المفاتيح.

4.1.3 اختزال قيم الخاصيات

نستخدم غالبا قيم متغيرات موجودة مسبقًا لتكون قيمًا لأسماء الخاصيات في الشيفرات الحقيقية. اطلع مثلًا على الشيفرة التالية:

```
function makeUser(name, age) {
  return {
    name: name,
    age: age
    // ... خاصيات أخرى
  };
}

let user = makeUser("Ahmad", 30);
alert(user.name); // Ahmad
```

تحمل الخاصيات في المثال السابق نفس أسماء المتغيرات. يُعدُّ إنشاء خاصية من متغير موجود مسبقًا حالة استخدام شائعة. أي أنه يوجد اختزال خاص لقيمة الخاصية لاختصار الشيفرة. بدلا من كتابة name:name، يمكننا كتابة name فقط، كما يلي:

```
function makeUser(name, age) {
  return {
    name, // name: name يماثل كتابة
    age   // age: age يماثل كتابة
    // ...
  };
}
```

يمكننا استخدام كلًا من الخاصيات الاعتيادية والاختزال في الكائن ذاته:

```
let user = {
  name,  // name:name يماثل
  age: 30
};
```

4.1.4 فحص الكينونة

قابلية الوصول إلى أي خاصية في الكائن هي إحدى مميزات الكائنات، ولكن ألَا يوجد أي خطأ في حال لم تكن الخاصية موجودة؟! عند محاولة الوصول إلى خاصية غير موجودة، تُرجَع القيمة `undefined`. مما يُعطي طريقة متعارفة لفحص كينونة (وجود) خاصية ما من عدمه بموازنتها مع القيمة `"undefined"` ببساطة:

```
let user = {};

alert( user.noSuchProperty === undefined ); // تحقق هذه الموازنة يشير إلى عدم وجود الخاصية
```

يوجد أيضا مُعامل خاص `"in"` لفحص تواجد أي خاصية. طريقة استخدام هذا المعامل كالتالي:

```
"key" in object
```

مثلا:

```
let user = { name: "Ahmad", age: 30 };

alert( "age" in user ); // true, user.age موجود
alert( "blabla" in user ); // false, user.blabla غير موجود
```

لاحظ أنه من الضروري وجود اسم الخاصية على يسار `in`. ويكون عادة نصًا بين علامتي اقتباس. إن لم نستخدم علامتي الاقتباس فهذا يعني فحص متغير يحمل الاسم ذاته مثل:

```
let user = { age: 30 };

let key = "age";
alert( key in user ); // key تطبع القيمة true إذ تؤخذ قيمة المتغير
                      // user ويُتحقق من وجود خاصية بذلك الاسم في الكائن
```

ا. استخدام "in" مع الخاصيات التي تُخزن القيمة undefined

تفحص عملية الموازنة الصارمة "=== undefined" غالبًا وجود الخاصية وفق المطلوب. لكن يوجد حالة خاصة تفشل فيها هذه العملية، بينما لا يفشل المعامل in إن استعمل مكانها. هذه الحالة هي عند وجود الخاصية في الكائن لكنها تُخزن القيمة undefined:

```
let obj = {
  test: undefined
};

alert( obj.test ); // يطبع القيمة undefined ولكن هل تُعدُّ الخاصية موجودة أم لا؟

alert( "test" in obj ); // تُطبع القيمة true وتُعدُّ الخاصية موجودة في الكائن
```

الخاصية obj.test موجودة فعليًا في الشيفرة أعلاه، لذا يعمل المُعامل in بصحة.

تحدث مثل هذه الحالات نادرًا فقط لأن القيمة undefined لا تُستخدَم بكثرة. نستخدم غالبا القيمة null للقيم الفارغة أو الغير معرفة، لذلك يُعد المُعامل in قليل الاستخدام في الشيفرات.

4.1.5 الحلقة for...in

يوجد شكل خاص للحلقة for..in للمرور خلال جميع مفاتيح كائنٍ ما. هذه الحلقة مختلفة تمامًا عما درسناه سابقًا، أي الحلقة for(;;).

صياغة الحلقة تكون بالشكل التالي:

```
for (key in object) {
  // يتنفذ ما بداخل الحلقة لكل مفتاح ضمن خاصيات الكائن
}
```

مثلا، لنطبع جميع خاصيات الكائن user:

```
let user = {
  name: "Ahmad",
```

```
  age: 30,
  isAdmin: true
};

for (let key in user) {
  // المفاتيح
  alert( key );  // name, age, isAdmin
  // قيم المفاتيح
  alert( user[key] ); // Ahmad, 30, true
}
```

لاحظ أن جميع تراكيب "for" تتيح لنا تعريف متغير التكرار بداخل الحلقة، مثل `let key` في المثال السابق. يمكننا أيضًا استخدام اسم متغير آخر بدلًا من `key`. إليك مثال يُستخدم بكثرة:

```
for (let prop in obj)
```

ا. الترتيب في الكائنات

هل الكائنات مرتبة؟ بمعنى آخر، إن تنقلنا في حلقة خلال كائن، هل نحصل على جميع الخاصيات بنفس الترتيب الذي أُضيفت به؟ وهل يمكننا الاعتماد على هذا؟

الإجابة باختصار هي: "مرتب بطريقة خاصة": الخاصيات الرقمية يُعاد ترتيبها، تظهر باقي الخاصيات بترتيب الإنشاء ذاته كما في التفاصيل التالية.

لنرَ مثالًا لكائن بِرموز الهاتف:

```
let codes = {
  "49": "Germany",
  "41": "Switzerland",
  "44": "Great Britain",
  // ..,
  "1": "USA"
};

for (let code in codes) {
  alert(code); // 1, 41, 44, 49
}
```

قد تُستخدم الشيفرة لاقتراح قائمة من الخيارات للمستخدم. إن كنا نبني موقعًا لزوار من ألمانيا فقد نريد أن تظهر 49 أولا. لكن، عند تشغيل الشيفرة، نرى شيئا مختلفًا تماما:

- تظهر USA (1) أولًا
- ثم Switzerland (41) وهكذا.

تُستخدم رموز الهواتف بترتيب تصاعدي لأنها أعدادٌ، لذا نرى 1، 41، 44، 49.

خاصيات عددية؟ ما هذا؟

تعني "الخاصية العددية" (integer property) نصًا يمكن تحويله من وإلى عدد دون أن يتغير. لذا فإن 49 هو اسم خاصية عددي لأنه عند تحويله إلى عدد وإرجاعه لنص يبقى كما هو. لكن "1.2" و "+49" ليست كذلك:

```
// هي دالة تحذف الجزء العشري Math.trunc
alert( String(Math.trunc(Number("49"))) );
// "49", الخاصية العددية ذاتها
alert( String(Math.trunc(Number("+49"))) );
// "49" مختلفة عن "+49" <= إذًا ليست خاصية عددية
alert( String(Math.trunc(Number("1.2"))) );
// "1" مختلفة عن "1.2" <= إذًا ليست خاصية عددية
```

في المقابل، إن كانت المفاتيح غير عددية، فتُعرَض بالترتيب الذي أُنشِئت به. إليك مثال على ذلك:

```
let user = {
  name: "Ahmad",
  surname: "Smith"
};
user.age = 25; // add one more

// تُعرض الخاصيات الغير رقمية بترتيب الإنشاء
for (let prop in user) {
  alert( prop ); // name, surname, age
}
```

لذا، لحل مشكلة رموز الهواتف يمكننا التحايل وجعلها غير عددية بإضافة "+" قبل كل رمز كما يلي:

```
let codes = {
  "+49": "Germany",
  "+41": "Switzerland",
  "+44": "Great Britain",
```

```
  // ..,
  "+1": "USA"
};

for (let code in codes) {
  alert( +code ); // 49, 41, 44, 1
}
```

الآن تعمل وفق المطلوب!

4.1.6 الخلاصة

الكائنات عبارة عن مصفوفات ترابطية بميزات خاصة عديدة. تُخزن الكائنات خاصيات (أزواج مفتاح-قيمة)، بشرط أنه:

- يجب أن تكون مفاتيح الخاصيات نصوصًا أو رموزًا (غالبًا نصوص).
- يمكن أن تكون القيم من أي نوع.

يمكننا استخدام ما يلي للوصول إلى خاصية:

- الصيغة النُقَطِيَّة: `obj.property`.
- صيغة الأقواس المعقوفة `obj["property"]`. تتيح لنا الأقواس المعقوفة أخذ مفتاح من متغير، مثل `obj[varWithKey]`.

عمليات أخرى:

- لِحذف خاصية: `delete obj.prop`.
- لِفحص تواجد خاصية بمفتاح معين: `"key" in obj`.
- للتنقل خلال كائن: الحلقة `for (let key in obj)`.

يُسمى ما درسناه في هذا الفصل "كائن بسيط" أو كائن فقط. يوجد العديد من أنواع الكائنات الأخرى في JavaScript:

- الكائن `Array` (مصفوفة): لتخزين مجموعة البيانات المرتبة،
- الكائن `Date` (تاريخ): لتخزين معلومات عن الوقت والتاريخ،
- الكائن `Error` (خطأ): لتخزين معلومات عن خطأ ما.

- وغيرها من أنواع الكائنات.

لدى هذه الأنواع ميزاتها الخاصة التي سيتم دراستها لاحقًا. يقول بعض الأشخاص أحيانًا شيئًا مثل "نوع مصفوفة" أو "نوع تاريخ" (الاسم الذي وضعته بين قوسين بجانب نوع الكائن)، لكن هذه الأنواع ليست أنواعًا مستقلة بحد ذاتها، إنما تنتمي إلى نوع البيانات `Object` (كائن) وتتفرع عنه بأشكال مختلفة.

تُعد الكائنات في JavaScript قوية جدًا. درسنا في هذا الفصل جزءًا بسيطًا من موضوع هائل جدًا. سنتعامل مع الكائنات لاحقًا بصورة أقرب وسَنتعلم أكثر عنها في فصول أخرى.

4.1.7 تمارين

ا. مرحبًا، بالكائن

الأهمية: ★★★★★

اكتب الشيفرة البرمجية، سطر لكل متطلب:

1. أنشئ كائنًا فارغًا باسم `user`.
2. أضف الخاصية `name` بالقيمة `Ahmad`.
3. أضف الخاصية `surname` بالقيمة `Smith`.
4. غير قيمة الخاصية `name` إلى `Pete`.
5. احذف الخاصية `name` من الكائن.

الحل:

```
let user = {};
user.name = "Ahmad";
user.surname = "Smith";
user.name = "Pete";
delete user.name;
```

ب. التحقق من الفراغ

اكتب الدالة `isEmpty(obj)` التي تُرجع القيمة `true` إن كان الكائن فارغًا، وتُرجِع القيمة `false` في الحالات الأخرى. يجب أن تعمل كالتالي:

```
let schedule = {};
alert( isEmpty(schedule) ); // true
```

```
schedule["8:30"] = "get up";
alert( isEmpty(schedule) ); // false
```

إليك تجربة حية للمثال.

الحل:

قم بالمرور خلال الكائن ونفذ الأمر `return false` مباشرة إن عثرت على أي خاصية:

```
function isEmpty(obj) {
  for (let key in obj) {
    // إن بدأت الحلقة بالعمل، فهناك خاصية في الكائن
    return false;
  }
  return true;
}
```

ج. جمع خاصيات الكائن

لدينا كائن يُخزن رواتب الفريق:

```
let salaries = {
  Ahmad: 100,
  Ann: 160,
  Pete: 130
}
```

اكتب الشيفرة التي تجمع الرواتب وتُخزنها في المتغير `sum`. يجب أن يكون مجموع المثال أعلاه 390. إن كان `salaries` فارغًا، فإن الناتج سيكون 0.

الحل:

```
let salaries = {
  Ahmad: 100,
  Ann: 160,
  Pete: 130
};
let sum = 0;
for (let key in salaries) {
  sum += salaries[key];
```

```
}
alert(sum); // 390
```

د. ضرب الخاصيات العددية بالقيمة 2

الأهمية: ★★★☆☆

أنشئ دالةً باسم `multiplyNumeric(obj)` تضرب جميع الخاصيات العددية في الكائن `obj` في العدد 2. مثلا:

```
// قبل الاستدعاء
let menu = {
  width: 200,
  height: 300,
  title: "My menu"
};
multiplyNumeric(menu);
// بعد الاستدعاء
menu = {
  width: 400,
  height: 600,
  title: "My menu"
};
```

لاحظ أنَّ الدالة `multiplyNumeric` لا يجب أن تُرجِع أي شيء. يجب أن تُعدِّل القيم بداخل الكائن.

> استخدام `typeof` لفحص الأعداد.

إليك تجربة حية للتمرين.

الحل:

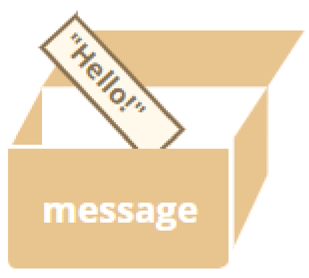
```
function multiplyNumeric(obj) {
  for (let key in obj) {
    if (typeof obj[key] == 'number') {
      obj[key] *= 2;
    }
  }
}
```

4.2 نسخ كائن: الفرق بين القيمة والمرجع

أحد الاختلافات الأساسية بين الكائنات (objects) وأنواع البيانات الأولية أو الأساسية (primitives) هي أنَّ الكائنات تُخزن وتُنسخ "بالمرجع" (by reference) بينما تُخزن/تُنسخ البيانات الأولية: النصوص، والأرقام، والقيم المنطقية بِعدِّها قيمةً كاملةً. إليك المثال التوضيحي التالي:

```
let message = "Hello!";
let phrase = message;
```

لدينا في هذه الشيفرة متغيرين مستقلين كلاهما يُخزِن النص "Hello!".

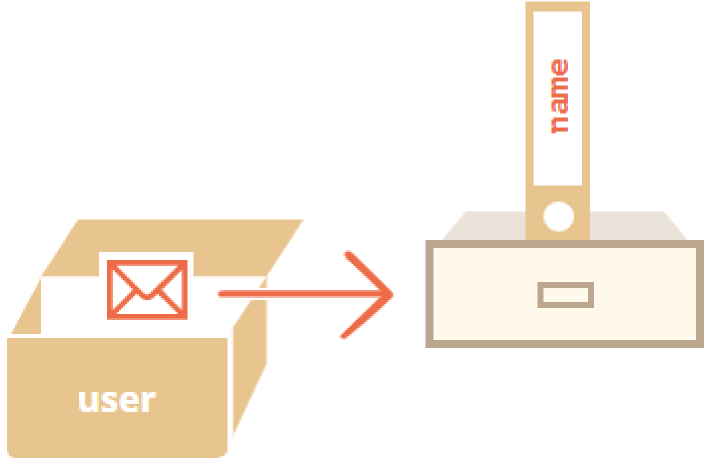


أما الكائنات ليست كذلك. الكائنات ليست كذلك.

لا يُخزِّن المتغير الكائن نفسه، وإنما "عنوانه في الذاكرة". بمعنى آخر، "مرجع للكائن".

هنا صورة للكائن:

```
let user = {
  name: "Ahmad"
};
```

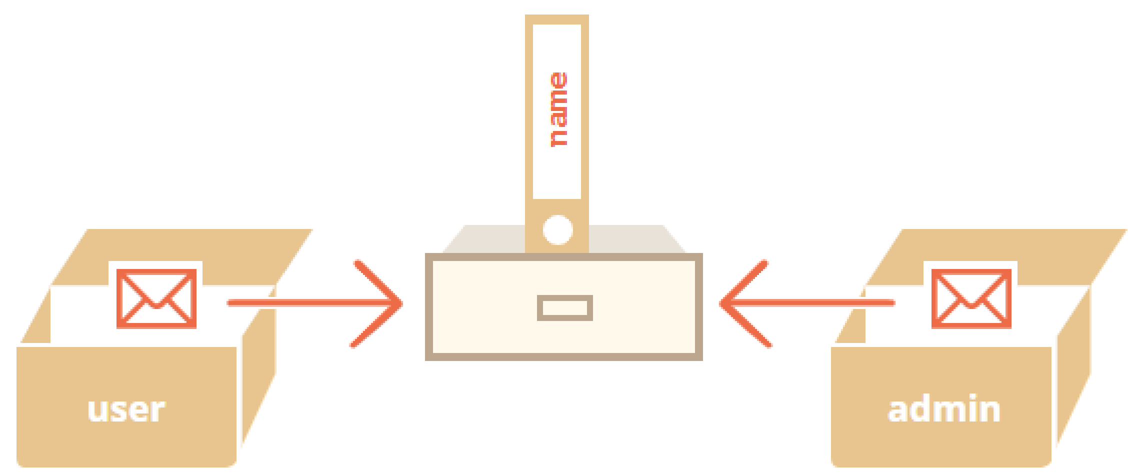


كما نرى، يُخزَّن المتغير في مكان ما في الذاكرة ويُخزِّن المتغير `user` مرجعًا إليه.

عند نسخ متغير من نوع كائن، يُنسَخ المرجع الذي يشير إلى ذلك الكائن ولا يُنسَخ الكائن نفسه ويُكرَّر. إذا تخيلنا الكائن كخزانة، فإن المتغير هو مفتاح الخزانة ونسخ المتغير يكرِّر المفتاح وليس الخزانة ذاتها. إليك مثال يوضح ما ذكرناه:

```
let user = { name: "Ahmad" };

let admin = user; // نسخ المرجع
```

الآن، أصبح لدينا متغيرين، كلاهما يحمل مرجعًا للكائن ذاته:



يمكننا استخدام كلا المتغيرين (المفتاحين) للوصول إلى الخزانة وتعديل محتواها:

```
let user = { name: 'Ahmad' };
```

```
let admin = user;

admin.name = 'Pete'; // غيرت القيمة من خلال المتغير admIn المرجعي

alert(user.name); // 'Pete' ظهر التغيير على المتغير "user" المرجعي
```

يوضح المثال أعلاه وجود كائن واحد فقط. كما لو كان لدينا خزانة بمفتاحين، واستخدمنا أحدهما (`admin`) للوصول إلى الخزانة. ثم إذا استخدمنا الآخر (`user`) لاحقًا، فسنرى انعكاس التغييرات التي أجراها الأول.

4.2.1 الموازنة بحسب المرجع

يعمل مُعاملي المساواة `==` والمساواة الصارمة `===` بنفس الطريقة للكائنات. يكون الكائنان متساويان إذا كانًا الكائن نفسه فقط. أي، إذا كان متغيران يشيران للكائن ذاته، فهما متساويان:

```
let a = {};
let b = a; // نسخ المرجع

alert( a == b ); // true, كِلا المتغيّرين يشيران إلى الكائن نفسه
alert( a === b ); // true
```

وهنا متغيران مستقلان ليسا متساويين أي يشيران إلى كائنين منفصلين حتى وإن كانا متماثلين تمامًا:

```
let a = {};
let b = {}; // كائنان منفصلان

alert( a == b ); // false
```

يُحوَّل الكائن إلى قيمة أولية (أساسية) في الموازنات مثل `obj1 > obj2` أو `obj == 5`. سندرس كيفية تحويل الكائنات قريبًا، لكن، في الحقيقة، مثل هذه الموازنات تكون نادرة الضرورة وتنتج غالبًا من خطأ في كتابة الشيفرة.

ا. الكائنات الثابتة

يمكن تغيير الكائن المُعَرَّف على أنَّه ثابت (أي وسم المتغير الذي يحوي الكائن بالكلمة المفتاحية `const`) دون حصول أي أخطاء. إليك الشيفرة التالية مثلًا:

```
const user = {
  name: "Ahmad"
};
```

```
user.age = 25; // (*)
alert(user.age); // 25
```

قد يبدو أن السطر (`*`) سيسبب خطأ، لكن لا يوجد أي مشكلة البتة. ذلك لأنَّ `const` تحافظ على قيمة `user` ذاتها وتمنع تغييرها. ويُحزِّن `user` المرجع للكائن نفسه دائمًا ولا يتغيِّر بتعديل الكائن نفسه. يدخل السطر (`*`) إلى الكائن ولا يعيد تعيين قيمة المتغير `user`.

يمكن أن يعطي `const` خطأ إن حاولنا تغيير قيمة المتغير `user` مثل:

```
const user = {
  name: "Ahmad"
};

// (user خطأ (لا يمكن تغيير قيمة المتغير
user = {
  name: "Pete"
};
```

لكن، ماذا إن أردنا إنشاء خاصيات ثابتة ضمن الكائن؟ سيُعطي `user.age = 25` آنذاك خطأ، وذلك ممكن أيضًا. سيتم شرحه في فصل "رايات الخاصيات وواصفاتها".

4.2.2 الاستنساخ والدمج

إذًا، نسخ كائن يُنشِئ مرجعًا إضافيًّا له. لكن ماذا إن كنا نريد تكرار كائن فعليًّا أي إنشاء نسخة مستقلة منه؟ ذلك ممكن أيضًا، لكنه أصعب قليلًا لعدم وجود دالة تقوم بذلك في JavaScript ذلك لأنَّه لا حاجة لذلك بكثرة. النسخ بالمرجع كافٍ غالبًا.

لكن إن أردنا ذلك فعلًا، فإننا نحتاج لإنشاء كائن وتكرار هيكل الكائن الموجود عبر التنقل خلال خاصياته ونسخها على المستوى الأولي. وإليك مثال على ذلك:

```
let user = {
  name: "Ahmad",
  age: 30
};

let clone = {}; // الكائن الجديد الفارغ

// نسخ جميع خاصيات المتغير user إليه
```

```
for (let key in user) {
  clone[key] = user[key];
}

// الآن أصبحت النسخة مستقلة تماما
clone.name = "Pete"; // تغيير البيانات في النسخة

alert( user.name ); // تظل Ahmad في الكائن الأصلي
```

يمكننا استخدام الدالة Object.assign للغرض ذاته.

الصياغة الدالة هي:

```
Object.assign(dest, [src1, src2, src3...])
```

- تُعد المُعاملات `dest`، و `src1`، وحتى `srcN` كائنات (يمكن أن تكون بالعدد المُراد).
- تنسخ الدالة خاصيات جميع الكائنات `src1, ..., srcN` إلى الكائن `dest`. بمعنى آخر، تُنسخ جميع الخاصيات لجميع المُعاملات بدءًا من المُعامل الثاني إلى المُعامل الأول. ثم يتم إرجاع `dest`.

مثلا، يمكننا استخدام الدالة لدمج عدة كائنات إلى كائن واحد:

```
let user = { name: "Ahmad" };

let permissions1 = { canView: true };
let permissions2 = { canEdit: true };

// user إلى permissions2 و permissions1 تنسخ جميع الخاصيات من
Object.assign(user, permissions1, permissions2);
// now user = { name: "Ahmad", canView: true, canEdit: true }
```

إن كان الكائن `user` يحوي أحد أسماء الخاصيات مسبقًا، فسيتم إعادة كتابة محتواها:

```
let user = { name: "Ahmad" };
// isAdmin وإضافة name إعادة كتابة
Object.assign(user, { name: "Pete", isAdmin: true });
// now user = { name: "Pete", isAdmin: true }
```

يمكننا أيضًا استخدام الدالة `Object.assign` بدلًا من الحلقة للاستنساخ البسيط:

```
let user = {
  name: "Ahmad",
  age: 30
};

let clone = Object.assign({}, user);
```

تنسخ الدالة جميع خاصيات الكائن user إلى الكائن الفارغ وتُرجِعه كما في الحلقة لكن بشكل أقصر.

4.2.3 الاستنساخ المتداخل

حتى الآن، عدَدْنا جميع خاصيات user أولية (أساسية)، لكن قد تكون بعض الخاصيات مرجعًا لكائن آخر مثل الشيفرة التالية فما العمل؟

```
let user = {
  name: "Ahmad",
  sizes: {
    height: 182,
    width: 50
  }
};

alert( user.sizes.height ); // 182
```

في هذه الحالة نسخ user.sizes = clone.sizes ليس كافيًا لأنَّ user.sizes عبارة عن كائن، فسيُنسَخ على أنَّه مرجعٌ. هكذا، سيصبح لدى clone و user الحجم ذاته:

```
let user = {
  name: "Ahmad",
  sizes: {
    height: 182,
    width: 50
  }
};

let clone = Object.assign({}, user);
alert( user.sizes === clone.sizes ); // صحيح، الكائن ذاته
```

```
// sizes يتشارك الكائنان user و clone الكائن الفرعي
user.sizes.width++;       // تغيير خاصية من مكان ما
alert(clone.sizes.width); // 51, يعرض النتيجة من مكان آخر
```

لإصلاح هـذا، يجب اسـتخدام حلقـة الاستنسـاخ الـتي تفحص كـل قيمـة في `user[key]`، وإن كـان كائنًا نستبدل الهيكل الخاص به أيضًا. هذه الطريقة تُسمى "استنساخ عميق" (deep cloning).

يوجد خوارزمية عامة للاستنسـاخ العميـق تنفِّذ الحالـة السـابقة بشـكل صـحيح، بالإضـافة إلى حـالات أكـثر تعقيدًا. تُدعى هذه الخوارزمية خوارزمية الاستنساخ المُهيكلة. حتى لا نُعيد اختراع العجلة مجدَّدًا، يمكننا استخدام تنفيذ جاهز للحالة من مكتبة lodash JavaScript، تُدعى الدالة _.cloneDeep(obj).

4.2.4 الخلاصة

تُخَزَّن الكائنات وتُنسخ باستخدام المرجع. بمعنًى آخر، لا يُخزن المتغير قيمـة الكـائن (object value) لكنـه يُخزن مرجعًا (reference) يمثِّل موقع قيمة الكائن في الذاكرة. لـذا فـإن نسـخ هـذا المتغـير أو تمريـره إلى دالـة سَينسخ هذا المرجع وليس الكائن ككُل. جميع العمليات (مثـل إضـافة أو حـذف خاصـيات) المُنفَّذة على مرجـع منسوخ تُنفَّذ على الكائن نفسه.

لعمل نسخة حقيقية (الاستنساخ) يمكننا استخدام `Object.assign` أو _.cloneDeep(obj).

4.2.5 تمارين

1. كائنات ثابتة؟

الأهمية: ★★★★★

هل من الممكن تغيير كائن صُرِّح عنه بالكلمة المفتاحية `const`؟ ما رأيك؟

```
const user = {
  name: "Ahmad"
};

// هل تعمل؟
user.name = "Pete";
```

الحل:

بالفعل ستعمل بدون مشاكل. تحمي الكلمة المفتاحية `const` المتغير نفسه من التغيير فقط. بمعنى آخر، يخزن `user` مرجعًا للكائن ولا يمكن تغييره مع وجود التصريح عنه بالكلمة المفتاحية `const` لكن يمكن تغيير محتوى الكائن.

```
const user = {
  name: "Ahmad"
};

// تعمل
user.name = "Pete";

// خطأ
user = 123;
```

4.3 كنس البيانات المهملة

تُدَار الذاكرة في JavaScript تلقائيًا في الخفاء. نحن ننشئ المتغيرات الأولية، والكائنات، والدوال وجميعها تأخذ مكانًا في الذاكرة ولكن هل سألت نفسك ماذا يحدث عندما يصبح أحد هذه الأشياء مهملًا وغير مهم؟ كيف يكتشف ذلك مُحرِّك JavaScript ويتخلص منه؟ هذا ما سنعرفه في هذا الفصل.

4.3.1 قابلية الوصول

المبدأ الرئيسي لإدارة الذاكرة في JavaScript هو قابلية الوصول reachability. ببساطة، القيم "القابلة للوصول" هي القيم التي يمكن الوصول إليها واستخدامها بطريقة ما وهذه القيم مخولة لتُخزَّن في الذاكرة.

أولًا، يوجد مجموعة من القيم القابلة للوصول بطبيعة الحال والتي لا يمكن التخلص منها لأسباب وجيهة مثل:

- المتغيرات المحلية والمُعاملات للدالة الحاليّة.
- المتغيرات والمُعامِلات لدوال أخرى ضمن السلسلة الحالية في الاستدعاء المُتداخل.
- المتغيرات العامة.
- (بالإضافة إلى بعض المتغيرات الأخرى الداخلية)

تُدعى هذه القيم "جذورًا" roots.

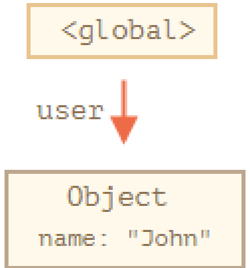
ثانيًا، تكون أي قيمة أخرى قابلة للوصول إن كان بالإمكان الوصول إليها من جذر باستخدام مرجع أو سلسلة من المراجع. مثلًا، إن كان هناك كائن في متغير محلي، ولدى هذا الكائن خاصية تشير لكائن آخر، فإنَّ هذا الكائن قابل للوصول. وهذه الكائنات التي يُشار إليها قابلة للوصول أيضًا. ستجد أمثلة توضيحية لاحقًا.

يوجد عملية خلفيَّة في محرك JavaScript تُدعَى "كانس المهملات" (garbage collector) تعمل على مراقبة الكائنات وحذف التي لم تَعُد قابلة للوصول.

4.3.2 مثال بسيط

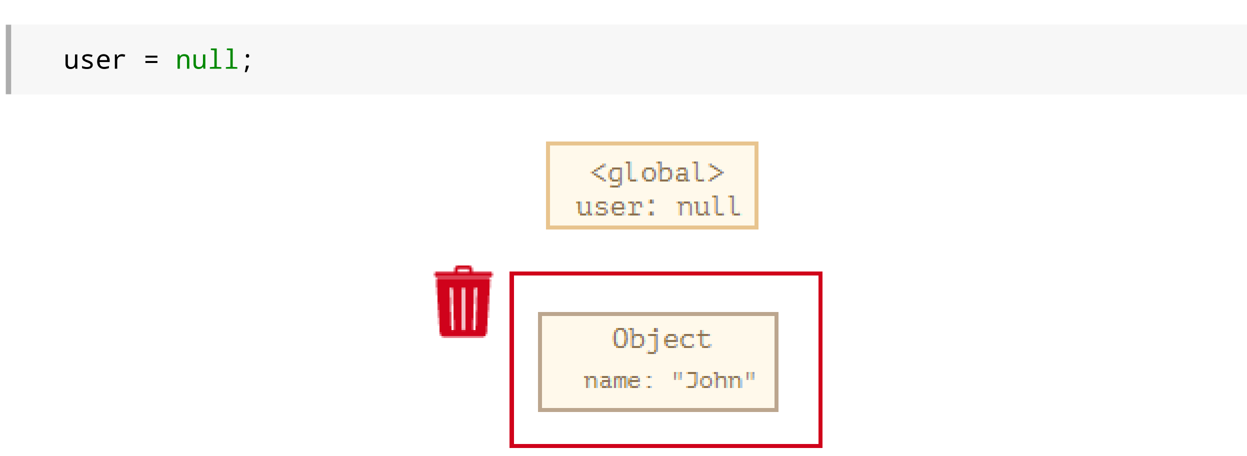
أبسط مثال هو:

```
// user يرجع لكائن
let user = {
  name: "Ahmad"
};
```



يُصوِّر السهم في الصورة مرجعًا لكائن. المتغير العام `"user"` يشير للكائن `{name: "Ahmad"}` (سنُسميه Ahmad اختصارًا). الخاصية `"name"` للكائن Ahmad تُخزن قيمة أولية، لذ رُسمَت بداخل الكائن. إن استُبدلَت قيمة المتغير `user`، فَسَنفقِد المرجع الذي يشير إلى الكائن Ahmad:

```
user = null;
```

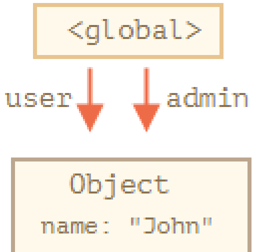


أصبح الكائن Ahmad غير قابل للوصول الآن أي لا يوجد طريقة للوصول إليه. سيعمل كانس المهملات على حذف البيانات المتمثلة في الكائن Ahmad وتحرير الذاكرة التي يحتلها.

4.3.3 مرجعان لكائن

لنفترض أننا نسخنا المرجع من المتغير `user` إلى متغير آخر باسم `admin`:

```
// user يرجِع لكائن
let user = {
  name: "Ahmad"
};

let admin = user;
```



إن قمنا بالأمر السابق ذاته الآن:

```
user = null;
```

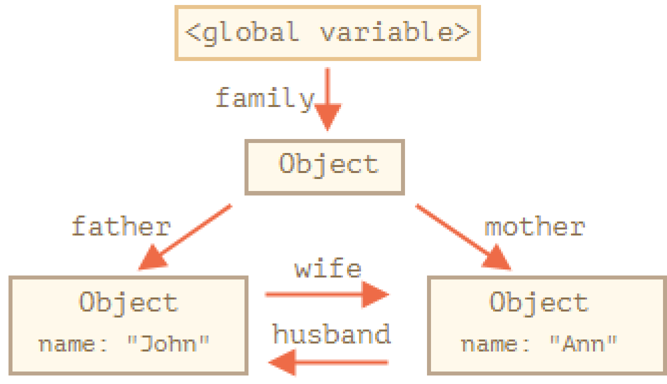
فسيكون الكائن قابلًا للوصول من خلال المتغير العام admin، لذا يبقى في الذاكرة. إن استبدلنا محتوى المتغير admin أيضًا فسيُحذَف الكائن.

4.3.4 الكائنات المتداخلة

الآن ننتقل لمثال أكثر تعقيدًا. ألق نظرة على الشيفرة التالية:

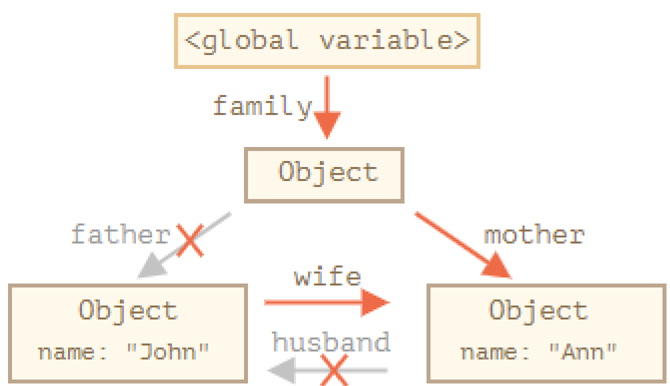
```
function marry(man, woman) {
  woman.husband = man;
  man.wife = woman;

  return {
    father: man,
    mother: woman
  }
}

let family = marry({
  name: "Ahmad"
}, {
  name: "Ann"
});
```

تربط الدالة marry كائنين بجعل كلاهما يشير إلى الآخر ثم ترجِع كائنًا جديدًا يحوي كلاهما. هيكل الذاكرة الناتج يكون كالتالي:

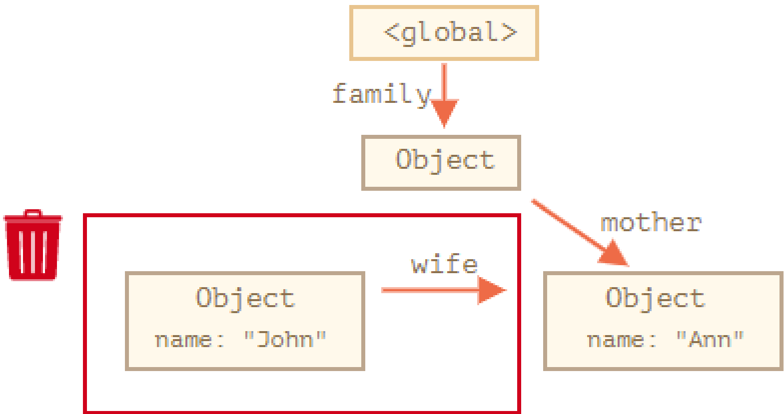


تكون جميع البيانات قابلة للوصول حتى الآن. دعنا نجرب حذف مرجعين الآن:

```
delete family.father;
delete family.mother.husband;
```

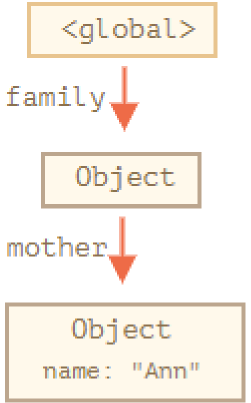


ليس من الكافِ حذف أحد المرجِعين فقط، لأنَّ جميـع الكائنـات سـتظل قابلـة للوصـول لكن إن حـذفنا كلا المرجعين، فسنرى عدم وجود أي مرجع إلى الكائن Ahmad:



لا يهم وجود مرجع من الكائن، إذ ما يجعله قابلًا للوصول هو المراجع التي تشير إليه، لذا فإنَّ الكائن Ahmad أصبح غير قابل للوصول وسيُحذَف من الذاكرة مع جميع بياناته التي أصبحت غير قابلة للوصول أيضًا.

بعد تجميع البيانات الغير مرغوب بها، يبقى لدينا:



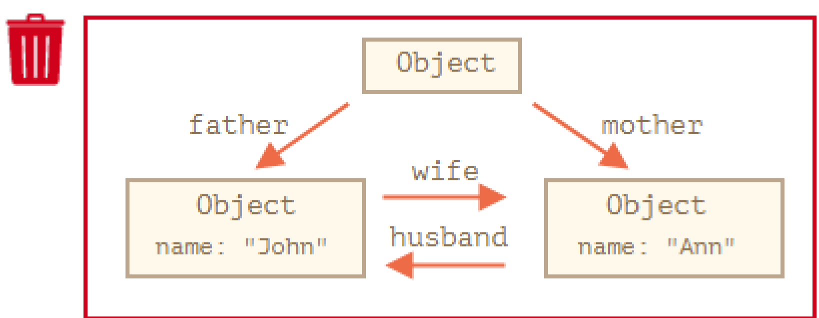
4.3.5 جزيرة غير قابلة للوصول

يمكن أن تصبح جزيرة من الكائنات المترابطة غير قابلة للوصول وتُحذَف من الذاكرة. الكائن الرئيسي هو الكائن أعلاه ذاته:

```
family = null;
```

تُصبح الصورة في الذاكرة كما يلي:

<global>
family: null



يوضح هذا المثال أهمية مبدأ قابلية الوصول. من الواضح أن الكائنين Ahmad و Ann ما زالا مرتبطين ولكل منهما مراجع لبعضهما، لكن ذلك غير كافٍ.

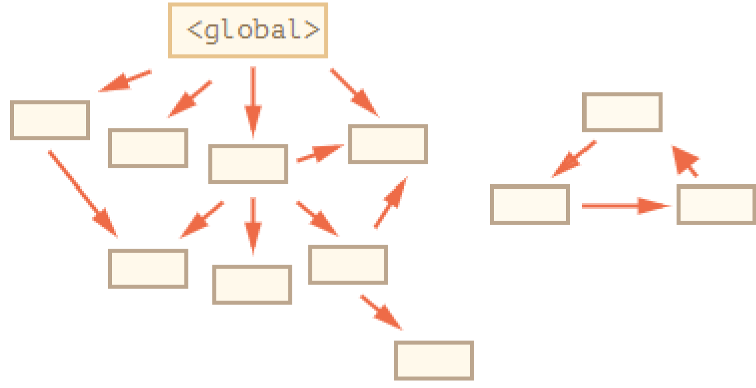
الكائن السابق `"family"` أصبح غير مربوط بالجذر، أي لم يعد هناك أي مرجع إليه لذا فإن الجزيرة كاملةً تصبح غير قابلة للوصول وتُحذَف.

4.3.6 الخوارزميات الداخلية

تُدعى الخوارزمية الأساسية لتجميع البيانات المهملة "الاستهداف والتمشيط" (mark-and-sweep)؛ تُنفَّذ خطوات جمع البيانات المهملة دوريًا وفق الخطوات التالية:

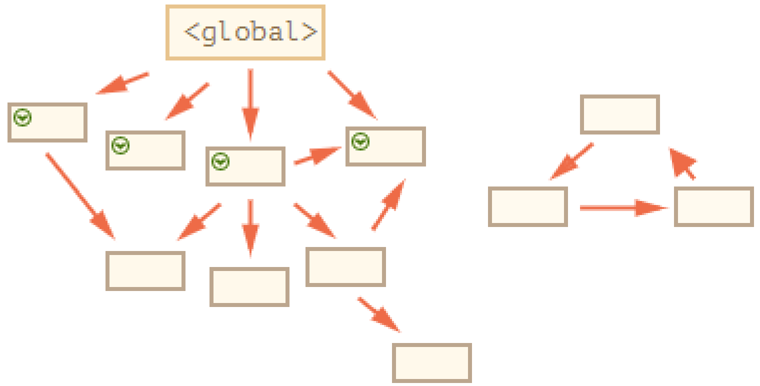
- يأخذ كانس المهملات الجذور ويحفظها (يُحدِّدها هدفًا له).
- ثم يُمشِّط جميع الإشارات الخارجة منها (مراجع لكائنات أخرى) ويحفظها لاستهدافها أيضًا.
- ثم يُمشِّط جميع الكائنات التي استهدفها مسبقًا ويحفظ مراجعها لاستهدفها لاحقًا. يُمشِّط جميع الكائنات بتلك الطريقة ويتذكرها لكي لا يُمشِّط أي كائن مرةً ثانية مستقبلًا.
- تستمر العملية مرارًا وتكرارًا حتى يصبح هناك مراجع لم تُمشَّط (غير قابلة للوصول من أي جذر).
- تُحذَف جميع الكائنات باستثناء تلك استُهدِفَت وعُلِّمَت بأنَّها غير مهملة.

مثلًا، ليكن هيكل الكائنات لدينا كما يلي:

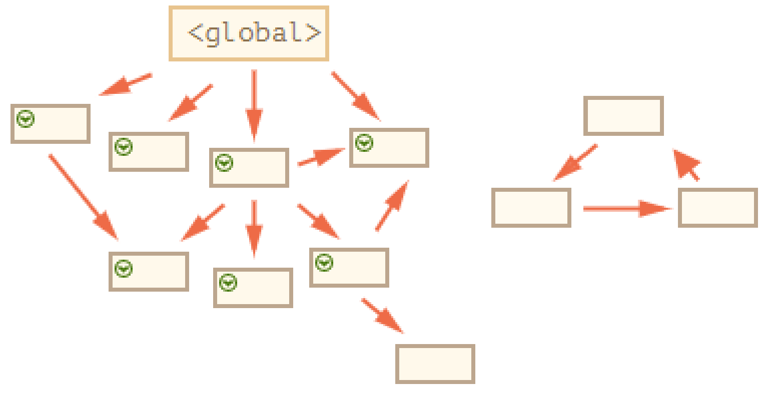


يمكن رؤية جزيـرة غـير قابلـة للوصـول في اليمين. الآن لـنرى كيـف يتعامـل معهـا كـانس المهملات وفـق خوارزمية الاستهداف والتمشيط.

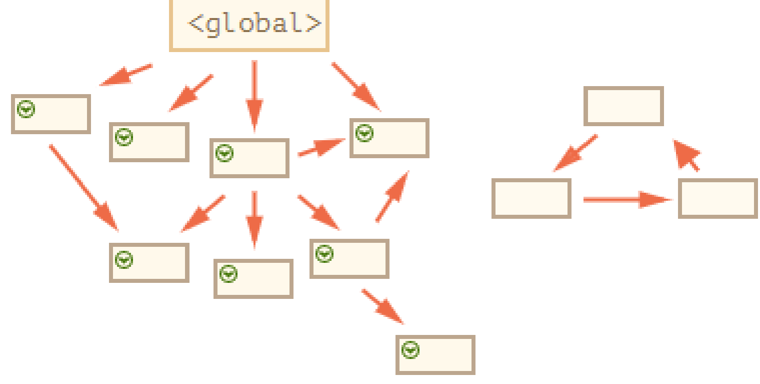
الخطوة الأولى هي تحديد الجذور:



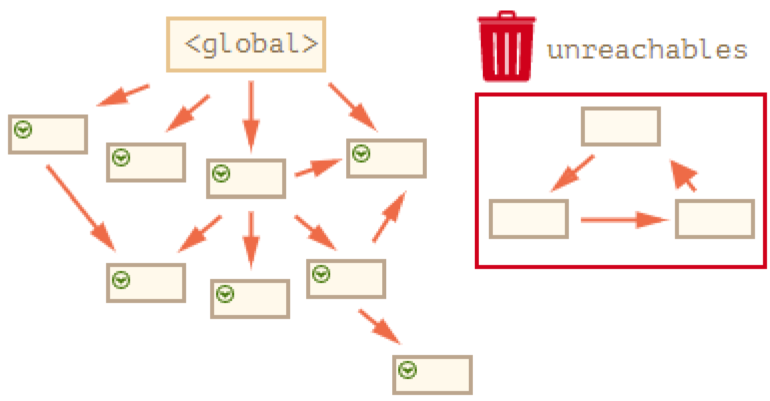
ثم تُحدَّد المراجع التي تشير إليها:



تُحدَّد المراجع التي تشير لها هذه الكائنات أيضًا:



تُعدُّ الكائنات التي لم تُمشَّط أثناء العملية غير قابلة للوصول ويجري حذفها:



هذه الآلية التي يعمل بها كانس المهملات.

تطبق محركات JavaScript العديد من التحسينات لتحسين هذه الخوارزمية وتسريع عملها بطريقة لا تؤثر على سرعة التنفيذ.

بعض التحسينات:

- التجميع وفق الجيل (Generational collection): تُقَسَّم الكائنات إلى مجموعتين: "كائنات جديدة" و "كائنات قديمة". تُنشَأ العديد من الكائنات ثمَّ تؤدي عملها ثمَّ تموت بسرعة، ويمكن تنظيفها بقوة. وتلك التي تنجو تصبح قديمة وتُفحَص بوتيرة أقل.
- التجميع التدريجي (Incremental collection): إن وُجِد العديد من الكائنات وأردنا المرور خلال جميع الكائنات وتحديدها دفعة واحدة، فقد يأخذ ذلك وقتًا ويظهِر آنذاك تأخيرٌ ملحوظٌ في التنفيذ، لذا يحاول المُحرِّك تقسيم عملية كنس البيانات المهملة إلى أجزاء ثم تنفيذ الأجزاء واحدًا تلو الآخر بشكل منفصل. قد يتطلب ذلك إجراء حسابات إضافية لتتبع التغييرات، لكن يصبح لدينا الكثير من التأخيرات الغير ملحوظة بدلًا من تأخير واحد كبير.

- التجميع وقت الخمول (Idle-time collection): يحاول كانس المهملات العمل عندما يكون المعالج غير مشغول لتقليل أي تأثيرات محتملة على التنفيذ.

يوجد تحسينات وإضافات أخرى على خوارزمية كنس المهملات لكن يجب على التوقف هنا لأنَّ المحركات المختلفة تُطبِّق تقنيات مختلفة. والأهم من ذلك، تتغير الأمور بتطور المحركات، لذا فإنَّ التعمق مسبقًا دون الحاجة لذلك لا يستحق العناء. إلا إن كان ذلك رغبة شخصية، فَسنضَع بعض الروابط في الأسفل.

4.3.7 الخلاصة

الأشياء التي يجب معرفتها:

- تُكنَس البيانات المهملة تلقائيًا، وهو أمر لا يمكن فرضه أو تجنبه.
- تبقى الكائنات في الذاكرة طالما يمكن الوصول إليها من أي جذر.
- وجود مرجعٍ للكائن ليس مثل أن يكون قابلًا للوصول من جذر، ويمكن لمجموعة من الكائنات المترابطة أن تصبح غير قابلة للوصول.

تستخدم المحركات الحديثة خوارزميات متطورة لكنس البيانات المهملة.

يغطي كتاب "The Garbage Collection Handbook: The Art of Automatic Memory Management" (لمؤلفه R. Jones وغيره) بعضًا منها.

إن كنت معتادًا على البرمجة بلغات ذات مستوى منخفض، يوجد معلومات مفصَّلة عن كانس المهملات في المُحرِّك V8 في الفصل رحلة إلى V8: كنس البيانات المهملة.

تنشر مدونة V8 أيضًا مقالات عن التغييرات في إدارة الذاكرة من وقت لآخر. لتتعلم عن كنس البيانات المهملة، يجب أن تتجهز بتعلم أمور V8 الداخلية كما يُفضَّل أن تقرأ مدونة Vyacheslav Egorov الذي عمل كأحد مهندسي V8. أنا أقول V8 لوجود الكثير من المقالات عنه على الإنترنت. العديد من الجوانب متشابهة بالنسبة لباقي المحركات، لكن يختلف كنس البيانات المهملة من عدة نواحي بينها.

المعرفة العميقة بالمحركات جيدة عندما تحتاج إلى إجراء تحسين منخفض المستوى. فكِّر في ذلك وليكن خطوتك التالية بعد أن تعتاد على اللغة.

4.4 الدوال في الكائنات واستعمالها this

تُنشَأ الكائنات عادة لتُمَثِّل أشياء من العالم الحقيقي مثل المستخدمين، والطلبات، وغيرها:

```
let user = {
  name: "Ahmad",
  age: 30
};
```

يمكن للمستخدم في العالم الحقيقي أن يقوم بِعدَّة تصرفات: مثل اختيار شيء من سلة التسوق، وتسجيل الدخول، والخروج ...إلخ. تُمَثَّل هذه التصرفات في لغة JavaScript بإسناد دالة إلى خاصية وتدعى الدالة آنذاك بالتابع (method، أي دالة [function] تابعة لكائن).

4.4.1 أمثلة على الدوال

بدايةً، لنجعل المستخدم `user` يقول مرحبًا:

```
let user = {
  name: "Ahmad",
  age: 30
};

user.sayHi = function() {
  alert("Hello!");
};

user.sayHi(); // Hello!
```

استخدمنا هنا تعبير الدالة لإنشاء دالة تابع للكائن `user` وربطناها بالخاصية `user.sayHi` ثم استدعينا الدالة. هكذا أصبح بإمكان المستخدم (أي الكائن user) التحدث! الآن أصبح لدى الكائن `user` الدالة `sayHi`.

يمكننا أيضًا استخدام دالة معرفة مسبقًا بدلًا من ذلك كما يلي:

```
let user = {
  // ...
};

// أولا، نعرف دالة
function sayHi() {
```

```
  alert("Hello!");
};

// أضِف الدالة للخاصية لإنشاء تابع
user.sayHi = sayHi;

user.sayHi(); // Hello!
```

البرمجة الكائنية Object-oriented programming

يسمى كتابة الشيفرة البرمجية باستخدام الكائنات للتعبير عن الأشياء "بالبرمجة الشيئة/الكائنية" (object-oriented programming، تُختَصَر إلى "OOP").

موضوع البرمجة الكائنية OOP كبير جدًا، فهو علم مشوق ومستقل بذاته. يعلمك كيف تختار الكائنات الصحيحة؟ كيف تنظم التفاعل فيما بينها؟ كما يعد علمًا للهيكلة ويوجد العديد من الكتب الأجنبية الجيدة عن هذا الموضوع مثل كتاب "Design Patterns: Elements of Reusable Object-Oriented Software" للمؤلفين E.Gamma، و R.Helm، و R.Ahmadson، و J.Vissides أو كتاب "Object-Oriented Analysis and Design with Applications" للمؤلف G.Booch، وغيرهما.

ا. اختصار الدالة

يوجد طريقة أقصر لكتابة الدوال في الكائنات المعرفة تعريفًا مختصـرًا باسـتعمال الأقـواس تكـون بالشـكل التالي:

```
// يتصرف الكائنان التاليان بالطريقة نفسها

user = {
  sayHi: function() {
    alert("Hello");
  }
};
// يبدو شكل الدالة المختصر أفضل، أليس كذلك؟
user = {
  sayHi() { // مثل "sayHi: function()"
    alert("Hello");
  }
};
```

يمكننا حذف الكلمة المفتاحية `"function"` وكتابة `sayHi()` كما هو موضح. حقيقةً، التعبيرين ليسا متطابقين تمامًا، يوجد اختلافات خفية متعلقة بالوراثة في الكائنات (سيتم شرحها لاحقًا)، لكن لا يوجد مشكلة الآن. يفضل استخدام الصياغة الأقصر في كل الحالات تقريبًا.

4.4.2 الكلمة المفتاحية "this" في الدوال

من المتعارف أن الدوال تحتاج للوصول إلى المعلومات المخزنة في الكائن لِتنفذ عملها. مثلًا، قد تحتاج الشيفرة التي بداخل `user.sayHi()` لِاسم المستخدم `user`. هنا، يمكن للدالة استخدام الكلمة المفتاحية `this` للوصول إلى نسخة الكائن التي استدعتها. أي، قيمة `this` هي الكائن "قبل النقطة" الذي استُخدِم لاستدعاء الدالة. مثلًا:

```
let user = {
  name: "Ahmad",
  age: 30,

  sayHi() {
    // هو الكائن الحالي "this"
    alert(this.name);
  }

};

user.sayHi(); // Ahmad
```

أثناء تنفيذ `user.sayHi()` هنا، ستكون قيمة `this` هي الكائن `user`.

عمليًا، يمكن الوصول إلى الكائن بدون استخدام `this` بالرجوع إليه باستخدام اسم المتغير الخارجي:

```
let user = {
  name: "Ahmad",
  age: 30,

  sayHi() {
    alert(user.name); // "user" بدلًا من "this"
  }

};
```

لكن، لا يمكن الاعتماد على الطريقة السابقة. فإذا قرَّرنا نسخ الكائن user إلى متغير آخر، مثل admin = user وغيرنا محتوى user لشيء آخر، فسيتم الدخول إلى الكائن الخطأ كما هو موضح في المثال التالي:

```
let user = {
  name: "Ahmad",
  age: 30,

  sayHi() {
    alert( user.name ); // يتسبب في خطأ
  }

};


let admin = user;
user = null; // تغيير المحتوى لتوضيح الأمر

admin.sayHi(); // استخدام الاسم القديم بِداخل sayHi() يُرجِع خطأ
```

إن استخدمنا this.name بدلًا من user.name بداخل alert، فستعمل الشيفرة عملًا صحيحًا.

4.4.3 "this" غير محدودة النطاق

تتصرَّف الكلمة المفتاحية this في JavaScript تصرُّفًا مختلفًا عن باقي اللغات البرمجية فيمكن استخدامها في أي دالة. انظر إلى المثل التالي، إذ لا يوجد خطأ في الصياغة:

```
function sayHi() {
  alert( this.name );
}
```

تُقَيَّم قيمة this أثناء تنفيذ الشيفرة بالاعتماد على السياق. مثلًا، في المثال التالي، تم تعيين الدالة ذاتها إلى كائنين مختلفين فيصبح لكل منهما قيمة مختلفة لـ "this" أثناء الاستدعاء:

```
let user = { name: "Ahmad" };
let admin = { name: "Admin" };
function sayHi() {
  alert( this.name );
}
```

```
// استخدام الدالة ذاتها مع كائنين مختلفين
user.f = sayHi;
admin.f = sayHi;

// لدى الاستدعاءان قيمة مختلفة لـ this
// "this" التي بداخل الدالة تعني المتغير الذي قبل النقطة
user.f(); // Ahmad  (this == user)
admin.f(); // Admin  (this == admin)

admin['f'](); // Admin (يمكن الوصول إلى الدالة عبر الصيغة النقطية أو الأقواس المربعة - لا يوجد مشكلة في ذلك)
```

القاعدة ببساطة: إذا استُدعِيَت الدالة `obj.f()`، فإن `this` هي `obj` أثناء استدعاء `f`؛ أي إما `user` أو `admin` في المثال السابق.

استدعاءٌ دون كائن: this == undefined

يمكننا استدعاء الدالة دون كائن:

```
function sayHi() {
  alert(this);
}

sayHi(); // undefined - غير معرَّف
```

في هذه الحالة ستكون قيمة `this` هي `undefined` في الوضع الصارم. فإن حاولنا الوصول إلى `this.name` سيكون هناك خطأ.

في الوضع غير الصارم، فإن قيمة `this` في هذه الحالة ستكون المتغير العام (في المتصفح `window` والتي سَنشرحها في فصل المتغيرات العامة). هذا السلوك قديمٌ وقد أصلحه الوضع الصارم `"use strict"`. أي يُعد هذا الاستدعاء خطأً برمجيًا غالبًا، فإن وُجِدت `this` بداخل دالة، فمن المتوقع استدعاؤها من خلال كائن.

الأمور المترتبة على this الغير محدودة النطاق

إن أتيت من لغة برمجية أخرى، فمن المتوقع أنك معتاد على "`this` المحدودة" إذ يمكن لِلدوال المعرَّفة في الكائن استخدام `this` التي ترجع للكائن.

تستخدم `this` بحرية في JavaScript، وتُقَيَّم قيمتها أثناء التنفيذ ولا تعتمد على المكان حيث عُرِّفت فيه، بل على الكائن الذي قبل النقطة التي استدعت الدالة.

> يوجد إيجابيات وسلبيات لمبدأ تقييم this أثناء وقت التشغيل. فمن ناحية، يمكن إعادة استخدام الدالة مع عدة كائنات، ومن الناحية الأخرى، المرونة الأكثر تعطي فرصًا أكثر للخطأ. لسنا بصدد الحكم على تصميم اللغة ونعته بالجيد أم سيء، بل نحاول فهم طريقة عملها وكيفية الاستفادة من ميزاتها وتجنب الأخطاء.

4.4.4 ميزة داخلية: النوع المرجعي

يُغطي هذا الجزء ميزة متقدمة - من ميزات اللغة - لفهم أفضل لحالة معينة. إن كنت على عجلة من أمرك، يمكنك تخطي أو تأجيل هذا الجزء.

يمكن لاستدعاء الدالة المعقد أن يُفقِد this، فمثلًا:

```
let user = {
  name: "Ahmad",
  hi() { alert(this.name); },
  bye() { alert("Bye"); }
};

user.hi(); // Ahmad (يعمل الاستدعاء البسيط)

// user.hi الآن، لنستدعي user.bye أو وفقًا للاسم
(user.name == "Ahmad" ? user.hi : user.bye)(); // خطأ!
```

يوجد معامل شرطي في السطر الأخير والذي يختار إما user.hi أو user.bye. في هذه الحالة يتم اختيار user.hi ثم يتم استدعاء الدالة مع الأقواس (). لكنها لا تعمل!

كما ترى، ينتج خطأ من الاستدعاء لأن قيمة "this" بداخل الاستدعاء أصبحت undefined.

ستعمل بهذه الطريقة (الكائن.الدالة):

```
user.hi();
```

هذه الصياغة لا تُعطي دالة:

```
(user.name == "Ahmad" ? user.hi : user.bye)(); // خطأ!
```

لماذا؟ إن أردنا فهم سبب حدوث ذلك، لِنكشف الغطاء عن كيفية عمل الاستدعاء obj.method(). عند النظر عن قرب، يمكننا ملاحظة عمليتين في التعليمة obj.method():

1. أولا، النقطة '.' تسترجع الخاصية obj.method
2. ثم الأقواس () تنفذ الدالة.

إذًا، كيف تُمرَّر المعلومات عن `this` من الجزء الأول للثاني؟ إن وضعنا العمليتين في سطرين منفصلين، فَسنفقد `this` بالتأكيد:

```
let user = {
  name: "Ahmad",
  hi() { alert(this.name); }
}

// فصل الحصول على الدالة واستدعائها في سطرين منفصلين
let hi = user.hi;
hi(); // خطأ، لأن this غير مُعَرَّفَة
```

تُسنِد التعليمة `hi = user.hi` الدالة إلى المتغير، ثم، في السطر الأخير تصبح مستقلة، فلا يوجد `this` هنا ضمن النطاق.

تستخدم JavaScript خدعة لجعل `user.hi()` تعمل، فصيغة النقطة `'.'` لا تُرجِع دالة، بل قيمة من النوع المرجعي الخاص.

النوع المرجعي هو "نوع للتخصيص". لا يمكننا استخدام هذا النوع بشكل واضح، بل يُستخدَم داخليًا بواسطة اللغة. تُشَكَّل قيمة النوع المرجعي من ثلاث قيم `(base, name, strict)`، إذ:

- `base` هي الكائن.
- `name` هو اسم الخاصية.
- `strict` تساوي "true" إن كان الوضع الصارم `use strict` مُفعلًا.

النتيجة من الوصول إلى خاصية `user.hi` ليست دالة، إنما قيمة من النوع المرجعي. بالنسبة لـ `user.hi` في الوضع الصارم تكون:

```
// قيمة من النوع المرجعي
(user, "hi", true)
```

عند استدعاء الأقواس `()` في النوع المرجعي فإنها تستقبل المعلومة كاملة عن الكائن والدالة، وتتمكن من تعيين `this` بطريقة صحيحة (`user` في هذه الحالة).

النوع المرجعي هو نوع "وسيط" داخلي، وغرضه هو تمرير المعلومات من الصيغة النُقطية `.` إلى أقواس الاستدعاء `()`.

أي عملية أخرى مثل الإسناد `hi = user.hi` تُلغي النوع المرجعي ككل، فهي تأخذ قيمة الدالة `user.hi` وتُمررها. فَتفقد العمليات التالية `this`. لذا، ونتيجة لذلك، تُمرَّر قيمة `this` بالطريقة الصحيحة إن كانت الدالة مُستدعاه مباشرة باستخدام صيغة النقطة `obj.method()` أو الأقواس المربعة `obj['method']()` (يؤديان العمل ذاته). سنتعلم طرائق أخرى لحل هذه المشكلة لاحقًا، مثل استخدام func.bind().

4.4.5 الدوال السهمية لا تحوي this

الدوال السهمية (Arrow function) هي دوال خاصة: فهي لا تملك `this` مُخصصَّة لها. إن وضعنا `this` في إحدى هذه الدوال فَستؤخَذ قيمة `this` من الدالة الخارجية.

مثلًا، تحصل الدالة `arrow()` على قيمة `this` من الدالة الخارجية `user.sayHi()`:

```
let user = {
  firstName: "Ilya",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};

user.sayHi(); // Ilya
```

يُعد ذلك إحدى ميزات دوال الدوال السهمية، وهي مفيدة عندما لا نريد استخدام `this` مستقلة، ونريد أخذها من السياق الخارجي بدلًا من ذلك. سَنتعمَّق في موضوع الدوال السهمية لاحقًا في فصل "إعادة النظر في الدوال السهمية".

4.4.6 الخلاصة

- الدوال المُخزَّنة في الكائنات تسمى "توابع" (methods).
- تسمح هذه الكائنات باستدعائها بالشكل `object.doSomething()`.
- يمكن للدوال الوصول إلى الكائن المعرفة فيه (أو النسخة التي استدعته المشتقة منه) باستخدام الكلمة المفتاحية`this`.
- تُعَرَّف قيمة `this` أثناء التنفيذ.
- قد نستخدم `this` عند تعريف دالة، لكنها لا تملك أي قيمة حتى استدعاء الدالة.

- يمكن نسخ دالة بين الكائنات.
- عند استدعاء دالة بالصيغة `object.method()`، فإن قيمة `this` أثناء الاستدعاء هي `object`.

لاحظ أن الدوال السهمية مختلفة تتعامل تعاملًا مختلفًا مع `this` إذ لا تملك قيمة لها. عند الوصول إلى `this` بداخل دالة سهمية فإن قيمتها تؤخذ من النطاق الموجودة فيه.

4.4.7 تمارين

ا. فحص الصياغة

الأهمية: ★★☆☆☆

ما نتيجة هذه الشيفرة؟

```
let user = {
  name: "Ahmad",
  go: function() { alert(this.name) }
}

(user.go)()
```

ملاحظة: يوجد فخ :)

الحل:

خطأ! جرب تشغيل الشيفرة:

```
let user = {
  name: "Ahmad",
  go: function() { alert(this.name) }
}

(user.go)() // !خطأ
```

لا تعطي مُعظم رسائل الخطأ في المتصفحات توضيح لسبب الخطأ.

سبب الخطأ هو فاصلة منقوطة مفقودة بعد `user = {...}`.

لا تضع JavaScript فاصلة منقوطة قبل القوس `(user.go)()`. لذا فإنها تقرأ الشيفرة كالتالي:

```
let user = { go:... }(user.go)()
```

يمكننا أيضًا رؤية أن هذا التعبير المتداخل هو استدعاء للكائن `{ go: ... }` كدالة بالمعامل `(user.go)`. ويحدث ذلك أيضًا في السطر نفسه مع `let user`، لذا فإن الكائن `user` لم يُعَرَّف بعد، وهكذا يظهر الخطأ.

إن وضعنا الفاصلة المنقوطة، سيصبح كل شيء صحيح:

```
let user = {
  name: "Ahmad",
  go: function() { alert(this.name) }
};

(user.go)() // Ahmad
```

لاحظ أن الأقواس حول `(user.go)` لا تعمل شيئًا هنا. فهي ترتب العمليات غالبًا، لكن النقطة لها الأولوية على أي حال. لذا فليس هناك أي تأثير. فقط الفاصلة المنقوطة هي الخطأ.

ب. شرح قيمة this

الأهمية: ★★★☆☆

استدعينا الدالة `user.go()` في الشيفرة التي بالأسفل 4 مرات متتالية. لكن الاستدعاءان `(1)` و `(2)` يعملان عملًا مختلفًا عن الاستدعاءين `(3)` و `(4)`. لماذا؟

```
let obj, method;

obj = {
  go: function() { alert(this); }
};

obj.go();               // (1) [object Object]

(obj.go)();             // (2) [object Object]

(method = obj.go)();    // (3) غير معرف

(obj.go || obj.stop)(); // (4) غير معرف
```

الحل:

هنا التوضيح:

أولًا، يُعد استدعاء دالة عادي.

ثانيًا، مثل 1 تمامًا، لا تغير الأقواس ترتيب العمليات هنا، تعمل النقطة أولًا على أي حال.

ثالثًا، هنا لدينا استدعاء أكثر تعقيدًا `(expression).method()`. يعمل الاستدعاء كما لو تم فصله إلى سطرين:

```
f = obj.go; // حساب التعبير
f();        // الاستدعاء
```

تُنَفَّذ `f()` هنا كدالة، دون `this`.

رابعًا، مشابه للنقطة `(3)`، لدينا تعبيرًا يسار النقطة `.`.

لشرح سلوك الاستدعاءين `(3)` و `(4)`، نحتاج لإعادة استدعاء معاملات الوصول لتلك الخاصية (النقطة أو الأقواس المربعة) التي ترجع قيمة من النوع المرجعي.

أي عملية عليها عدا استدعاء الدالة (مثل التعيين `=` أو `||`) تُرجِعُها إلى قيمة عادية، والتي لا تحمل المعلومات التي تسمح بتعيين `this`.

ج. استخدام this في الكائن معرَّف باختصار عبر الأقواس

الأهمية: ★★★★★

تُرجِع الدالة `makeUser` كائنًا هنا. ما النتيجة من الدخول إلى `ref` الخاص بها؟ ولماذا؟

```
function makeUser() {
  return {
    name: "Ahmad",
    ref: this
  };
};

let user = makeUser();

alert( user.ref.name ); // ما النتيجة؟
```

الحل:

الإجابة: ظهور خطأ. جربها:

```
function makeUser() {
  return {
    name: "Ahmad",
    ref: this
  };
};

let user = makeUser();

alert( user.ref.name ); // خطأ: لا يمكن قراءة الخاصية 'name' لِقيمة غير معرفة
```

ذلك لأن القواعد التي تعين `this` لا تنظر إلى تعريف الكائن. ما يهم هو وقت الاستدعاء. قيمة `this` هنا بداخل `makeUser()` هي `undefined`، لأنها استُدعيَت كدالة منفصلة، وليس كدالة بصياغة النقطة.

قيمة `this` هي واحدة للدالة ككل، ولا تؤثر عليها أجزاء الشيفرة ولا حتى الكائنات. لذا فإن `ref: this` تأخذ `this` الحالي للدالة.

هنا حالة معاكسة تمامًا:

```
function makeUser() {
  return {
    name: "Ahmad",
    ref() {
      return this;
    }
  };
};

let user = makeUser();

alert( user.ref().name ); // Ahmad
```

أصبحت تعمل هنا لأن `user.ref()` هي دالة، وقيمة `this` تعَيَّن للكائن الذي قبل النقطة `'.'`.

د. إنشاء آلة حاسبة

الأهمية: ★★★★★

أنشئ كائنًا باسم `calculator` يحوي الدوال الثلاث التالية:

- `read()` تطلب قيمتين وتحفظها كخصائص الكائن.
- `sum()` تُرجِع مجموع القيم المحفوظة.
- `mul()` تضرب القيم المحفوظة وتُرجِع النتيجة.

```
let calculator = {
  // ... ضع شيفرتك هنا ...
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

الحل:

```
let calculator = {
  sum() {
    return this.a + this.b;
  },

  mul() {
    return this.a * this.b;
  },

  read() {
    this.a = +prompt('a?', 0);
    this.b = +prompt('b?', 0);
  }
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

٥. التسلسل

الأهمية: ★★☆☆☆

لدينا الكائن `ladder` (سُلَّم) الذي يتيح الصعود والنزول:

```
let ladder = {
  step: 0,
  up() {
    this.step++;
  },
  down() {
    this.step--;
  },
  showStep: function() { // يعرض الخطوة الحالية
    alert( this.step );
  }
};
```

الآن، إن أردنا القيام بعدة استدعاءات متتالية، يمكننا القيام بما يلي:

```
ladder.up();
ladder.up();
ladder.down();
ladder.showStep(); // 1
```

عَدِّل الشيفرة الخاصة بالدوال `up`، و `down`، و `showStep` لجعل الاستدعاءات متسلسلة كما يلي:

```
ladder.up().up().down().showStep(); // 1
```

يُستخدَم هذا النمط بنطاق واسع في مكتبات JavaScript.

الحل:

الحل هو إرجاع الكائن نفسه من كل استدعاء.

```
let ladder = {
  step: 0,
  up() {
    this.step++;
    return this;
```

```
  },
  down() {
    this.step--;
    return this;
  },
  showStep() {
    alert( this.step );
    return this;
  }
}

ladder.up().up().down().up().down().showStep(); // 1
```

يمكننا أيضا كتابة استدعاء مستقل في كل سطر ليصبح سهل القراءة بالنسبة للسلاسل الأطول:

```
ladder
  .up()
  .up()
  .down()
  .up()
  .down()
  .showStep(); // 1
```

4.5 الباني والعامل new

نُنشِئ الكائنات باستخدام الصيغة الاعتيادية المختصرة `{...}`. لكننا نحتاج لإنشاء العديد من الكائنات المتشابهة غالبًا، مثل العديد من المستخدمين، أو عناصر لقائمة وهكذا. يمكن القيام بذلك باستخدام الدوال البانية (constructor functions) لكائن والعامِل `"new"`.

4.5.1 الدالة البانية

تقنيًا، الدوال البانية هي دوال عادية، لكن يوجد فكرتين متفق عليها:

1. أنها تبدأ بأحرف كبيرة.
2. يجب تنفيذها مع العامِل (operator) `"new"` فقط.

إليك المثال التالي:

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Jack");

alert(user.name); // Jack
alert(user.isAdmin); // false
```

عند تنفيذ دالة مع العامِل `new`، تُنَفَّذ الخطوات التالية:

1. يُنشَأ كائن فارغ ويُسنَد إلى `this`.
2. يُنَفَّذ محتوى الدالة. تقوم غالبًا بتعديل `this`، وإضافة خاصيات إليه.
3. تُرجَع قيمة `this`.

بمعنى آخر، تقوم `new User(...)` بشيء يشبه ما يلي:

```
function User(name) {
  // this = {};  (ضمنيًا)

  // إضافة خاصيات إلى this
  this.name = name;
```

```
    this.isAdmin = false;

    // return this;  (ضمنيًا)
  }
```

إذًا، تُعطي `let user = new User("Jack")` النتيجة التالية ذاتها:

```
let user = {
  name: "Jack",
  isAdmin: false
};
```

الآن، إن أردنا إنشاء مستخدمين آخرين، يمكننا استدعاء `new User("Ann")`، و `new User("Alice")` وهكذا. تعدُّ هذه الطريقة في بناء الكائنات أقصر من الطريقة الاعتيادية عبر الأقواس فقط، وأيضًا أسهل للقراءة. هذا هو الغرض الرئيسي للبانيات، وهي تطبيق شيفرة قابلة لإعادة الاستخدام لإنشاء الكائنات.

لاحظ أنَّه يمكن استخدام أي دالة لتكون دالة بانية تقنيًا. يعني أنه يمكن تنفيذ أي دالة مع `new`، وستُنَفَّذ باستخدام الخوارزمية أعلاه. استخدام الأحرف الكبيرة في البداية هو اتفاق شائع لتمييز الدالة البانية من غيرها وأنَّه يجب استدعاؤها مع `new`.

ا. new function() { … }

إن كان لدينا العديد من الأسطر البرمجية، وجميعها عن إنشاء كائن واحد مُعَقَّد، فبإمكاننا تضمينها في دالة بانية، هكذا:

```
let user = new function() {
  this.name = "Ahmad";
  this.isAdmin = false;

  // ...شيفرة إضافية لإنشاء مستخدم
  // ربما منطق معقد أو أي جمل
  // ..متغيرات محلية وهكذا
};
```

لا يمكن استدعاء المُنشِئ مجددًا، لأنه غير محفوظ في أي مكان، يُنشَأ ويُستدعى فقط. لذا فإن الخدعة تهدف لتضمين الشيفرة التي تُنشِئ كائنًا واحدًا، دون إعادة الاستخدام وتكرار العملية مستقبلًا.

4.5.2 وضع اختبار الباني: new.target

ميزة متقدمة

تُستخدم الصيغة في هذا الجزء نادرًا، ويمكنك تخطيها إلا إن كنت تُريد الإلمام بكل شيء.

يمكننا فحص ما إن كانت الدالة قد استدعيت باستخدام new أو دونه من داخل الدالة، وذلك باستخدام الخاصية الخاصة `new.target`.

تكون الخاصية فارغة في الاستدعاءات العادية، وتساوي الدالة البانية إذا استُدعِيَت باستخدام new:

```
function User() {
  alert(new.target);
}

//  "new" بدون:
User(); // undefined

// باستخدام "new":
new User(); // function User { ... }
```

يمكن استخدام ذلك بداخل الدالة لمعرفة إن استُدعِيَت مع new، "في وضع بناء كائن"، أو بدونه "في الوضع العادي". يمكننا أيضًا جعل كلًا من الاستدعاء العادي و new ينفِّذان الأمر ذاته -بناء كائن- هكذا:

```
function User(name) {
  if (!new.target) { // new إن كنت تعمل بدون
    return new User(name); // new ...سأضيف
  }

  this.name = name;
}

let john = User("Ahmad"); // new User تُوَجِّه الاستدعاء إلى
alert(john.name); // Ahmad
```

يستخدم هذا الأسلوب في بعض المكتبات أحيانًا لجعل الصيغة أكثر مرونة حتى يتمكن الأشخاص من استدعاء الدالة مع new أو بدونه، وتظل تعمل.

ربما ليس من الجيد استخدام ذلك في كل مكان، لأن حذف new يجعل ما يحدث أقل وضوحًا. لكن مع new، يعلم الجميع أن كائنًا جديدًا قد أُنشِئ.

4.5.3 ما تُرجِعه الدوال البانية

لا تملك الدوال البانية عادةً التعليمة return. فَمُهِمَّتُهَا هي كتابة الأمور المهمة إلى this، وتصبح تلقائيًا هي النتيجة. لكن إن كان هناك التعليمة return فإن القاعدة بسيطة:

- إن استُدعِيَت return مع كائن، يُرجَع الكائن بدلًا من this.
- إن استُدعِيَت return مع متغير أولي، يُتَجاهَل.

بمعنى آخر، return مع كائن يُرجَع الكائن، وفي الحالات الأخرى تُرجَع this. مثلًا، يعاد في المثال التالي الكائن المرفق بعد return ويُهمَل الكائن المُسنَد إلى this:

```
function BigUser() {

  this.name = "Ahmad";

  return { name: "Godzilla" };  // <-- تُرجِع هذا الكائن
}

alert( new BigUser().name );  // Godzilla, حصلنا على الكائن
```

وهنا مثال على استعمال return فارغة (أو يمكننا وضع متغير أولي بعدها، لا فرق):

```
function SmallUser() {

  this.name = "Ahmad";

  return; // ← this تُرجِع
}

alert( new SmallUser().name );  // Ahmad
```

لا تحوي الدوال البانية غالبًا على تعليمة الإعادة return. نذكر هنا هذا التصرف الخاص عند إرجاع الكائنات بغرض شمول جميع النواحي.

حذف الأقواس

بالمناسبة، يمكننا حذف أقواس new في حال غياب المعاملات:

```
let user = new User; // <-- لا يوجد أقواس
// الغرض ذاته
let user = new User();
```

لا يُعد حذف الأقواس أسلوبًا جيدًا، لكن الصيغة مسموح بها من خلال المواصفات.

4.5.4 الدوال في الباني

استخدام الدوال البانية لإنشاء الكائنات يُعطي مرونة كبيرة. قد تحوي الدالة البانية على مُعامِلات ترشد في بناء الكائن ووضعه، إذ يمكننا إضافة خاصيات ودوال إلى this بالطبع. مثلًا، تُنشِئ new User(name) في الأسفل كائنًا بالاسم المُعطَى name والدالة sayHi:

```
function User(name) {
  this.name = name;

  this.sayHi = function() {
    alert( "My name is: " + this.name );
  };
}

let john = new User("Ahmad");

john.sayHi(); // My name is: Ahmad

/*
john = {
   name: "Ahmad",
   sayHi: function() { ... }
}
*/
```

لإنشاء كائنات أكثر تعقيدًا، يوجد صيغة أكثر تقدمًا، سنغطيها لاحقًا في فصل "الأصناف" (classes).

4.5.5 الخلاصة

- الدوال البانية، أو باختصار البانيات، هي دوال عادية، لكن يوجد اتفاق متعارف عليه ببدء اسمها بحرف كبير.
- يجب استدعاء الدوال البانية باستخدام `new` فقط. يتضمن هذا الاستدعاء إنشاء كائن فارغ وإسناده إلى `this` وبدء العملية ثم إرجاع هذا الكائن في نهاية المطاف.

يمكننا استخدام الدوال البانية لإنشاء كائنات متعددة متشابهة.

تزود JavaScript دوالَ بانية للعديد من الأنواع (الكائنات) المدمجة في اللغة: مثل النوع `Date` للتواريخ، و `Set` للمجموعات وغيرها من الكائنات التي نخطط لدراستها.

عودة قريبة

غطينا الأساسيات فقط عن الكائنات وبانياتها في هذا الفصل. هذه الأساسيات مهمة تمهيدًا لتعلم المزيد عن أنواع البيانات والدوال في الفصل التالي. بعد تعلم ذلك، سنعود للكائنات ونغطيها بعمق في فصل الخصائص، والوراثة، والأصناف.

4.5.6 تمارين

ا. دالتين - كائن واحد

الأهمية: ★★☆☆☆

هل يمكن إنشاء الدالة `A` و `B` هكذا `new A()==new B()`؟

```
function A() { ... }
function B() { ... }

let a = new A;
let b = new B;

alert( a == b ); // true
```

إن كان ممكنًا، وضح ذلك بمثال برمجي.

الحل:

نعم يمكن ذلك.

إن كان هناك دالة تُرجِع كائنًا فإن `new` تُرجِعه بدلًا من `this`. لذا فمن الممكن، مثلًا، إرجاع الكائن المعرف خارجيًا `obj`:

```
let obj = {};

function A() { return obj; }
function B() { return obj; }

alert( new A() == new B() ); // true
```

4.5.7 إنشاء حاسبة جديدة

الأهمية: ★★★★★

أنشئ دالة بانية باسم `Calculator` تنشئ كائنًا بثلاث دوال:

- `read()` تطلب قيمتين باستخدام سطر الأوامر وتحفظها في خاصيات الكائن.
- `sum()` تُرجِع مجموع الخاصيتين.
- `mul()` تُرجِع حاصل ضرب الخاصيتين.

مثلًا:

```
let calculator = new Calculator();
calculator.read();

alert( "Sum=" + calculator.sum() );
alert( "Mul=" + calculator.mul() );
```

الحل:

```
function Calculator() {

  this.read = function() {
    this.a = +prompt('a?', 0);
    this.b = +prompt('b?', 0);
  };
```

```
    this.sum = function() {
      return this.a + this.b;
    };

    this.mul = function() {
      return this.a * this.b;
    };
  }

  let calculator = new Calculator();
  calculator.read();

  alert( "Sum=" + calculator.sum() );
  alert( "Mul=" + calculator.mul() );
```

4.5.8 إنشاء مجمّع

الأهمية: ★★★★★

أنشئ دالة بانية باسم `Accumulator(startingValue)`، إذ يجب أن يتصف هذا الكائن بأنَّه:

- يخزن القيمة الحالية في الخاصية `value`. تُعَيَّن قيمة البدء عبر المعامل `startingValue` المعطى من الدالة البانية.
- يجب أن تَستخدِم الدالة `read()` الدالة `prompt` لقراءة عدد جديد وإضافته إلى `value`.

بمعنى آخر، الخاصية `value` هي مجموع القيم المدخلة بواسطة المستخدم بالإضافة إلى القيمة الأولية `startingValue`.

هنا مثال على ما يجب أن يُنَفَّذ:

```
let accumulator = new Accumulator(1); // القيمة الأولية 1

accumulator.read(); // يضيف قيمة مدخلة بواسطة المستخدم
accumulator.read(); // يضيف قيمة مدخلة بواسطة المستخدم

alert(accumulator.value); // يعرض مجموع القيم
```

الحل:

```
function Accumulator(startingValue) {
  this.value = startingValue;

  this.read = function() {
    this.value += +prompt('How much to add?', 0);
  };

}

let accumulator = new Accumulator(1);
accumulator.read();
accumulator.read();
alert(accumulator.value);
```

4.6 التسلسل الاختياري "?."

> هذه إضافة حديثة للغة، لذا قد تحتاج المتصفحات القديمة لترقيع هذا النقص وإضافة الدعم اللازم.

التسلسل الاختياري `?.` هو طريقة مقاومة للأخطاء للوصول إلى خاصيات الكائن المتداخلة، حتى إذا كانت الخاصية الوسيطة غير موجودة.

4.6.1 المشكلة

إذا كنت قد بدأت للتو في قراءة هذا الكتاب وتتعلم JavaScript، فلربما لم تواجه هذه المشكلة بعد، لكنَّها شائعة جدًا. فمثلًا، يمتلك بعض من المستخدمين عناوين، لكن هنالك قليل منهم لم يقدمها، لذا لا يمكننا قراءة `user.address.street` بأمان هكذا:

```
let user = {}; // تحدث للمستخدم user في حالة كان ليس لديه عنوان

alert(user.address.street); // خطأ!
```

أو عند تطويرنا لموقع وِب، ونرغب في الحصول على معلومات حول عنصر ما في الصفحة، لكنه قد لا يكون موجودًا:

```
// إذا كان نتيجة querySelector(...) فارغًا
let html = document.querySelector('.my-element').innerHTML;
```

قبل ظهور "?." في اللغة، كان يستخدم المعامل `&&` للتغلب على المشكلة. فمثلًا:

```
let user = {}; // إذا كان user لا يملك عنوان

alert( user && user.address && user.address.street ); // undefined ( أي
ليس خطأ)
```

وكونها تتطلب كتابة طويلة للتأكد من وجود جميع المكونات، لذلك كان استخدامها مرهقًا.

4.6.2 تسلسل اختياري Optional chaining

التسلسل الاختياري "?." يوقف التقييم ويعيد غير معرَّف `undefined` إذا كان الجزء قبل "?." غير معرَّف `undefined` أو فارغ `null`.

للإيجاز سنفترض في هذا الفصل أن شيئًا ما "موجود" إذا لم تكن القيمة "فارغة" `null` أو غير معرَّفة `undefined`. إليك الطريقة الآمنة للوصول إلى `user.address.street`:

```
let user = {}; // إذا كان user لا يملك عنوان

alert( user?.address?.street ); // undefined (ليس خطأً)
```

إن قراءة العنوان باستخدام هذه الطريقة `user?.address` ستعمل حتى ولو كان الكائن `user` غير موجود:

```
let user = null;

alert( user?.address ); // undefined
alert( user?.address.street ); // undefined
```

الرجاء ملاحظة أن: صياغة جملة `?.` تجعل القيمة الموجودة قبلها اختيارية، ولكن ليس القيمة التي تأتي بعدها. في المثال أعلاه، يُسمح التعليمة "user?." للكائن `user` فقط بأن يكون غير معرف أو فارغ"null/undefined".

من ناحية أخرى، إذا كان الكائن `user` موجودًا، فيجب أن يحتوي على خاصية `user.address`، وإلا فإن `user?.address.street` ستُعطي خطأ في النقطة الثانية.

لا تفرط في استخدام التسلسل الاختياري

يجب أن نستخدم `?.` فقط في حالة عدم وجود شيء ما. فمثلًا، بحسب منطق الشيفرة خاصتنا يجب أن يكون الكائن `user` موجودًا، ولكن الخاصية `address` اختيارية، بهذه الحالة سيكون `user.address?.street` أفضل.

لذلك، إذا حدث أن كان الكائن `user` غير معرف بسبب خطأ ما، فسنعرف ذلك ونصلحه إذ من غير المناسب إسكات مثل هذا الخطأ لأنَّ معرفته وتصحيحه آنذاك (أي عند استعمال `user?.address?.street`) سيكون أكثر صعوبة.

يجب التصريح عن المتغير الموجود قبل `?.`

إذا لم يكن هناك متغير `user` على الإطلاق، فإن التعليمة `user?.anything` ستؤدي حتمًا إلى حدوث خطأ:

```
user?.address;
// ReferenceError: user is not defined
```

يجب أن يكون هناك تعريف واضح للمتغير `let / const / var user`. لأن التسلسل الاختياري يعمل فقط للمتغيرات المصرح عنها.

4.6.3 سلوك الطريق المختصر Short-circuiting

كما قلنا سابقًا، ستوقف `?.` فورًا (أي سيُسلَك الطريق الأقصر [Short-circuiting]) إذا لم يكن الجزء الأيسر موجودًا. لذلك، إذا كان هناك أي استدعاءات دوال أخرى أو آثار جانبية، فلن تحدث:

```
let user = null;
let x = 0;

user?.sayHi(x++); // لا يحدث شيء

alert(x); // 0, لم تزداد القيمة
```

4.6.4 حالات أخرى ()?. و []?.

إن التسلسل الاختياري `?.` ليس عامل (operator)، ولكنه طريقة معينة لصياغة تعليمة، يعمل أيضًا مع الدوال والأقواس المربعة. فمثلًا، تُستخدَم `?.()` لاستدعاء دالّة قد لا تكون موجودة. نلاحظ في الشيفرة أدناه، أنه لدى بعض مستخدمينا يملك التابع `admin` والبعض لا يملكه:

```
let user1 = {
  admin() {
    alert("I am admin");
  }
}

let user2 = {};

user1.admin?.(); // I am admin
user2.admin?.();
```

هنا، في كلا السطرين، نستخدم النقطة `.` أولًا للحصول على خاصية `admin`، لأن كائن المستخدم `user` يجب أن يكون موجودًا، لذا فهو آمن للقراءة منه ثم يتحقق `?.()` من الجزء الأيسر: إذا كانت الدالة admin موجودة، فستنفذ (على الكائن `user1`). وبخلاف ذلك (بالنسبة للكائن `user2`) يتوقف التقييم الشيفرة بدون أخطاء.

تعمل الصياغة `?.[]` أيضًا، إذا أردنا استخدام الأقواس `[]` للوصول إلى الخصائص بدلًا من النقطة `.`. على غرار الحالات السابقة، فإنه يسمح بقراءة خاصية بأمان من كائن قد لا يكون موجودًا.

```
let user1 = {
```

```
  firstName: "Ahmad"
};

let user2 = null; // تخيل حالة عدم القدرة على التحقق من موثوقية مستخدم

let key = "firstName";

alert( user1?.[key] ); // Ahmad
alert( user2?.[key] ); // undefined

alert( user1?.[key]?.something?.not?.existing); // undefined
```

كما يمكننا استخدام `?.` مع `delete`:

```
delete user?.name; // احذف المستخدم user.name إذا كان موجودًا
```

لاحظ أنه يمكننا استخدام `?.` للقراءة الآمنة والحذف، ولكن ليس الكتابة، إذ التسلسل الاختياري `?.` ليس له أي فائدة في الجانب الأيسر من المهمة:

```
// فكرة الشيفرة أدناه أن تكتب قيمة user.name إذا لم تكن موجودة

user?.name = "Ahmad"; // خطأ لن تعمل الشيفرة
// undefined = "Ahmad" لأن الإسناد بين
```

4.6.5 الخلاصة

يتكون بناء جملة التسلسل الاختياري `?.` من ثلاثة أشكال:

1. `obj?.prop` - ستُعيد `obj.prop` إذا كان الكائن `obj` موجودًا، وإلا ستعيد القيمة `undefined`.
2. `obj?.[prop]` - ستُعيد `obj[prop]` إذا كان الكائن `obj` موجودًا، وإلا ستُعيد `undefined`.
3. `obj?.method()` - تستدعي `obj.method()` إذا كان الكائن `obj` موجودًا، وإلا ستُعيد `undefined`.

كما رأينا كل الطرائق واضحة وسهلة الاستخدام. فتتحقق `?.` من الجزء الأيسر بحثًا عن قيمة فارغة أو غير معرفة `null/undefined` ويسمح للتقييم بالمتابعة إذا لم يكن كذلك. أي تسمح سلسلة `?.` بالوصول الآمن إلى الخصائص المتداخلة. ومع ذلك، يجب أن نطبق `?.` بحذر، وفقط في الحالات التي يكون فيها الجزء الأيسر غير موجود حتى لا نخفي عن أنفسنا الأخطاء البرمجية إذا حدثت.

4.7 النوع الرمزي Symbol

وفقًا للمواصفات، قد تكون مفاتيح خصائص الكائنات من نوع نصي (string) أو رمزي (Symbol)، ولا تكون أرقامًا ولا قيمًا منطقية، إما نصًا أو رمزًا فقط. حتى الآن، استخدمنا النوع النصي فقط وسنرى الآن فوائد النوع الرمزي (النوع symbol).

4.7.1 الرموز

يُمَثِّل الرمز مُعَرِّفًا فريدًا، ويمكن إنشاء قيمة من هذا النوع باستخدام `Symbol()`:

```
// id هو رمز جديد
let id = Symbol();
```

يمكننا إعطاء وصف للرمز أثناء الإنشاء (ما يُسمَّى أيضًا باسم الرمز)، ويكون مفيدًا لعملية تصحيح الأخطاء:

```
// id هو رمز مع الوصف "id"
let id = Symbol("id");
```

تتصف الرموز بكونها فريدة حتى إن أنشأنا أكثر من رمز بالوصف ذاته، فتبقى الرموز قيمًا مختلفة إذ يُعد الوصف مجرد طابع ولا يؤثر على أي شيء.

مثلًا، هنا رمزين بالوصف ذاته لكنهما غير متساويان:

```
let id1 = Symbol("id");
let id2 = Symbol("id");

alert(id1 == id2); // false
```

إن كانت لديك خلفية عن لغة Ruby أو أي لغة برمجية أخرى تستخدم الرموز فلا تخلط الأمور. فالرموز في JavaScript مختلفة.

انتبه إلى أن الرموز لا تتحول تلقائيًا إلى نص، فتتضمن أغلب القيم في JavaScript تحويلًا ضمنيًا إلى نص. مثلًا، يمكننا عرض أي قيمة تقريبًا باستخدام `alert` وتعمل بشكل صحيح. لكن الرموز خاصة لا تتحول تلقائيًا. مثلا، أمر `alert` التالي سَيعرض خطأ:

```
let id = Symbol("id");
alert(id); // TypeError: Cannot convert a Symbol value to a string
```

يُعد هذا الأمر من حرَّاس اللغة لتجنب الخطأ، لأنَّ النصوص والرموز مختلفة جذريًا ولا يجب أن يتحول نوع منهما للآخر عن طريق الخطأ.

إن كنا نريد عرض رمز ما، يجب أن نستدعي الدالة `.toString()`. كما في المثال التالي:

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id), أصبحت تعمل الآن
```

أو استخدم الخاصية `symbol.description` لعرض الوصف فقط:

```
let id = Symbol("id");
alert(id.description); // id
```

4.7.2 خاصيات "خفية"

تتيح لنا الرموز إنشاء خاصيات "خفية" للكائن، والتي لا يمكن لأي شيفرة الوصول إليها وتغيير محتواها حتى عن طريق الخطأ. مثلا، إن كنا نتعامل مع كائنات تنتمي إلى شيفرة من طرف ثالث ونريد إضافة مُعرفات لها، نستخدم مفتاحًا رمزيًا لذلك:

```
let user = { // ينتمي لِشيفرة أخرى
  name: "Ahmad"
};

let id = Symbol("id");

user[id] = 1;

alert( user[id] ); // يمكننا الوصول إلى البيانات باستخدام الرمز كمفتاح
```

ما الفائدة من استخدام `Symbol("id")` على النص `"id"`؟

بما أن كائنات `user` تنتمي لِشيفرة أخرى، وبما أن الشيفرة أعلاه تتعامل معها فليس آمنًا إضافة أي حقل لها، لكن لا يمكن الوصول إلى الرمز حتى عن طريق الخطأ، قد لا تراه شيفرة الطرف الثالث أيضًا، لذا فإنها الطريقة الصحيحة للقيام بذلك.

تخيل أيضًا أن سكربت آخر يريد إضافة معاملاته الخاصة بِداخل `user` لغرضه الخاص. قد يكون هذا السكربت مكتبة JavaScript أخرى، لذا فإن كل سكربت لا يعلم بتواجد الآخر بتاتًا، ثم يمكن لهذا السكربت إنشاء رمزه الخاص `Symbol("id")` هكذا:

```
// ...
let id = Symbol("id");
```

```
user[id] = "Their id value";
```

بهذه الطريقة، لن يوجد أي تعارض بين المُعاملات التي أنشأناها ومُعاملات السكربت الآخر لأن الرموز تختلف دومًا حتى إن كان لديها الاسم ذاته. لكن إن استخدمنا النص `"id"` بدلًا من الرمز للغرض ذاته، فَسيوجد تعارض:

```
let user = { name: "Ahmad" };

// "id" يستخدم السكربت الذي أنشأناه الخاصية
user.id = "Our id value";

// "id" يريد سكربت آخر استخدام الخاصية لغرض آخر

user.id = "Their id value"
// تعارض! تم تغيير المحتوى من قِبل السكربت الآخر!
```

ا. الرموز في التعريف المختصر لكائن

إن أردنا استخدام الرموز عند تعريف كائن بالطريقة المختصرة عبر الأقواس `{...}` فَسنحتَاج إلى تغليفها بأقواس مربعة هكذا:

```
let id = Symbol("id");

let user = {
  name: "Ahmad",
  [id]: 123 // "id: 123" ليس
};
```

ذلك لأننا نريد القيمة من المتغير `id` كمفتاح وليس النص "id".

4.7.3 تتخطى for…in الرموز

لا تشارك الخصائص الرمزية في الحلقة `for..in`. مثلًا:

```
let id = Symbol("id");
let user = {
  name: "Ahmad",
  age: 30,
  [id]: 123
```

```
};

for (let key in user) alert(key); // name, age (لا يوجد رموز)

// يعمل الوصول المباشر للرمز
alert( "Direct: " + user[id] );
```

يتجاهل `Object.keys(user)` الرموز أيضًا إذ يُعد هذا من جزءًا من مبدأ "إخفاء الخاصيات الرمزيـة". إن حاول سكربت آخر أو مكتبة JavaScript الدخول إلى كائن والتنقل فيه، فلن يصل إلى الخاصـيات الرمزيـة بتاتًـا. في المقابل، تنسخ Object.assign كلًا من النصوص والخاصيات الرمزية:

```
let id = Symbol("id");
let user = {
  [id]: 123
};

let clone = Object.assign({}, user);

alert( clone[id] ); // 123
```

لا يوجد لَبسٌ هنا، فهذا الأمر ضمن التصميم. الفكرة هي أنه عند استنساخ كائن أو دمج كائنين، فإننا نريد أن تُنسَخ جميع الخصائص (بما فيها الرموز مثل `id`).

تُحوَّل مفاتيح الخاصيات التي من نوع آخر غير الرمز إلى نصوص جبريًا.

يمكننا استخدام النصوص أو الرموز فقط كمفاتيح في الكائنات، ويُحوَّل أي نـوع آخـر إلى نص. مثلًا، الـرقم `0` يصبح النص `"0"` عندما يستخدم كمفتاح لخاصية:

```
let obj = {
"test" // "0": "test" مثل
};

// "0" تصل الدالة alert إلى الخاصية ذاتها أي يحوَّل الرقم 0 إلى نص
alert( obj["0"] ); // اختبار
alert( obj[0] ); // (اختبار (الخاصية ذاتها
```

4.7.4 الرموز العامة

جميع الرموز تكون مختلفة دائمًا كما رأينا، حتى إن كانت تحمل الأسماء ذاتها. لكن قد نريد أحيانا أن تكون الرموز التي تحمل الاسم ذاته هي نفس الكائنات. مثلًا، تحتاج أجزاء متعددة في التطبيق الوصول إلى الرمز "id" ما يعني الخاصيات ذاتها.

لنتمكن من القيام بذلك، يوجد "سجل للرموز العامة" (global symbol registry). يمكننا إنشاء رموز فيه والوصول إليها لاحقًا، كما يضمن أن الوصول المتكرر إلى الاسم ذاته يُرجِع الرمز ذاته في كل مرة.

لقراءة، أو إنشاء رمز في حال عدم تواجده في السجل، نستخدم `Symbol.for(key)`. يفحص هذا الاستدعاء السجل العام للرموز، إن كان هناك رمزًا بالوصف `key`، فإنه يُرجِعه، وإن لم يجده فإنه يُنشئ رمزًا جديدًا بالاستدعاء `Symbol(key)` ويُخزِّنه في السجل بالوصف المُعطى `key`. إليك المثال التالي:

```
// يقرأ من السجل العام
let id = Symbol.for("id"); // إن لم يجد الرمز، ينشئه

// يُقرأ مجددًا، ربما من جزء آخر في الشيفرة
let idAgain = Symbol.for("id");

// الرمز ذاته
alert( id === idAgain ); // true
```

تُدعَى الرموز داخل السجل العام رموزًا عامة، ونستعلمها في حال أردنا رمزًا على مستوى تطبيق كامل، وقابلًا للوصول في الشيفرة.

هذا الأمر مشابه لما في لغة Ruby

يوجد في بعض اللغات البرمجية مثل لغة Ruby رمزًا واحدًا لكل اسم. كما رأينا، في JavaScript الأمر ذاته صحيح بالنسبة للرموز العامة.

أ. Symbol.keyFor

بالنسبة للرموز العامة، ليس فقط الدالة `Symbol.for(key)` تُرجِع الرمز وفقًا لاسمه، بل يوجد استدعاء عكسي: `Symbol.keyFor(sym)`، والتي تعكس ما تقوم به الأخرى: تُرجِع اسمًا بواسطة رمز عام. مثلًا:

```
// نَسترجع الرمز بالاسم
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");
```

```
// نسترجع الاسم بالرمز
alert( Symbol.keyFor(sym) ); // name
alert( Symbol.keyFor(sym2) ); // id
```

تستخدم `Symbol.keyFor` سجل الرموز العام داخليًا للبحث عن مفتاح الرمز، لذا فإنها لا تعمل مع الرموز الغير عامة. إن كان الرمز غير عام، فلن يتمكن من العثور عليه وسيُرجِع `undefined`.

يمكن القول أن أي رمز لدية الخاصية `description`.

مثلا:

```
let globalSymbol = Symbol.for("name");
let localSymbol = Symbol("name");

alert( Symbol.keyFor(globalSymbol) ); // name, رمز عام
alert( Symbol.keyFor(localSymbol) ); // undefined, غير عام

alert( localSymbol.description ); // name
```

4.7.5 رموز النظام

يوجد العديد من رموز "النظام" التي تستخدمها JavaScript داخليًا، ويمكن استخدامها لضبط نواحي متعددة من الكائنات بدقة. هذه الرموز مُدرَجة في وصف جدول الرموز المتعارف عليها:

- `Symbol.hasInstance`
- `Symbol.isConcatSpreadable`
- `Symbol.iterator`
- `Symbol.toPrimitive`
- وهكذا...

مثلًا، يتيح لنا الرمز `Symbol.toPrimitive` وصف كائن إلى تحويل أولي وسنرى استخدامها قريبًا. ستعتاد على باقي الرموز عند دراسة ميزات اللغة المُقابلة.

4.7.6 الخلاصة

النوع `Symbol` هو نوع أولي للمُعاملات الفريدة. يتم إنشاء الرموز عبر استدعاء الدالة `Symbol()` مع وصف اختياري (اسم).

الرموز تكون دائمًا مختلفة القيم حتى إن كان لها الاسم أو الوصف ذاته، أي تتصف بأنها فريدة. إن أردنا أن يكون الرمز بالاسم ذاته وبالقيمة ذاتها، يجب أن نستخدم السجل العام: تُرجِع `Symbol.for(key)` (وتُنشِئ في حال الحاجة) رمزًا عامًا بالمفتاح كاسم. تُرجِع الاستدعاءات العديدة لِلدالة `Symbol.for` باستخدام المفتاح ذاته الرمز نفسه.

لدى الرموز حالتي استخدام:

1. خاصيات الكائن "الخفية". إن أردنا إضافة خاصية إلى كائن ينتمي إلى سكربت أو مكتبة أخرى، يمكننا إنشاء رمز واستخدامه كمفتاح خاصية. لا تظهر الخاصية الرمزية في `for..in`، حتى لا تتم معالجته عن طريق الخطأ مع باقي الخاصيات. لن يتم الوصول إليه مباشرةً أيضًا لأن السكربت الخارجي لا يحوي على هذا الرمز. هكذا تكون الخاصية محمية من الاستخدام الخارجي، لذا، يمكننا إخفاء شيء ما بشكل تام في الكائنات التي نحتاج إليها، والتي لا يجب أن يراها الآخرون باستخدام الخاصيات الرمزية.
2. يوجد العديد من رموز النظام المستخدمة بواسطة JavaScript التي يمكن الوصول إليها بواسطة `Symbol.*`. يمكننا استخدامها لتعديل سلوك مُدمَج، مثلا، سنستخدم `Symbol.iterato` لاحقًا في الشرح للحلقات، و `Symbol.toPrimitive` لإعداد التحويل من كائن لقيمة أولية وهكذا...

عمليًا، الرموز ليست خفية 100%. يوجد دالة مدمجة Object.getOwnPropertySymbols(obj) تتيح لنا الوصول إلى جميع الرموز. كما يوجد دالة تُدعى Reflect.ownKeys(obj) والتي تُرجِع جميع مفاتيح الكائن بما فيها الرموز. لذا فإن الرموز ليست مخفية فعلًا، لكن أغلب المكاتب، والدوال المدمجة والهياكل لا تستخدم هذه الدوال.

4.8 التحويل من نوع كائن إلى نوع أولي

ماذا يحدث عند جمع كائنين بالشكل `obj1 + obj2`، أو طرح كائنين `obj1 - obj2` أو طباعة الكائنات باستخدام `alert(obj)`؟ تحوَّل الكائنات في مثل هذه الحالات إلى أنواع أولية (primitives) ثم تُنَفَّذ العملية.

رأينا في فصل تحويل الأنواع قواعد التحويل بين الأنواع الأساسية مثل الأعداد، أو النصوص أو القيم المنطقية، لكن بقيت فجوة متعلقة بالكائنات. الآن بعد أن تعرفنا على الرموز والدوال، يمكن ملء هذه الفجوة.

1. في السياق المنطقي (boolean)، جميع الكائنات تساوي القيمة `true`، إذ عمليات التحويل المتاحة للكائنات هي التحويل إلى أرقام ونصوص فقط.
2. يحدث التحويل العددي (numeric) عند طرح الكائنات أو تنفيذ العمليات الرياضية عليها. مثلًا، يمكن طرح كائنات `Date` (سيتم شرحها في فصل التاريخ والوقت) ويكون ناتج طرح `date1 - date2` هو الفارق الزمني بين التاريخين.
3. بالنسبة للتحويل إلى نص (string)، يحدث ذلك عادةً عند إخراج كائن بالطريقة `alert(obj)` وأي سياق مشابه.

4.8.1 التحويل إلى أنواع أساسية عبر toPrimitive

يمكننا ضبط التحويل إلى نص أو عدد باستخدام دوال خاصة بالكائنات. يوجد ثلاثة أنواع للتحويل بين الأنواع (يطلق عليها "النوع المخمَّن" [hint] الذي سيجري تحويل الكائن إليه)، مشروحة في هذه المواصفات:

ا. إلى سلسلة نصية string

يُجرَى التحويل من كائن لنص عند القيام بعمليات على كائن تتوقع أن يكون الكائن نصًا، مثل `alert`:

```
// المخرجات
alert(obj);

// استخدام كائن كمفتاح للخاصية
anotherObj[obj] = 123;
```

ب. إلى عدد number

يجري تحويل كائن لعدد عند القيام بعمليات حسابية:

```
// تحويل صريح
let num = Number(obj);
```

```
// عمليات رياضية (عدا binary plus)
let n = +obj; // عملية جمع أحادية
let delta = date1 - date2;

// عملية موازنة أصغر/أكبر
let greater = user1 > user2;
```

ج. إلى نوع افتراضي default

يحدث في حالات نادرة عندما لا يكون المُشَغِّل متأكدًا من نوع البيانات المُتوَقَّع. فمثلًا، يمكن لِمُعامِل الجمع الثنائي + أن يتعامل مع كلًا من النصوص (دمج السلسلتين) ومع الأرقام (بجمعها)، لذا فإن كلًا من النصوص والأرقام تعمل بشكل صحيح. أيضًا، عند موازنة كائن بنص، أو رقم أو رمز، فلا يكون واضحًا أي نوع من التحويلات هو المطلوب.

```
// الجمع الثنائي
let total = car1 + car2;

// obj == string/number/symbol
if (user == 1) { ... };
```

يمكن لِمعاملات الموازنة أكبر > /أصغر < التعامل أيضًا مع كل من النصوص والأعداد لكنها ترجح التحويل إلى عدد دون النظر إلى التحويل الافتراضي المرجح في هذه الحالة وذلك لأسباب تاريخية.

عمليًا، تُطبِّق جميع الكائنات المُدمجة في اللغة التحويل الافتراضي `"default"` بنفس طريقة التحويل إلى عدد `"number"`. وربما يجب أن نقوم بذلك أيضًا. (عدا الكائن `Date`، والذي سَنتعلمه لاحقًا.)

لاحظ أن هناك ثلاثة أنواع للتحويل فقط، إذ الأمر بهذه البساطة. لا يوجد تحويل للنوع المنطقي (boolean)، فجميع الكائنات تحمل القيمة `true` في السياق المنطقي) أو أي شيء آخر. وإن عاملنا كلًا من التحويل الافتراضي `"default"` والتحويل العددي `"number"` بالطريقة ذاتها كما تقوم أغلب الدوال المدمجة، فسيكون هنالك تحويلين فقط.

تحاول JavaScript العثور على ثلاث دوال للكائن واستدعائها عند القيام بالتحويل

1. استدعاء `obj[Symbol.toPrimitive](hint)` - الدالة ذات المفتاح الرمزي `Symbol.toPrimitive` (رمز نظام)، إن كانت هذه الدالة موجودة.
2. أو إن كان النوع المخمَّن هو نص `"string"`
    - جرب `obj.toString()` و `obj.valueOf()`، أيًا كانت متواجدة.

3. إن كان hint هو عدد `"number"` أو `"default"`
   - جرب `obj.valueOf()` و `obj.toString()` أيًا كانت متواجدة.

د. تابع التحويل Symbol.toPrimitive

لنبدأ من التابع الأول؛ يوجد رمز مُضَمَّن في JavaScript مُسَمَّى `Symbol.toPrimitive` والذي يجب استخدامه لتسمية تابع التحويل، هكذا:

```
obj[Symbol.toPrimitive] = function(hint) {
  // يجب أن تُرجِع قيمة أولية
  // النوع المخمَّن هو إما نص أو عدد أو النوع الافتراضي
};
```

مثلا، الكائن `user` هنا يتضمنها:

```
let user = {
  name: "Ahmad",
  money: 1000,

  [Symbol.toPrimitive](hint) {
    alert(`hint: ${hint}`);
    return hint == "string" ? `{name: "${this.name}"}` : this.money;
  }
};

// تجربة للتحويل:
alert(user); // hint: string -> {name: "Ahmad"}
alert(+user); // hint: number -> 1000
alert(user + 500); // hint: default -> 1500
```

كما يمكننا أن نرى في الشيفرة، أصبح `user` نصًا يصف نفسه أو كمية من المال وفقًا للتحويل. تقوم الدالة `user[Symbol.toPrimitive]` بجميع حالات التحويل.

ه. التحويل إلى نص عبر toString أو valueOf

ظهر التابعان `toString` و `valueOf` منذ وقت طويل، ولا يتبعان إلى نوع الرمز (symbol)، إذ لم تكن الرموز موجودة في ذلك الحين). لكنَّهما تابعين مسميان بأسماء توحي بارتباطهما بالنوع النصي، إذ كانت توفر آلية لعملية التحويل أصبحت قديمة النمط الآن.

إن لم يكن هناك تنفيذًا للتابع `Symbol.toPrimitive` فتحاول JavaScript إيجاد هذه الـدوال بـالترتيب التالي:

- `toString -> valueOf` للنصوص.
- `valueOf -> toString` لباقي الأنواع.

مثلا، يقوم الكائن `user` بنفس الغرض السابق باستخدام `toString` و `valueOf`:

```
let user = {
  name: "Ahmad",
  money: 1000,

  // في حال النوع المخمَّن سلسلة نصية
  toString() {
    return `{name: "${this.name}"}`;
  },

  // في حال النوع المخمَّن عدد أو النوع الافتراضي
  valueOf() {
    return this.money;
  }

};

alert(user); // toString -> {name: "Ahmad"}
alert(+user); // valueOf -> 1000
alert(user + 500); // valueOf -> 1500
```

كما رأينا، يُنفَّذ السلوك ذاته كما في المثال السابق الذي استخدم `Symbol.toPrimitive`.

نحتاج غالبًا إلى مكان واحد يقوم بجميع التحويلات للأنواع الأوليـة (نص أو عـدد). ففي هـذه الحالـة، يمكننـا استخدام التابع `toString` فقط لينفِّذ جميع عمليات التحويل، كما يلي:

```
let user = {
  name: "Ahmad",

  toString() {
    return this.name;
```

```
  }
};

alert(user); // toString -> Ahmad
alert(user + 500); // toString -> Ahmad500
```

سيُنفَّذ التابع toString بجميع التحويلات الأولية في غياب كلًا من Symbol.toPrimitive و valueOf.

4.8.2 الأنواع المعادة

الأمر المهم الذي يجب معرفته عن جميع دوال التحويل بين الأنواع الأولية هو أنها لا تُرجِع بالضرورة النوع المخمَّن (hint) الأولي ذاته. لا يوجد تحكم في ما إن كانت toString() ترجع نصًا، أو أن Symbol.toPrimitive تُرجِع عددًا عند تحويل عدد. الأمر الوحيد المعروف والثابت هو أنَّ هذه الدوال تُرجِع نوعًا أوليًا وليس كائنًا.

ملاحظة تاريخية

لا يوجد خطأ إن أعاد التابع toString أو valueOf كائنًا، وذلك لأسباب تاريخية، لكن يتم تجاهل مثل هذه القيم (وكأن الدالة ليست موجودة). وذلك لعدم وجود مبدأ الخطأ الجيد (good error) في JavaScript حينها. بالمقابل، يجب أن يعيد التابع Symbol.toPrimitive قيمة أولية، وإلا فسيكون هناك خطأ.

4.8.3 عمليات تحويل إضافية

عرفنا مسبقًا أن جميع العوامل (operator) والدوال تجري عمليات تحويل على الأنواع التي تتعامل معها مثل عامل الضرب * يحول جميع المُعامَلات (operands) إلى أعداد. فإن مرَّرنا كائنًا عبر وسيط إلى إحدى الدوال أو العمليات، فستمر عملية التحويل على مرحلة أو مرحلتين هما:

1. يحوَّل الكائن إلى نوع أولي (باستعمال القواعد التي تحدثنا عنها آنفًا)
2. إن لم يكن النوع الأولي الناتج مطابقًا للنوع المطلوب، فيحوَّل إلى النوع المطلوب وفق مبدأ التحويل بين الأنواع الأولية

إليك المثال التالي:

```
let obj = {
  // يجري التابع toString جميع التحويلات في غياب باقي الدوال
  toString() {
    return "2";
```

```
    }
  };

  alert(obj * 2); // 4, حُوِّل الكائن إلى القيمة الأولية "2"، ثم جعلته عملية الضرب عددًا
```

1. حوَّلت عملية الضرب `obj * 2` الكائن إلى نوع أولي، وكان النوع المعاد من عملية التحويل سلسلةً نصية، هي `"2"`.
2. جرى بعدئذٍ تحويل تلك السلسلة في العملية `"2" * 2`، إلى عدد ليصبح `2 * 2`.

في المثال التالي، يقوم الجمع الثنائي بدمج النصوص في هذه الحالة والاكتفاء بعملية تحويل الكائن إلى سلسلة نصية:

```
  let obj = {
    toString() {
      return "2";
    }
  };

  alert(obj + 2); // 22 (ارجع التحويل إلى نوع أولي نصًا => دمج)
```

4.8.4 الخلاصة

تُجرَى عملية التحويل من كائن لنوع أولي تلقائيًا بواسطة العديد من الدوال المدمجة في اللغة والعاملات التي تتوقع أن تتعامل مع قيم أولية.

يخمَّن النوع الأولي المراد تحويل الكائن إليه إلى:

- سلسلة نصية `"string"` (للدالة `alert` والعمليات الأخرى التي تتعامل مع نصوص)
- عدد `"number"` (للعمليات الرياضية)
- النوع الافتراضي `"default"` (لبعض العوامل)

تُحدِّد المواصفات النوع المخمَّن (hint) الذي يستخدمه كل مُعامل بوضوح. يوجد القليل من العوامل التي لا تعلم ما النوع المتوقع وتستخدم النوع الافتراضي `"default"`. يُعامل النوع الافتراضي `"default"` معاملة العدد `"number"` في الكائنات أغلب الأحيان؛ لذا، يتم دمج النوعين عمليًا.

خوارزمية التحويل كالتالي:

1. استدعاء `obj[Symbol.toPrimitive](hint)` إن وُجِدَت،

2. أو إن كان النوع المخمَّن هو نص `"string"`
   - جرب `obj.toString()` و `obj.valueOf()`، أيًا كانت متواجدة.
3. إن كان النوع المُخمَّن هو عدد `"number"` أو `"default"`
   - جرب `obj.valueOf()` و `obj.toString()` أيًا كانت متواجدة.

عمليًا، يكفي استخدام `obj.toString()` فقط لإجراء جميع التحويلات، إذ تعيد "شيئًا مقروءًا" يمثِّل الكائن يستعمل هذا التمثيل لعملية التسجيل أو التنقيح.

مستقل
mostaql.com

ادخل سوق العمل ونفذ المشاريع باحترافية
عبر أكبر منصة عمل حر بالعالم العربي

ابدأ الآن كمستقل

5. أنواع البيانات

يتضمن هذا الفصل الأقسام التالية:

1. توابع الأنواع الأولي
2. النوع number: الأعداد
3. النوع string: السلاسل النصية
4. المصفوفات Arrays
5. توابِع المصفوفات
6. المُكرَّرات Iterables
7. النوع Map الخرائط والنوع Set الأطقم
8. النوع WeakMap والنوع WeakSet
9. مفاتيح الكائنات وقيمها ومدخلاتها
10. الإسناد بالتفكيك
11. النوع Date: التاريخ والوقت
12. صيغة JSON وتوابعها

5.1 توابع الأنواع الأولية

تتيح لنا JavaScript التعامل مع أنواع البيانات الأولية (النصوص، الأرقام، وغيرها) كما لو أنها كائنات، كما تزودنا بتوابع (methods) لاستدعائها كما الكائنات. وسندرس هذه التوابع قريبًا، لكن لنرى أولًا كيف تعمل لأن الأنواع الأولية (primitives) ليست كائنات (أي object) وسنجعل الأمر واضحًا هنا.

لنرى إلى الفروق الأساسية بين الأنواع الأولية والكائنات.

النوع الأولي:

- هو قيمة من نوع أولي (primitive type).
- يوجد 6 أنواع أولية: نص `string`، رقم `number`، قيمة منطقية `boolean`، رمز `symbol`، قيمة فارغة `null` ، وقيمة غير معرفة `undefined`.

الكائن:

- قادر على تخزين العديد من القيم ضمن خاصيات.
- يمكن إنشاؤه باستخدام `{}`، مثلًا: `{name: "Ahmad", age: 30}`. يوجد أنواع أخرى من الكائنات في JavaScript: مثلا، تعد الدوال كائنات.

أحد أفضل الأشياء بالنسبة للكائنات هو إمكانية تخزين دالة في خاصية من خواص هذا الكائن.

```
let john = {
  name: "Ahmad",
  sayHi: function() {
    alert("Hi buddy!");
  }
};

john.sayHi(); // Hi buddy!
```

إذًا، أنشأنا الكائن `john` محتويًا الدالة `sayHi`.

يوجد العديد من الكائنات المدمجة في اللغة، مثل التي تتعامل مع التواريخ، الأخطاء، عناصر HTML، وغيرها، ولديها خاصيات وتوابع مختلفة، لكن لهذه الميزات ثمن!

الكائنات أثقل من المتغيرات الأولية، فهي تتطلب موارد (resources) أكثر لدعم آليتها الداخلية.

5.1.1 نوع أولي مثل كائن

هنا نجد التناقض الذي واجهه مُنشِئ JavaScript:

- يوجد الكثير من الأشياء التي قد يريد أحد القيام بها مع المتغيرات الأولية مثل النص أو الرقم. سيكون من الرائع الوصول إليها كتوابع.
- يجب أن تكون المتغيرات الأولية سريعة وخفيفة قدر الإمكان.

يبدو الحل صعبًا قليلًا، لكن هذا هو:

1. تبقى المتغيرات الأولية كما هي، قيمة واحدة مثل المطلوب.
2. تتيح لنا اللغة الوصول إلى توابع وخاصيات النصوص، الأرقام، القيم المنطقية، والرموز.
3. حتى يعمل ذلك، يُنشَأ "كائن مغلِّف" (object wrapper) خاص يزود المتغيرات بالوظائف الإضافية، ثم يُدَمَّر.

يختلف الكائن المغلِّف "object wrappers" من نوع أولي لآخر ويُسمَى: `String`، و `Number`، و `Boolean`، و `Symbol`، لذا فإنه يزود كل نوع بمجموعة مختلفة من التوابع الخاصة به.

مثلا، يوجد دالة للنصوص str.toUpperCase() والتي تُرجِع النص `str` بأحرف كبيرة.

آلية عملها:

```
let str = "Hello";

alert( str.toUpperCase() ); // HELLO
```

الأمر بسيط، أليس كذلك؟ ما يحدث فعلا في الدالة `str.toUpperCase()` هو كالتالي:

1. النص `str` هو متغير أولي، لذا فعند محاولة الوصول إلى خاصيتِه، يُنشَأ كائن خاص يعرف قيمة النص ويحتوي هذا الكائن على توابع مفيدة من بينها التابع `toUpperCase()`.
2. تعمل هذه الدالة وتُرجِع نصًا جديدًا (يمكن عرضه باستخدام `alert`).
3. يُدَمَّر الكائن الخاص تاركًا المتغير الأولي `str`.

هكذا يمكن للمتغيرات الأولية أن تحوي توابع، وتبقى خفيفة في الوقت ذاته. يُحَسِّن محرك JavaScript هذه العملية بدرجة عالية. قد يتخطى إنشاء الكائن الإضافي، لكن يجب أن يظل قائمًا بالعمل المطلوب كما لو كان قد أنشأ الكائن.

لدى الأعداد توابع خاصة بها، مثلا، toFixed(n) تُقرِّب الرقم المُعطَى إلى الدقة المطلوبة:

```
let n = 1.23456;

alert( n.toFixed(2) ); // 1.23
```

سنرى المزيد من التوابع المخصصة في فصل النصوص والأعداد.

ا. البانيات String أو Number أو Boolean هي للاستخدام الداخلي فقط

تتيح لنا بعض اللغات مثل Java إنشاء كائنات مغلِّفة "wrapper objects" بشكل صريح باستخدام صيغة مثل: `new Number(1)` أو `new Boolean(false)`. ذلك ممكن أيضًا في JavaScript لأسـباب تاريخيـة، لكنه غير مستحسن لأن الأمور قد تسير بشكل خاطئ في العديد من الأماكن. إليك المثال التالي:

```
alert( typeof 0 ); // "number"

alert( typeof new Number(0) ); // "object"!
```

تكون قيمة الكائنات دائمًا `true` في `if`، لذا سيتم عرض ما بداخل `alert` أدناه:

```
let zero = new Number(0);

if (zero) { //
  alert( "zero is truthy!?!" );
}
```

تقييم قيمة المتغير `zero` هنا هي القيمة المنطقية `true`، لأنه كائن.

بالمقابل، من الممكن اسـتخدام التوابـع `String/Number/Boolean` بـدون `new`، إذ تقـوم هـذه التوابـع بتحويل القيمة إلى النوع المقابل: إلى نص، أو رقم، أو قيمة منطقية (أولية).

مثال: الأمر التالي صحيح تمامًا:

```
let num = Number("123"); // تحوِّل النص إلى رقم
```

ب. ليس لدى القيمتان الأوليتان null/undefined توابع

النوعان الأوليان `null` و `undefined` هما حالة استثناء، فليس لديها كـائن مغلِّف wrapper object ولا توابع، إذ يعدان من الأنواع الأكثر أولية.

ستتسبب المحاولة في الوصول إلى خاصية بظهور خطأ:

```
alert(null.test); // error
```

5.1.2 الخلاصة

- لدى الأنـواع الأوليـة توابـع مسـاعدة عـدا `null` و `undefined` تسـهل التعامـل معهـا، سندرسـها في الفصول اللاحقة.
- تعمل هذه التوابع عبر كائنات مؤقتة، لكن محرك JavaScript مُعـد لتحسـين العمليـة داخليًـا. لـذا فـإن استدعاء الكائن لا يتطلب الكثير من الموارد.

5.1.3 المهام

ا. هل من الممكن إضافة خاصية نصية؟

الأهمية: ★★★★★

خذ بالحسبان الشيفرة التالية:

```
let str = "Hello";

str.test = 5;

alert(str.test);
```

ماذا تظن؟ هل ستعمل؟ هل ستُعرَض؟

الحل:

جرب تشغيلها:

```
let str = "Hello";

str.test = 5; // (*)

alert(str.test);
```

يعتمد الأمر على إن كنت تستخدم `use strict` أم لا، قد تكون النتيجة أحد الخيارين:

1. `undefined` بدون استخدام الوضع الصارم
2. خطأ في الوضع الصارم

لماذا؟ لنفكر فيمَ يحصل في السطر `(*)`:

1. يُنشئ "wrapper object" عند محاولة الوصول إلى خاصية للمتغير `str`.

2. الكتابة إلى هذه الخاصية يُعَدُّ خطأ في الوضع الصارم.
3. في الحالة الأخرى، تستمر عملية التخزين في الخاصية، يحصل الكائن على الخاصية `test`. لكن، يُدمَّر الكائن بعد ذلك فلا يصبح لدى `str` مرجِعًا إليه مما يجعل قيمتها غير معرفة.

يوضح هذا المثال أن المتغيرات الأولية ليست كائنات.

5.2 النوع number: الأعداد

يوجد نوعان من الأعداد في JavaScript:

1. أعداد عادية تخزَّن بصيغة 64-بت IEEE-754، تُعرف أيضًا باسم "الأعداد العشرية مضاعفة الدقة" (double precision floating point numbers). هذا النوع هو ما سنستعمله أغلب الوقت وسنسلط عليه الضوء في هذا الفصل.
2. أعداد صحيحة كبيرة (BigInt numbers) تمثِّل عددًا صحيحًا متغير الحجم، إذ قد نلجأ إليها أحيانًا لأن النوع السابق لا يمكن أن يتجاوز القيمة 2^53 أو أن تقل عن -2^53، وسنخصص لهذا النوع فصلًا خاصًا به نظرًا للحاجة إليه في حالات خاصة.

حاليًا، لِنتوسع عن ما نعرفه عنها، وننتقل إلى الحديث عن النوع الأول، الأعداد العادية.

5.2.1 طرائق أخرى لكتابة عدد

تخيل أننا نريد كتابة 1 بليون. الطريقة الواضحة هي:

```
let billion = 1000000000;
```

لكن، نتجنب غالبًا كتابة سلسلة طويلة من الأصفار في الحياة الواقعية لأنه من السهل الخطأ في ذلك ولكون ذلك الأمر يأخذ وقتًا أكثر. نكتب غالبًا شيئا مثل "1bn" بدلًا من بليون أو "7.3bn" بدلًا من 7 بليون و 300 مليون. يمكن تطبيق الأمر ذاته مع الأعداد الكبيرة.

نُقَصِّر أرقام الأعداد في JavaScript بإضافة الحرف "e" للعدد وتحديد عدد الأصفار فيه:

```
let billion = 1e9;  // بليون، حرفيًا: 1 وجانبه 9 أصفار

alert( 7.3e9 );  // 7,300,000,000
```

بمعنى آخر، يضرب الشكل "XeY" العدد X في 1 متبوعًا بعدد Y من الأصفار.

```
1e3 = 1 * 1000
1.23e6 = 1.23 * 1000000
```

لنكتب الآن شيئًا صغيرًا جدًا. مثلًا، جزء من المليون من الثانية:

```
let ms = 0.000001;
```

كما قمنا سابقًا، يمكن استخدام "e" لتجنب كتابة الأصفار، يمكننا القول:

```
let ms = 1e-6; // ستة أصفار على يسار 1
```

إن قمنا بعد الأصفار في 0.000001، سنجد عددها 6. لذا يكون الرقم 1e-6. بمعنى آخر، وجود رقم سالب بعد "e" يعني القسمة على 1 متبوعًا بِعدد الأصفار المعطى:

```
// -3 بالقسمة على 1 متبوعًا ب 3 أصفار
1e-3 = 1 / 1000 (=0.001)

// -6 بالقسمة على 1 متبوعًا ب 6 أصفار
1.23e-6 = 1.23 / 1000000 (=0.00000123)
```

ا. الأعداد الست عشرية، والثنائية والثمانية

تُستخدَم الأعداد الست عشرية بكثرة في JavaScript لتمثيل الألوان، وتشفير الأحرف ولأشياء أخرى عديدة. لذا فإن هناك طريقة أقصر لكتابتها وذلك بوضع السابقة 0x ثم الرقم. مثلًا:

```
alert( 0xff ); // 255
alert( 0xFF ); // 255 (العدد ذاته, لا يوجد اختلاف باختلاف حالة الأحرف)
```

تستخدم الأنظمة الثنائية والثمانية نادرًا، لكنها مدعومة أيضًا باستخدام السابقة 0b والسابقة 0o على التوالي:

```
let a = 0b11111111; // الهيئة الثنائية لِلعدد 255
let b = 0o377; // الهيئة الثمانية لِلعدد 255
alert( a == b ); // صحيح، العدد ذاته 255 في كلا الجانبين
```

يوجد ثلاثة أنظمة عددية فقط مدعومة بالشكل السابق. يجب استخدام الدالة parseInt لباقي الأنواع (سنشرحها لاحقًا في هذا الفصل).

5.2.2 toString(base)

يُرجِع التابع num.toString(base) تمثيلًا نصيًا للمتغير num إلى النظام العددي المُعطى base. مثلًا:

```
let num = 255;
alert( num.toString(16) );  // ff
alert( num.toString(2) );   // 11111111
```

يمكن أن تختلف قيمة base من 2 حتى 36، والقيمة الافتراضية هي 10.

حالات الاستخدام الشائعة:

- base=16: تستخدم للألوان الست عشرية، وتشفير الأحرُف وغيرها، قد تحوي الخانات الأرقام 0..9 أو الأحرف A..F.
- base=2: يستخدم بكثرة في تصحيح العمليات الدقيقة، يمكن أن يحوي الرقمين 0 أو 1.
- base=36: هو الحد الأعلى، يمكن أن يحوي الأرقام 0..9 أو الأحرُف A..Z. يمكن استخدام جميع الأحرف اللاتينية لتمثيل عدد. قد يبدو أمرًا ممتعًا لكن يكون مفيدًا في حال احتجنا لتحويل معرف عددي طويل إلى عدد أقصر، مثلًا، لتقصير رابط url. يمكن تمثيله بالنظام العددي ذي الأساس 36:

```
alert( 123456..toString(36) ); // 2n9c
```

نقطتين لِاستدعاء تابع

لاحظ أن النقطتين في 123456..toString(36) ليست خطأ كتابي. إن أردنا استدعاء تابع مباشرة على عدد، مثل التابع toString في المثال أعلاه، فسنحتاج إلى نقطتين بعد العدد ...

إن وضعنا نقطة واحدة فقط 123456.toString(36) فسيكون هناك خطأ، لأن JavaScript ستَعتبر أن النقطة هي فاصلة عشرية وأن ما بعدها هو جزء عشري للعدد. فإذا وضعنا نقطة أخرى فستعرف أن الجزء العشري فارغ وتنتقل إلى الدالة.

يمكن كتابتها بهذه الطريقة أيضًا (123456).toString(36).

5.2.3 التقريب Rounding

أحد الخصائص المستخدمة عند التعامل مع الأعداد هي التقريب، يوجد العديد من دوال للتقريب هي:

- Math.floor: تقريب للجزء الأصغر: 3.1 تصبح 3، و ‎-1.1 تصبح ‎-2.
- Math.ceil: تقريب للجزء الأكبر: 3.1 تصبح 4، و ‎-1.1 تصبح ‎-1.
- Math.round: تقريب لأقرب عدد صحيح: 3.1 تصبح 3، 3.6 تصبح 4 و ‎-1.1 تصبح ‎-1.
- Math.trunc (ليست مدعومة بواسطة المتصفح Internet Explorer): تحذف أي شيء بعد الفاصلة العشرية بدون تقريب: 3.1 تصبح 3، ‎-1.1 تصبح ‎-1.

يختصر الجدول في الأسفل الاختلافات بين هذه التوابع:

	Math.floor	Math.ceil	Math.round	Math.trunc
3.1	3	4	3	3
3.6	3	4	4	3
-1.1	-2	-1	-1	-1
-1.6	-2	-1	-2	-1

تُعَطِّي هذه التوابع جميع الاحتمالات الممكنة للتعامل مع الجزء العشري للعدد، لكن ماذا إن كنا نريد تقريب العدد إلى خانة محدَّدة بعد الفاصلة العشرية؟

مثلًا، لدينا العدد `1.2345` ونريد تقريب إلى خانتين لنحصل على `1.23` فقط. يوجد طريقتين للقيام بذلك:

أولًا، الضرب والقسمة:مثلًا، لتقريب الرقم إلى الخانة الثانية بعد الفاصلة العشرية، يمكننا ضرب العدد في 100، ثم نستدعي تابع التقريب ثم نقسم على نفس العدد.

```
let num = 1.23456;

alert( Math.floor(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
```

ثانيًا، يقرب التابع toFixed(n) العدد المستدعى معه إلى الخانة n بعد الفاصلة العشرية ويُرجِع تمثيلًا نصيًا للنتيجة.

```
let num = 12.34;
alert( num.toFixed(1) ); // "12.3"
```

يعمل التابع على تقريب العدد للأكبر أو الأصغر وفقًا إلى أقرب قيمة، مثل التابع `Math.round`:

```
let num = 12.36;
alert( num.toFixed(1) ); // "12.4"
```

لاحظ أن مخرجات التابع `toFixed` هي نص. إن كان الجزء العشري أقل من المطلوب، تُضاف الأصفار إلى نهاية الرقم:

```
let num = 12.34;
alert( num.toFixed(5) ); // "12.34000", 5 أصفار مضافة لجعل عدد الخانات
```

يمكننا تحويل المخرجات إلى عدد باستخدام الجمع الأحادي أو باستدعاء الدالة `Number()`: `+num.toFixed(5)`.

5.2.4 حسابات غير دقيقة

يُمَثَّل العدد داخليًا بصيغة 64-بِت IEEE-754، لذا يوجد 64 بِت لتخزين العدد: تستخدم 52 منها لتخزين أرقام العدد، و 11 منها لتخزين مكان الفاصلة العشرية (تكون أصفارًا للأعداد الصحيحة)، و 1 بِت لإشارة العدد.

إن كان العدد كبيرًا جدًا، فَسيزداد عن مساحة التخزين 64-بِت، معطيًا ما لا نهاية:

```
alert( 1e500 ); // Infinity
```

ما قد يكون أقل وضوحًا، ويحدث غالبًا هو ضياع الفاصلة. لاحظ الاختبار الخطأ التالي:

```
alert( 0.1 + 0.2 == 0.3 ); // false
```

الجملة السابقة تحدث فعليًا، إن فحصنا ما إن كان مجموع 0.1 و 0.2 هو 0.3، نحصل على false. غريب أليس كذلك؟! ما النتيجة إذًا إن لم تكن 0.3؟

```
alert( 0.1 + 0.2 ); // 0.30000000000000004
```

يوجد خطأ آخر هنا غير الموازنة الخطأ. تخيل أننا نقوم بموقع للتسوق الإلكتروني، ووضع الزائر بضائع بقيم $ 0.10 و $0.20 في السلة. سيكون مجموع الطلب $0.30000000000000004. مما قد يفاجئ أي أحد. لكن السؤال الأهم، لم يحدث هذا؟

يُخَزَّن العدد في الذاكرة بهيئته الثنائية، سلسلة من البِت - واحدات وأصفار. لكن الأجزاء مثل 0.1 و 0.2 والتي تبدو بسيطة بالنسبة للنظام العددي العشري هي في الحقيقة أجزاء غير منتهية في النظام الثنائي.

بمعنى آخر، ما هو 0.1؟ هو واحد مقسوم على عشرة 1/10، عُشر. ويكون من السهل تمثيله بنظام الأعداد العشري. موازنة بالثلث: 1/3. الذي يصبح بكسور غير منتهية 0.33333(3).

لذا، فإن من المؤكد أن تعمل الأعداد المقسومة على مضاعفات العدد 10 في النظام العشري، ولا تعمل المقسومة على 3. وكذلك أيضًا في النظام الثنائي، تعمل الأعداد المقسومة على مضاعفات العدد 2، لكن يصبح العدد 1/10 كسورًا ثنائية غير منتهية.

لا يوجد طريقة لتخزين العدد ذاته 0.1 أو 0.2 بالضبط باستخدام النظام الثنائي، كما لا يمكن تخزين الثلث كجزء عشري. يحل نمط الأعداد IEEE-754 هذه المشكلة بتقريب العدد إلى أقرب عدد ممكن. لا تتيح لنا قواعد التقريب هذه رؤية خسارة الأجزاء الصغيرة، لكنها تكون موجودة. يمكننا رؤية ذلك فعليًا:

```
alert( 0.1.toFixed(20) ); // 0.10000000000000000555
```

وعند جمع عددين، فإن الأجزاء المفقودة تظهر.

هذا يفسر لِمَ 0.1 + 0.2 لا تساوي 0.3 بالضبط.

ليس فقط في JavaScript

هذه المشكلة موجودة في العديد من اللغات البرمجية الأخرى مثل PHP، و Java، و C، و Perl، و Ruby تُعطي النتيجة ذاتها، لأنها تعتمد على الصيغة العددية ذاتها.

هل يمكننا تجنب المشكلة؟ بالطبع، أفضل طريقة هي بتقريب النتيجة بمساعدة التابع toFixed(n):

```
let sum = 0.1 + 0.2;
```

```
alert( sum.toFixed(2) ); // 0.30
```

يرجى ملاحظة أن `toFixed` تُرجِع نصًا دائمًا. وتتأكد من وجود خانتين فقط بعد العلامة العشرية. هذا يجعل الأمر مريحًا إن كان لدينا موقع تسوق إلكتروني وأردنا عرض `$0.30`. يمكننا استخدام الجمع الأحادي في الحالات الأخرى لتحويله إلى عدد:

```
let sum = 0.1 + 0.2;
alert( +sum.toFixed(2) ); // 0.3
```

يمكننا أيضا ضرب الأعداد مؤقتًا في 100 أو أي عدد أكبر لتحويل إلى أعداد صحيحة ثم إعادة قسمتها على العدد المضروب فيه. هكذا قد ترتفع نسبة الخطأ كما لو كنا نتعامل مع أعداد صحيحة لكننا سنتخلص منها بالقسمة:

```
alert( (0.1 * 10 + 0.2 * 10) / 10 ); // 0.3
alert( (0.28 * 100 + 0.14 * 100) / 100); // 0.4200000000000001
```

لذا، يقلل نهج الضرب والقسمة الخطأ لكنه لا يزيله كليًا.

قد نحاول تجنب الأجزاء كليًا في بعض الأحيان. كما لو كنا نتعامل مع متجر إلكتروني، حتى نتمكن من تخزين الأسعار بالسنت بدلًا من الدولار. لكن ماذا إن وضعنا تخفيضًا ب 30%؟ لنكن أكثر دقة، يكون تجنب الأجزاء كليًا نادرًا جدًا. قم بتقريبها فقط لتتخلص من الأجزاء الغير المرغوبة عند الحاجة.

شر البلية ما يضحك

جرب تشغيل ما يلي:

```
// مرحبًا! أنا عدد يزداد من تلقاء نفسه
alert( 9999999999999999 ); // يظهر 10000000000000000
```

هذه الحالة تعاني من المشكلة ذاتها: ضياع الدقة. يوجد 64 بِت للعدد، يمكن استخدام 52 منها لتخزين العدد، لكنها غير كافية. لذا يظهر الرقم الأخير.

لا تعرض JavaScript خطأ في مثل هذه الحالات فهي تقوم بأفضل ما لديها لملائمة العدد للصيغة المطلوبة، لكن، لسوء الحظ هذه الصيغة ليست كبيرة كفاية.

صفران

نتيجة أخرى سلبية للتمثيل الداخلي للأعداد هو وجود صفرين 0 و ‎-0. ذلك لأن الإشارة تُمَثَّل ببِت مستقل، لذا فيمكن لأي عدد أن يكون سالبًا أو موجبًا بما في ذلك الصفر.

لا يكون هذا الفرق ملحوظًا في أغلب الحالات، لأن المعاملات مُعَدَّة لتعاملهما كعدد واحد.

5.2.5 الفحص: isFinite و isNaN

هل تذكر القيم العددية الخاصة التالية؟

- `Infinity` (و `-Infinity`) هي قيمة عددية خاصة تكون أكبر أو أصغر من أي شيء.
- `NaN` تُمَثِّل وجود خطأ (ليس عددًا Not a Number).

تنتمي هذه القيم إلى النوع `number`، لكنها ليست أعداد، لذا يوجد توابع خاصة لفحصها:

- `isNaN(value)` يُحوِّل المُعامل إلى عدد ثم يفحص ما إن كان `NaN`:

```
alert( isNaN(NaN) ); // true
alert( isNaN("str") ); // true
```

لكن هل نحتاج لهذا التابع؟ أليس من الممكن استخدام الموازنة فقط `=== NaN`؟ الإجابة للأسف هي لا. القيمة `NaN` هي فريدة ولا يمكن أن تساوي أي شيء، حتى نفسها:

```
alert( NaN === NaN ); // false
```

- `isFinite(value)` يُحوِّل مُعامله إلى عدد ويُرجِع القيمة `true` إن كان عددًا عاديًا، أي لا يكون `NaN` أو `Infinity` أو `-Infinity`:

```
alert( isFinite("15") ); // true
alert( isFinite("str") ); // false, NaN
alert( isFinite(Infinity) ); // false, Infinity
```

تستخدم `isFinite` أحيانًا للتحقق ما إن كان النص عددًا عاديًا:

```
let num = +prompt("Enter a number", '');

// سيكون صحيحًا إلا إن أدخلت Infinity أو -Infinity أو قيمة غير عددية
alert( isFinite(num) );
```

يرجى ملاحظة أن الفراغ أو المسافة الواحدة تُعامل معاملة الصفر `0` في جميع التوابع العددية بما فيها `isFinite`.

الموازنة باستخدام Object.is

يوجد تابع خاص مدمج في اللغة يدعى Object.is يوازن القيم كما === لكنه أكثر موثوقية لسببين:

1. أنه يعمل مع NaN: أي `Object.is(NaN, NaN) === true` وهذا أمر جيد.
2. القيمتان 0 و ‎-0 مختلفتان: `Object.is(0, -0) === false`، الأمر صحيح تقنيًا، لأن العدد لديه إشارة داخليًا مما يجعل القيم مختلفة حتى لو كانت باقي الخانات أصفارًا.

يكون التابع `Object.is(a, b)` نفس `a === b` في باقي الحالات.

تُستخدَم طريقة الموازنة هذه غالبًا في توصيف JavaScript. عندما تحتاج خوارزمية لموازنة كون قيمتين متطابقتان تمامًا فإنها تستخدم `Object.is` (تُسَمَّى داخليًا القيمة ذاتها "SameValue").

5.2.6 parseInt و parseFloat

التحويل العددي باستخدام الجمع + أو `Number()` يعد صارمًا. إن لم تكن القيمة عددًا فإنها تفشل:

```
alert( +"100px" ); // NaN
```

الاستثناء الوحيد هو المسافات الفارغة ببداية أو نهاية النص، إذ يتم تجاهلها.

لكن، يوجد لدينا في الواقع قيمًا بالوحدات، مثل `"100px"` أو `"12pt"` في CSS. في العديد من الدول أيضا، يُلحَق رمز العملة بالقيمة، فمثًلا، لدينا `"19€"` ونريد استخراج قيمة عددية من ذلك. هذا ما يقوم به التابعان `parseInt` و `parseFloat`، إذ يقرآن العدد من النص المعطى حتى تعجزان على ذلك فتتوقف العملية. في حال وجود خطأ، يعيدان العدد المُجَمَّع. فيعيد التابع `parseInt` عددًا صحيحًا، بينما يعيد التابع `parseFloat` عددًا عشريًا:

```
alert( parseInt('100px') ); // 100
alert( parseFloat('12.5em') ); // 12.5

// يُرجَع العدد الصحيح فقط
alert( parseInt('12.3') ); // 12

// تُوقِف النقطة الثانية عملية التحليل
alert( parseFloat('12.3.4') ); // 12.3
```

يوجد بعض الحالات يرجع فيها التابع `parseInt` أو `parseFloat` القيمة `NaN` وذلك عندما لا يوجد أي رقم لإرجاعه. ففي المثال التالي، يوقف الحرف الأول العملية:

```
alert( parseInt('a123') ); // NaN
```

المُعامِل الثاني للتابع parseInt(str, radix)

لدى التابع `parseInt()` مُعامِل ثانٍ اختياري. والذي يحدد نوع النظام العددي، لذا يمكن للتابع `parseInt` تحويل النصوص ذات الأعداد الست عشرية، الثنائية وهكذا:

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255

alert( parseInt('2n9c', 36) ); // 123456
```

5.2.7 دوال رياضية أخرى

تحتوي JavaScript على الكائن المُدمَج Math الذي يحتوي على مكتبة صغيرة بالدوال والثـوابت الرياضـية. إليك بعض الأمثلة:

- `Math.random()`: تُرجِع عددًا عشوائيًا من 0 إلى 1 (لا تتضمن 1):

```
alert( Math.random() );  // 0.1234567894322
alert( Math.random() );  // 0.5435252343232
alert( Math.random() );  // ... (أي رقم عشوائي)
```

- `Math.max(a, b, c...) / Math.min(a, b, c...)`: تُرجِع القيمة الأكـبر أو الأصـغر من المُعامِلات:

```
alert( Math.max(3, 5, -10, 0, 1) ); // 5
alert( Math.min(1, 2) ); // 1
```

- `Math.pow(n, power)`: تُرجع العدد n مرفوعًا إلى الأُس المُعطى:

```
alert( Math.pow(2, 10) ); // 2^10 = 1024
```

يوجد المزيد من الدوال والثوابت في الكائن `Math`، بما فيها علم المُثَلَّثات، والتي يمكنك إيجادها في توثيق الكائن Math في موسوعة حسوب.

5.2.8 الخلاصة

لكتابة أعداد كبيرة:

- أضِف `"e"` مع عدد الأصفار الخاصة بالعدد المطلوب، مثل: `123e6` هو `123` مع 6 أصفار.
- قيمة سالبة بعد `"e"` تقسم العدد على 1 مع عدد الأصفار المُعطى.

لِأنظمة العد المختلفة:

- يمكن كتابة الأعداد مباشرة بالنظام الست عشري (`0x`)، أو الثُماني (`0o`)، أو الثُنائي (`0b`).
- `parseInt(str, base)` تحوِّل النص `str` إلى عدد صحيح بالنظام العددي المُعطى `base`، و `2 ≤ base ≤ 36`.
- `num.toString(base)` تحوِّل العدد إلى نص بالنظام العددي المُعطى `base`.

لتحويل القيم مثل `12pt` and `100px` إلى عدد:

- استخدم `parseInt` أو `parseFloat` لتحويل سلس، والتي تقرأ العدد من نص وتُرجِع القيمة التي استطاعت قراءتها قبل حصول أي خطأ.

للأجزاء:

- التقريب باستخدام `Math.floor`، أو `Math.ceil`، أو `Math.trunc`، أو `Math.round` أو `num.toFixed(precision)`.
- تذكر وجود ضياع في دقة الجزء العشري عند التعامل مع الكسور.
- للمزيد من الدوال الرياضية:
- اطلع على الكائن Math عندما تحتاج ذلك، هذه المكتبة صغيرة جدًا، لكنها تغطي الاحتياجات الأساسية.

5.2.9 المهام

ا. جمع الأعداد من الزائر

الأهمية: ★★★★★

أنشِئ سكربت يتيح للمستخدم إدخال رقمين ثم أعرض مجموعهما.

الحل:

```
let a = +prompt("The first number?", "");
let b = +prompt("The second number?", "");

alert( a + b );
```

لاحظ عامل الجمع الأحادي `+` قبل `prompt`. يحوِّل القيم إلى أعداد. وإلا فإن `a` و `b` ستكون نصوصًا وسيكون مجموعهما بدمجهما: `"1" + "2" = "12"`.

ب. لماذا 6.35.toFixed(1) == 6.3؟

الأهمية: ★★★★☆

تُدَوِّر كلًا من `Math.round` و `toFixed` العدد إلى أقرب عدد له وفقًا للتوثيق: الأجزاء من `0..4` تُدَوَّر للأسفل، بينما الأجزاء `5..9` تُدَوَّر للأعلى.

مثلًا:

```
alert( 1.35.toFixed(1) ); // 1.4
```

في المثال المشابه أدناه، لِمَ تُدَوَّر `6.35` إلى `6.3`، وليس `6.4`؟

```
alert( 6.35.toFixed(1) ); // 6.3
```

كيف نُدَوِّر `6.35` بالطريقة الصحيحة؟

الحل:

الجزء `6.35` هو عبارة عن عدد غير منتهي في الصيغة الثنائية. وكجميع الحالات المشابهة، يُخَزَّن مع ضياع في الدقة. لنرَ:

```
alert( 6.35.toFixed(20) ); // 6.34999999999999964473
```

قد يتسبب ضياع الدقة في زيادة أو نقصان أي عدد. يكون العدد في هذه الحالة أقل بقليل من قيمته الفعلية، ولهذا يُدَوَّر للأسفل. ماذا عن العدد `1.35`؟

```
alert( 1.35.toFixed(20) ); // 1.35000000000000008882
```

جعل ضياع الدقة هذا الرقم أكبر بقليل مما هو عليه مما تسبب في تقريبه للأعلى.

كيف يمكننا حل مشكلة تقريب العدد `6.35` حتى يُدَوَّر بالشكل الصحيح

يجب أن نحوله إلى عدد صحيح قبل التقريب:

```
alert( (6.35 * 10).toFixed(20) ); // 63.50000000000000000000
```

لاحظ عدم وجود أي ضياع في دقة العدد `63.5`. ذلك لأن الجزء العشري `0.5` يساوي `1/2`. يمكن تمثيل الأجزاء المقسومة على `2` تُمَثَّل بشكل صحيح في النظام الثنائي. يمكننا تقريب العدد الآن:

```
alert( Math.round(6.35 * 10) / 10); // 6.35 -> 63.5 -> 64(rounded) -> 6.4
```

ج. كرر حتى يصبح المُدخَل عددًا

الأهمية: ★★★★★

أنشِئ الدالة `readNumber` والتي تطلب من الزائر إدخال عـدد حـتى يقـوم بإدخـال قيمـة عدديـة صـحيحة. يجب أن تكون القيمة المُرجَعة عددًا.

يمكن للزائر إيقاف العملية بإدخال سطر فارغ أو الضغط على "CANCEL". يجب أن تُرجِع الدالـة `null` في هذه الحالة.

الحل:

```
function readNumber() {
  let num;

  do {
    num = prompt("Enter a number please?", 0);
  } while ( !isFinite(num) );

  if (num === null || num === '') return null;

  return +num;
}

alert(`Read: ${readNumber()}`);
```

طريقة الحل معقدة قليلًا حتى نتمكن من معالجة حالات الأسطر الفارغة/`null`. لـذا فـإن الشـيفرة تسـتقبل المدخلات حتى تكون عددًا عاديًا. تُطَبِّق كلًا من (cancel) `null` والسطر الفـارغ شـروط الأعـداد كونهـا تسـاوي القيمة العددية 0.

بعد توقف الشيفرة يجب معاملة `null` والأسـطر الفارغـة بطريقـة خاصـة (إرجـاع `null`). لأن تحويلهـا إلى أعداد يُرجِع 0.

د. حلقة غير منتهية أحيانًا

الأهمية: ★★★★☆

الحلقة التالية غير منتهية، ولا تتوقف أبدًا. لماذا؟

```
let i = 0;
while (i != 10) {
  i += 0.2;
}
```

الحل:

ذلك لأن i لن يساوي 10 أبدًا. نفذ الشيفرة التالية لرؤية قيم i:

```
let i = 0;
while (i < 11) {
  i += 0.2;
  if (i > 9.8 && i < 10.2) alert( i );
}
```

لا توجد قيمة تساوي 10 تمامًا. تحدث مثل هذه الأمور بسبب ضياع الدقة عند إضافة الأجزاء مثل 0.2. الخلاصة، تجنب التحقق من المساواة عند التعامل مع الأجزاء العشرية.

ه. رقم عشوائي من العدد الأدنى إلى الأقصى

الأهمية: ★★☆☆☆

تُنشِئ الدالة Math.random() المُضَمَّنَة في اللغة قيمة عشوائية بين 0 و 1 (ليس بما في ذلك 1). اكتب الدالة random(min, max) لتوليد عدد عشري عشوائي من min إلى max (بما لا يتضمن max).

أمثلة عن عملها:

```
alert( random(1, 5) ); // 1.2345623452
alert( random(1, 5) ); // 3.7894332423
alert( random(1, 5) ); // 4.3435234525
```

الحل:

نريد تعيين جميع القيم من الفترة 0…1 إلى القيم من min إلى max. يمكن القيام بذلك في مرحلتين:

1. إذا ضربنا قيمة عشوائية من 0…1 في max-min. فإن فترة القيم الممكنة تزيد 0..1 إلى 0..max-min.
2. إذا أضفنا min الآن، تصبح الفترة من min إلى max.

الدالة:

```
function random(min, max) {
  return min + Math.random() * (max - min);
}

alert( random(1, 5) );
```

```
alert( random(1, 5) );
alert( random(1, 5) );
```

و. قيمة صحيحة عشوائية من min إلى max

الأهمية: ★★☆☆☆

أنشِئ دالة `randomInteger(min, max)` تقوم بتوليد قيمة صحيحة عشوائية من `min` إلى `max` بما في ذلك `min` و `max`.

يجب أن يظهر كل رقم من الفترة `min..max` بفرص متساوية. مثال على طريقة العمل:

```
alert( randomInteger(1, 5) ); // 1
alert( randomInteger(1, 5) ); // 3
alert( randomInteger(1, 5) ); // 5
```

يمكنك استخدام حل المثال السابق كأساس لهذه المهمة.

الحل:

الطريقة السهلة والخطأ

الحل الأسهل لكنه خطأ سيكون بتوليد قيمة من `min` إلى `max` وتقريبها:

```
function randomInteger(min, max) {
  let rand = min + Math.random() * (max - min);
  return Math.round(rand);
}

alert( randomInteger(1, 3) );
```

الدالة تعمل لكنها خطأ. احتمال ظهور القيم الطرفية `min` و `max` أقل بمرتين من باقي القيم. إن شغلنا المثال أعلاه لعدة مرات، فنرى ظهور 2 بصورة أكبر.

يحدث ذلك لأن `Math.round()` تأخذ رقما من الفترة `1..3` وتُدَوِرها كما يلي:

```
values from 1    ... to 1.4999999999  become 1
values from 1.5  ... to 2.4999999999  become 2
values from 2.5  ... to 2.9999999999  become 3
```

نلاحظ الآن أن لدى 1 قيم أقل بمرتين من 2 وكذلك 3.

الطريقة الصحيحة

يوجد العديد من الطرق الصحيحة لحل هذه المهمة. إحداها هو بتعديل حدود الفترة، وللتأكد من وجود فرص متساوية، نُوَلِّد قيمًا من 0.5 إلى 3.5، ثم إضافة الاحتمالات الممكنة للأطراف:

```
function randomInteger(min, max) {
  // التقريب الآن من (min-0.5) إلى (max+0.5)
  let rand = min - 0.5 + Math.random() * (max - min + 1);
  return Math.round(rand);
}

alert( randomInteger(1, 3) );
```

طريقة بديلة هي استخدام الدالة Math.floor لرقم عشوائي من min إلى max+1:

```
function randomInteger(min, max) {
  // التقريب الآن من min إلى (max+1)
  let rand = min + Math.random() * (max + 1 - min);
  return Math.floor(rand);
}

alert( randomInteger(1, 3) );
```

جميع الفترات أصبحت متوازنة الآن:

```
values from 1  ... to 1.9999999999  become 1
values from 2  ... to 2.9999999999  become 2
values from 3  ... to 3.9999999999  become 3
```

لدى جميع الفترات الطول ذاته مما يجعل التوزيع النهائي موحدًا.

5.3 النوع string: السلاسل النصية

تُخَزَّن النصـوص في JavaScript كسلاسـل نصـية أي سلاسـل من المحـارف (string of charecter). لا يوجد نوع بيانات مستقل للحرف أو المحرف الواحد (char). الصيغة الداخلية للنصـوص هي UTF-16 دائمًـا، ولا تكون مرتبطة بتشفير الصفحة.

5.3.1 علامات الاقتباس ""

لنراجع أنواع علامات الاقتباس (الاقتباس). يمكن تضمين النصوص إما في علامات الاقتبـاس الأحاديـة، أو الثنائية أو الفاصلة العليا المائلة:

```
let single = 'single-quoted';
let double = "double-quoted";

let backticks = `backticks`;
```

علامات الاقتباس الفردية والثنائية تكون متماثلة. أما الفاصلة العليا المائلة، فَتُتيح لنا تضمين أي تعبـير في السلسلة النصية، عبر تضمينها في `${…}`:

```
function sum(a, b) {
  return a + b;
}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

الميزة الأخرى لاستخدام الفاصلة العلوية المائلة هي إمكانية فصل السلسلة النصية إلى عدة أسطر:

```
let guestList = `Guests:
 * Ahmad
 * Pete
 * Mary
`;

// قائمة بالضيوف في أسطر منفصلة
alert(guestList);
```

يبدو الأمر طبيعيًا أليس كذلك؟ لكن علامات الاقتباس الفردية والثنائيـة لا تعمـل بهـذه الطريقـة. إن حاولنـا استخدامها في نص متعدد الأسطر، سنحصل على خطأ:

```
let guestList = "Guests: // خطأ: رمز غير متوقع
  * Ahmad";
```

أتى استخدام علامات الاقتباس الفردية والثنائية في أوقات مبكرة من إنشاء اللغة، عندما لم يُؤخَذ بالحسبان الحاجة إلى نص متعدد الأسطر. ظهرت الفاصلة العلوية المائلة مؤخرًا ولذا فإنها متعددة الاستعمالات.

تتيح لنا أيضا الفاصلة العلوية المائلة تحديد "دالة كنموذج" (template function) قبل الفاصلة العلوية المائلة الأولى. تكون الصيغة كما يلي: `` func`string` ``. تُستَدعى الدالة `func` تلقائيًا، وتستقبل النص والتعابير المُضمَّنة وتعالجها. يسمى هذا باسم "القوالب الملحقة" (tagged templates)، إذ تجعل هذه الميزة من السهل تضمين قوالب مخصصة، لكنها تستخدم بشكل نادر عمليًا. يمكنك قراءة المزيد عنها في هذا الدليل.

5.3.2 الرموز الخاصة

ما زال بالإمكان كتابة نصوص متعددة الأسطر باستخدام علامات الاقتباس الأحادية والثنائية باستخدام ما يسمى باسم "محرف سطر جديد" (newline character)، والذي يُكتَب `\n`، ويرمز لسطر جديد:

```
let guestList = "Guests:\n * Ahmad\n * Pete\n * Mary";

alert(guestList); // قائمة متعددة الأسطر بالضيوف
```

مثلًا، السطرين التاليين متماثلان، لكنهما مكتوبين بطريقة مختلفة:

```
// "سطران باستخدام "رمز السطر الجديد
let str1 = "Hello\nWorld";

// سطران باستخدام سطر جديد عادي والفواصل العليا المائلة
let str2 = `Hello
World`;

alert(str1 == str2); // true
```

يوجد رموز خاصة أخرى أقل انتشارًا.

هذه القائمة كاملة:

المحرف	الوصف
`\n`	محرف سطر جديد (Line Feed).
`\r`	محرف العودة إلى بداية السطر (Carriage Return)، ولا يستخدم بمفرده. تستخدم ملفات ويندوز النصية

المحرف	الوصف
	تركيبة من رمزين `\r\n` لتمثيل سطر جديد.
`\"` , `\'`	علامة اقتباس مزدوجة ومفردة.
`\\`	شرطة مائلة خلفية
`\t`	مسافة جدولة "Tab"
`\b`, `\f`, `\v`	فراغ خلفي (backspace)، محرف الانتقال إلى صفحة جديد (Form Feed)، مسافة جدولة أفقية (Vertical Tab) على التوالي – تُستعمَل للتوافق، ولم تعد مستخدمة.
`\xXX`	صيغة رمز يونيكود مع عدد ست عشري مُعطى `XX`، مثال: `'\x7A'` هي نفسها `'z'`.
`\uXXXX`	صيغة رمز يونيكود مع عدد ست عشري `XXXX` في تشفير UTF-16، مثلًا، `\u00A9` – هو اليونيكود لرمز حقوق النسخ ©. يجب أن يكون مكون من 6 خانات ست عشرية.
`\u{X…XXXXXX}`	(1 إلى 6 أحرف ست عشرية) رمز يونيكود مع تشفير UTF-32 المعطى. تُشَفَّر بعض الرموز الخاصة برمزي يونيكود، فتأخذ 4 بايتات، ويمكننا هكذا إدخال شيفرات طويلة.

أمثلة باستخدام حروف يونيكود:

```
alert( "\u00A9" ); // ©

// رمز نادر من الهيروغليفية الصينية (يونيكود طويل)
alert( "\u{20331}" ); // 佫

// رمز وجه مبتسم (يونيكود طويل آخر)
alert( "\u{1F60D}" ); //
```

لاحظ بدء جميع الرموز الخاصة بشرطة مائلة خلفية `\`. تدعى أيضا باسم "محرف تهريب (escape character)". يمكننا استخدامها أيضًا إن أردنا تضمين علامة اقتباس في النص: مثلًا:

```
alert( 'I\'m the Walrus!' ); // I'm the Walrus!
```

يجب إلحاق علامة الاقتباس الداخلية بالشرطة المائلة الخلفية `\'`، وإلا فستُعتَبَر نهاية السلسلة النصية.

لاحظ أن الشرطة المائلة الخلفية `\` تعمل من أجل تصحيح قراءة السلسلة النصية بواسطة JavaScript. ومن ثم

تختفي، لذا فإن النص في الذاكرة لا يحتوي على \. يمكننا رؤية ذلك بوضوح باستخدام alert على المثال السابق.

يجب استخدام محرف التهريب في حالة استخدام علامة الاقتباس المحيطة بالنص نفسها، لذا فإن الحل الأمثل هو استخدام علامات اقتباس مزدوجة أو فواصل عليا مائلة في مثل هذه الحالة:

```
alert( `I'm the Walrus!` ); // I'm the Walrus!
```

لكن ماذا إن أردنا عرض شرطة مائلة خلفية ضمن النص؟ يمكن ذلك، لكننا نحتاج إلى تكرارها هكذا \\:

```
alert( `The backslash: \\` ); // The backslash: \
```

5.3.3 طول النص

تحمل الخاصية length طول النص:

```
alert( `My\n`.length ); // 3
```

لاحظ أن \n هو رمز خاص، لذا يكون طول السلسلة الفعلي هو 3.

length هي خاصية

يُخطِئ بعض الأشخاص ذوي الخلفيات بلغات برمجية أخرى ويستدعون str.length() بدلًا من استدعاء str.length فقط، لذا لا يعمل هذا التابع لعدم وجوده. فلاحظ أن str.length هي خاصية عددية، وليس تابعًا ولا حاجة لوضع قوسين بعدها.

5.3.4 الوصول إلى محارف سلسلة

للحصول على حرف في مكان معين من السلسلة النصية pos، استخدم الأقواس المعقوفة [pos] أو استدعِ التابع str.charAt(pos). يبدأ أول حرف في الموضع رقم صفر:

```
let str = `Hello`;

// الحرف الأول
alert( str[0] ); // H
alert( str.charAt(0) ); // H

// الحرف الأخير
alert( str[str.length - 1] ); // o
```

الأقواس المعقوفة هي طريقة جديدة للحصول على حرف، بينما التابع `charAt` موجود لأسباب تاريخية. الاختلاف الوحيد بينهما هو إن لم تجد الأقواس المربعة `[]` الحرف تُرجِع القيمة `undefined` بينما يُرجِع `charAt` نصًا فارغًا:

```
let str = `Hello`;

alert( str[1000] ); // undefined
alert( str.charAt(1000) ); // '' (سلسلة نصية فارغ)
```

يمكننا أيضا التنقل خلال جميع محارف سلسلة باستخدام `for..of`:

```
for (let char of "Hello") {
  alert(char); // H,e,l,l,o
}
```

5.3.5 النصوص ثابتة

لا يمكن تغيير النصوص في JavaScript، فمن المستحيل تغيير حرف داخل سلسلة نصية فقط. لنجرب الأمر للتأكد من أنه لن يعمل:

```
let str = 'Hi';

str[0] = 'h'; // خطأ
alert( str[0] ); // لا تعمل
```

الطريقة المعتادة هي إنشاء نص جديد وإسناده للمتغير `str` بدلًا من النص السابق. مثلًا:

```
let str = 'Hi';

str = 'h' + str[1]; // تستبدل كامل السلسلة النصية

alert( str ); // hi
```

سنرى المزيد من الأمثلة عن ذلك في الأجزاء التالية.

5.3.6 تغيير حالة الأحرف الأجنبية

يقوم التابع toLowerCase() والتابع toUpperCase() بِتغيير حالة الأحرف الأجنبية:

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
```

```
alert( 'Interface'.toLowerCase() ); // interface
```

أو إن أردنا بتغيير حالة حرف واحد فقط:

```
alert( 'Interface'[0].toLowerCase() ); // 'i'
```

5.3.7 البحث عن جزء من النص

يوجد العديد من الطرق للبحث عن جزء من النص ضمن السلسلة النصية.

أ. str.indexOf

التابع الأول هو str.indexOf(substr, pos).

يبحث التابع عن `substr` في `str` بدءًا من الموضع المحدد `pos`، ثم يُرجِع الموضع الذي تطابق مع النص أو يُرجِع `-1` إن لم تعثر على تطابق. مثلًا:

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0
alert( str.indexOf('widget') ); // -1

alert( str.indexOf("id") ); // 1
```

لم تعثر على شيء في حالة البحث الثانية، إذ البحث هنا حساس لحالة الأحرف.

يتيح لنا المُعامِل الثاني الاختياري البحث من الموضع المُعطَى. مثلًا في الحالة الثالثة، أول ظهور ل `"id"` هو في الموضع `1`. لِلبحث عن الظهور التالي له نبدأ البحث من الموضع `2`:

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12
```

إن كنت مهتمًا بجميع المواضع التي يظهر فيها نص معين، يمكنك استخدام `indexOf` في حلقة. يتم كل استدعاء جديد من الموضِع التالي لِلموضع السابق الذي تطابق مع النص:

```
let str = 'As sly as a fox, as strong as an ox';

let target = 'as'; // لنبحث عنها

```

```
let pos = 0;
while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert( `Found at ${foundPos}` );
  pos = foundPos + 1; // استمر بالبحث من الموضع التالي
}
```

يمكن تقصير الخوارزمية:

```
let str = "As sly as a fox, as strong as an ox";
let target = "as";

let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
  alert( pos );
}
```

str.lastIndexOf(substr, position)

يوجد أيضًا تابع مشابه str.lastIndexOf(substr, position) والذي يبدأ البحث من نهاية السلسلة النصية حتى بدايتها. أي أنه يعيد موضع ظهور النص المبحوث عنه انطلاقًا من نهاية السلسلة.

يوجد خلل طفيف عند استخدام `indexOf` في `if`. فلا يمكن وضعها بداخل `if` بالطريقة التالية:

```
let str = "Widget with id";

if (str.indexOf("Widget")) {
    alert("We found it"); // لا تعمل!
}
```

لا يتحقق الشرط في المثال السابق لأن `str.indexOf("Widget")` يُرجِع `0` (ما يعني وجود تطابق في الموضع الأول) رغم عثور التابع على الكلمة، لكن `if` تعد القيمة `0` على أنها `false`. لذا يجب أن نفحص عدم وجود القيمة `-1` هكذا:

```
let str = "Widget with id";

if (str.indexOf("Widget") != -1) {
```

```
    alert("We found it"); // تعمل الآن
}
```

خدعة NOT على مستوى البِت

إحدى الخدع القديمة هي العامل الثنائي NOT `~` الذي يعمل على مستوى البِت. فهو يُحَوِّل العدد إلى عدد صحيح بصيغة 32-بِت (يحذف الجزء العشري إن وجد) ثم يُحوِّل جميع البتات إلى تمثيلها الثنائي. عمليًا، يعني ذلك شيئًا بسيطًا: بالنسبة للأعداد الصحيحة بصيغة 32-بِت `~n` تساوي `-(n+1)`. مثلًا:

```
alert( ~2 ); // -3 == -(2+1)
alert( ~1 ); // -2 == -(1+1)
alert( ~0 ); // -1 == -(0+1)
alert( ~-1 ); // 0 == -(-1+1)
```

كما نرى، يكون `~n` صفرًا فقط عندما تكون `n == -1` (وذلك لأي عدد صحيح `n` ذي إشارة). لذا، يكون ناتج الفحص `if ( ~str.indexOf("...") )` صحيحًا إذا كانت نتيجة `indexOf` لا تساوي `-1`. بمعنى آخر تكون القيمة `true` إذا وُجِد تطابق.

الآن، يمكن استخدام هذه الحيلة لتقصير الفحص باستخدام `indexOf`:

```
let str = "Widget";

if (~str.indexOf("Widget")) {
  alert( 'Found it!' ); // تعمل
}
```

لا يكون من المستحسن غالبًا استخدام ميزات اللغة بطريقة غير واضحة، لكن هذه الحيلة تُستخدم بكثرة في الشيفرات القديمة، لذا يجب أن نفهمها.

تذكر أن الشرط `if (~str.indexOf(...))` يعمل بالصيغة "إن وُجِد".

حتى نكون دقيقين، عندما تُحَوَّل الأرقام إلى صيغة 32-بِت باستخدام المعامل `~` يوجد أعداد أخرى تُعطي القيمة `0`، أصغر هذه الأعداد هي `~4294967295 == 0` ما يجعل هذا الفحص صحيحًا في حال النصوص القصيرة فقط.

لا نجد هذه الخدعة حاليًا سوى في الشيفرات القديمة، وذلك لأن JavaScript وفرت التابع `.includes` (ستجدها في الأسفل).

ب. includes, startsWith, endsWith

يُرجِع التابع الأحدث str.includes(substr, pos) القيمة المنطقية true أو false وفقًا لما إن كانت السلسلة النصية str تحتوي على السلسلة النصية الفرعية substr. هذه هي الطريقة الصحيحة في حال أردنا التأكد من وجود تطابق جزء من سلسلة ضمن سلسلة أخرى، ولا يهمنا موضعه:

```
alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false
```

المُعامِل الثاني الاختياري للتابع str.includes هو الموضع المراد بدء البحث منه:

```
alert( "Widget".includes("id") ); // true
alert( "Widget".includes("id", 3) ); // false
```

يعمل التابعان str.startsWith و str.endsWith بما هو واضح من مسمياتهما، "سلسلة نصية تبدأ بـ"، و "سلسلة نصية تنتهي بـ" على التوالي:

```
alert( "Widget".startsWith("Wid") ); // true,
alert( "Widget".endsWith("get") ); // true,
```

5.3.8 جلب جزء من نص

يوجد 3 توابع في JavaScript لجلب جزء من سلسلة نصية هي: substring، وsubstr، وslice.

ا. str.slice(start [, end])

يُرجِع جزءًا من النص بدءًا من الموضع start وحتى الموضع end (بما لا يتضمن end). مثلًا:

```
let str = "stringify";
alert( str.slice(0, 5) ); // 'strin'
alert( str.slice(0, 1) ); // 's'
```

إن لم يكن هناك مُعامل ثانٍ، فسيقتطع التابعslice الجزء المحدد من الموضع start وحتى نهاية النص:

```
let str = "stringify";
alert( str.slice(2) ); // ringify
```

يمكن أيضًا استخدام عدد سالبًا مع start أو end، وذلك يعني أن الموضع يُحسَب بدءًا من نهاية السلسلة النصية:

```
let str = "stringify";

// تبدأ من الموضع الرابع من اليمين، إلى الموضع الأول من اليمين
alert( str.slice(-4, -1) ); // gif
```

ب. str.substring(start [, end])

يُرجِع هذا التابع جزءًا من النص الواقع بين الموضع start والموضع end. يشبه هذا التابع تقريبًا التابع slice، لكنه يسمح بكون المعامل start أكبر من end. مثلًا:

```
let str = "stringify";

// الأمرين التاليين متماثلين بالنسبة لـ substring
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"

// لكن ليس مع slice
alert( str.slice(2, 6) ); // "ring" (نفس النتيجة السابقة)
alert( str.slice(6, 2) ); // "" (نص فارغ)
```

بعكس slice، القيم السالبة غير مدعومة ضمن المعاملات، وتقيَّم إلى 0 إن مُرِّرت إليه.

ج. str.substr(start [, length])

يُرجِع هذا التابع الجزء المطلوب من النص، بدءًا من start وبالطول length المُعطى. بعكس التوابع السابقة، يتيح لنا هذا التابع تحديد طول النص المطلوب بدلًا من موضع نهايته:

```
let str = "stringify";

// خذ 4 أحرف من الموضع 2
alert( str.substr(2, 4) ); // ring
```

يمكن أن يكون المُعامِل الأول سالبًا لتحديد الموضع بدءًا من النهاية:

```
let str = "stringify";

// حرفين ابتداءً من الموضع الرابع
alert( str.substr(-4, 2) ); // gi
```

لِنُلَخِّص هذه التوابع لتجنب الخلط بينها:

التابع	يقتطع …	المواضع السالبة
`slice(start, end)`	من الموضع `start` إلى الموضع `end` (بما لا يتضمن `end`)	مسموحة لكلا المعاملين
`substring(start, end)`	ما بين الموضع `start` والموضع `end`	غير مسموحة وتصبح 0
`substr(start, length)`	أرجع الأحرف بطول `length` بدءًا من `start`	مسموحة للمعامل `start`

أيهما تختار؟

يمكن لجميع التوابع تنفيذ الغرض المطلوب. لدى التابع `substr` قصور بسيط رسميًا: فهو غير ذكورة في توثيق JavaScript الرسمي، بل في Annex B والذي يغطي ميزات مدعومة في المتصفحات فقط لأسباب تاريخية، لذا فإن أي بيئة لا تعمل على المتصفح ستفشل في دعم هذا التابع، لكنه يعمل عمليًا في كل مكان.

ما بين الخيارين الآخرين، `slice` هو أكثر مرونة، فهو يسمح بتمرير مُعامِلات سالبة كما أنه أقصر في الكتابة. لذا، من الكافِ تذكر `slice` فقط من هذه التوابع الثلاث.

5.3.9 موازنة النصوص

توازن السلاسل النصية حرفًا حرفًا بترتيب أبجدي كما عرفنا في فصل معاملات الموازنة.

بالرغم من ذلك، يوجد بعض الحالات الشاذة.

1- الحرف الأجنبي الصغير دائما أكبر من الحرف الكبير:

```
alert( 'a' > 'Z' ); // true
```

2- الأحرف المُشَكَلَة خارج النطاق:

```
alert( 'Österreich' > 'Zealand' ); // true
alert( 'تُونس' > 'سوريا' ); // false
```

قد يقود ذلك إلى نتائج غريبة إن رتبنا مثلًا بين أسماء بلدان، فيتوقع الناس دائمًا أن `Zealand` تأتي بعد `Österreich` في القائمة وأن تونس تأتي قبل سوريا وهكذا. لفهم ما يحدث، لنراجع تمثيل النصوص الداخلي في JavaScript.

جميع النصوص مشفرة باستخدام UTF-16. يعني أن: لكل حرف رمز عددي مقابل له. يوجد دوال خاصة تسمح بالحصول على الحرف من رمزه والعكس.

أ. str.codePointAt(pos)

يُرجِع هذا التابع الرمز العددي الخاص بالحرف المعطى في الموضع pos:

```
// لدى الأحرف المختلفة في الحالة رموز مختلفة
alert( "z".codePointAt(0) ); // 122
alert( "Z".codePointAt(0) ); // 90
```

ب. String.fromCodePoint(code)

يُنشِئ حرفًا من رمزه العددي code:

```
alert( String.fromCodePoint(90) ); // Z
```

يمكننا إضافة حرف يونيكود باستخدام رمزه بواسطة \u متبوعة بالعدد الست عشري:

```
alert( '\u005a' ); // Z
```

يُمثَّل العدد العشري 90 بالعدد 5a في النظام الست عشري.

لنرَ الآن الأحرف ذات الرموز 65..220 (الأحرف اللاتينية وأشياء إضافية) عبر إنشاء نصوص منها:

```
let str = '';

for (let i = 65; i <= 220; i++) {
  str += String.fromCodePoint(i);
}
alert( str );
// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
// ¡¢£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜ
```

تبدأ الأحرف الكبيرة كما ترى، ثم أحرف خاصة، ثم الأحرف الصغيرة، ثم Ö بالقرب من نهاية المخرجات.

يصبح الآن واضحًا لم a > Z. أي تُوازَن الأحرف بواسطة قيمها العددية. فالرمز العددي الأكبر يعني أن الحرف أكبر. الرمز للحرف a هو 97 وهو أكبر من الرمز العددي للحرف Z الذي هو 90.

- تأتي الأحرف الصغيرة بعد الأحرف الكبيرة دائمًا لأن رموزها العددية دائمًا أكبر.
- تكون بعض الأحرف مثل Ö بعيدة عن الأحرف الهجائية. هنا، قيمة الحرف هذا أكبر من أي حرف بين a و Z.

ج. موازنات صحيحة

الخوارزمية الصحيحة لموازنة النصوص أكثر تعقيدًا مما يبدو عليه الأمر، لأن الأحرف تختلف باختلاف اللغات. لذا، يحتاج المتصفح لمعرفة اللغة لموازنة نصوصها موازنةً صحيحةً.

لحسن الحظ، تدعم جميع المتصفحات الحديثة المعيار العالمي ECMA 402 ‏(IE10- الذي يتطلب المكتبة الإضافية Intl.JS)، إذ يوفر تابعًا خاصًا لموازنة النصوص بلغات متعددة، وفقًا لقواعدها.

يُرجِع استدعاء التابع str.localeCompare(str2) عددًا يحدد ما إن كان النص `str` أصغر، أو يساوي، أو أكبر من النص `str2` وفقًا لقواعد اللغة المحلية:

- يُرجِع قيمة سالبة إن كان `str` أصغر من `str2`.
- يُرجِع قيمة موجبة إن كان `str` أكبر من `str2`.
- يُرجِع `0` إن كانا متساويين.

إليك المثال التالي:

```
alert( 'Österreich'.localeCompare('Zealand') ); // -1
```

في الحقيقة، لهذه الدالة مُعامِلين إضافيين كما في توثيقها على موسوعة حسوب، إذ يسمح هذان المُعاملان بتحديد اللغة (تؤخذ من بيئة العمل تلقائيًا، ويعتمد ترتيب الأحرف على اللغة) بالإضافة إلى إعداد قواعد أخرى مثل الحساسية تجاه حالة الأحرف، أو ما إن كان يجب معاملة `"a"` و `"á"` بالطريقة نفسها ...الخ.

5.3.10 ما خلف الستار، يونيكود

معلومات متقدمة

يتعمق الجزء التالي في ما يقبع خلف ستار النصوص التي تراها، وهذه المعلومات ستكون قَيِّمة إن كنت تخطط للتعامل مع الرموز التعبيرية، أو الأحرف الرياضية النادرة أو الهيروغليفية أو أي رموز نادرة أخرى. يمكنك تخطي هذا الجزء إن لم تكن مهتمًا به.

ا. أزواج بديلة Surrogate pairs

لكل الأحرف المستخدمة بكثرة رموز عددية (code) مؤلفة من 2-بايت. لدى أحرف اللغات الأوروبية، والأرقام، وحتى معظم الرموز الهيروغليفية تمثيل من 2-بايت. لكن، نحصل من 2-بايت على 65536 تركيبًا فقط وذلك غير كافٍ لكل الرموز (symbol) المُحتَمَلَة، لذا فإن الرموز (symbol) النادرة مرمزة بزوج من المحارف بحجم 2-بايت يسمى "أزواج بديلة" (Surrogate pairs).

طول كل رمز هو `2`:

```
// الحرف X في الرياضيات
alert( '𝒳'.length ); // 2

// وجه ضاحك بدموع
alert( '  '.length ); // 2

// حرف صيني هيروغليفي نادر
alert( '𩷶'.length ); // 2
```

لاحظ أن الأزواج البديلة لم تكن موجودة منذ إنشاء JavaScript، ولذا لا تعالج بشكل صحيح بواسطة اللغة. في النصوص السابقة لدينا رمز واحد فقط، لكن طول النص `length` ظهر على أنه `2`.

التابعان `String.fromCodePoint` و `str.codePointAt` نادران وقليلا الاستخدام، إذ يتعاملان مع الأزواج البديلة بصحة. وقد ظهرت مؤخرًا في اللغة. في السابق كان هنالك التابعان `String.fromCharCode` و `str.charCodeAt` فقط. هذان التابعان يشبهان `fromCodePoint` و `codePointAt`، لكنهما لا يتعاملان مع الأزواج البديلة.

قد يكون الحصول على رمز (symbol) واحد صعبًا، لأن الأزواج البديلة تُعامَل معاملة حرفين:

```
alert( '𝒳'[0] ); // رموز غريبة
alert( '𝒳'[1] ); // أجزاء من الزوج البديل
```

لاحظ أن أجزاء الزوج البديل لا تحمل أي معنى إذا كانت منفصلة عن بعضها البعض. لذا فإن ما يعرضه مر `alert` في الأعلى هو شيء غير مفيد.

يمكن تَوَقُّع الأزواج البديلة عمليًا بواسطة رموزها: إن كان الرمز العددي لحرف يقع في المدى `0xd800..0xdbff`، فإنه الجزء الأول من الزوج البديل. أما الجزء الثاني فيجب أن يكون في المدى `0xdc00..0xdfff`. هذا المدى محجوز للأزواج البديلة وفقًا للمعايير المتبعة.

وفقًا للحالة السابقة، سنستعمل التابع `charCodeAt` الذي لا يتعامل مع الأزواج البديلة، لذا فإنه يُرجِع أجزاء الرمز:

```
alert( '𝒳'.charCodeAt(0).toString(16) ); // d835
alert( '𝒳'.charCodeAt(1).toString(16) ); // dcb3
```

نجد أن العدد الست عشري الأول d835 يقع بين 0xd800 و 0xdbff، والعدد الست عشري الثاني يقع بين 0xdc00 و 0xdfff وهذا يؤكد أنها من الأزواج البديلة.

ستجد المزيد من الطرق للتعامل مع الأزواج البديلة لاحقًا في فصل المُكـرِّرات. يوجـد أيضًـا مكـاتب خاصـة لذلك، لكن لا يوجد شيء شهير محدد لِاقتراحه هنا.

ب. علامات التشكيل وتوحيد الترميز

يوجد حروف مركبة في الكثير من اللغات والتي تتكون من الحرف الرئيسي مع علامة فوقه/تحته. مثلًا، يمكن للحرف a أن يكون أساسًا للأحرف التالية: àáâäãåā. لدى معظم الحروف المركبة رمزهـا الخـاص بهـا في جـدول UTF-16. لكن ليس جميعها، وذلك لوجود الكثير من الاحتمالات.

لدعم التراكيـب الأساسية، تتيح لنا UTF-16 استخدام العديد من حـروف يونيكـود: الحـرف الرئيسـي متبوعًـا بعلامة أو أكثر لتشكيله. مثلًا، إن كان لدينا S متبوعًا بالرمز الخـاص "النقطـة العلويـة" (الـتي رمزهـا \u0307). فسيُعرَض ك Ṡ.

```
alert( 'S\u0307' ); // Ṡ
```

إن احتجنا إلى رمز آخر فوق أو تحت الحرف فلا مشكلة، أضِف العلامة المطلوبة فقط. مثلًا، إن ألحقنـا حـرف "نقطة بالأسفل" (رمزها \u0323)، فسنحصل على "S بنقاط فوقه وتحته"، Ṩ:

```
alert( 'S\u0307\u0323' ); // Ṩ
```

هذا يوفر مرونة كبيرة، لكن مشكلة كبيرة أيضًا: قد يظهر حرفان بالشكل ذاتـه، لكن يمثلان بـتراكيب يونيكـود مختلفة. مثلًا:

```
// S + نقطة في الأعلى + نقطة في الأسفل
let s1 = 'S\u0307\u0323'; // Ṩ

// S + نقطة في الأسفل + نقطة في الأعلى
let s2 = 'S\u0323\u0307'; // Ṩ,

alert( `s1: ${s1}, s2: ${s2}` );

alert( s1 == s2 ); // خطأ بالرغم من أن الحرفين متساويان ظاهريًا
```

لحل ذلك، يوجد خوارزمية تدعى "توحيد ترميز اليونيكود" (unicode normalization) والتي تُعيد كل نص إلى الصيغة الطبيعية المستقلة له. هذه الخوارزمية مُضَمَّنة في التابع str.normalize().

```
alert( "S\u0307\u0323".normalize() == "S\u0323\u0307".normalize() );
// true
```

من المضحك في حالتنا أن `normalize()` تجمع سلسلة من 3 أحرف مع بعضها بعضًا إلى حرف واحد: `\u1e68` (الحرف S مع النقطتين).

```
alert( "S\u0307\u0323".normalize().length ); // 1

alert( "S\u0307\u0323".normalize() == "\u1e68" ); // true
```

في الواقع، هذه ليست الحالة دائمًا. وذلك لأن الرمز Ṩ متعارف بكثرة، فضَمَّنَهُ مُنشِئو UTF-16 في الجدول الرئيسي وأعطوه رمزًا خاصًا.

إن أردت تعلم المزيد عن قواعد التوحيد واختلافاتها - فستجدها في ملحق معايير اليونيكود: نماذج توحيد ترميز اليونيكود، لكن للأغراض العملية المتعارفة فالمعلومات السابقة تفي بالغرض.

5.3.11 الخلاصة

- يوجد 3 أنواع لِعلامات الاقتباس. تسمح الشرطات العلوية المائلة للنص بالتوسع لأكثر من سطر وتضمين التعبير `${…}`.
- النصوص في JavaScript مُشَفَّرة بواسطة UTF-16.
- يمكننا استخدام أحرف خاصة مثل `\n` وإدخال أحرف باستخدام رمز يونيكود الخاص بها باستخدام `\u...`.
- استخدم `[]` للحصول على حرف ضمن سلسلة نصية.
- للحصول على جزء من النص، استخدم: `slice` أو `substring`.
- للتحويل من أحرف كبيرة/صغيرة، استخدم: `toLowerCase` أو `toUpperCase`.
- للبحث عن جزء من النص، استخدم: `indexOf`، أو `includes` أو `startsWith` أو `endsWith` للفحص البسيط.
- لموازنة النصوص وفقًا للغة، استخدم: `localeCompare`، وإلا فستوازن برموز الحروف.
- يوجد الكثير من التوابع الأخرى المفيدة في النصوص:
- `str.trim()` تحذف ("تقتطع") المسافات الفارغة من بداية ونهاية النص.
- `str.repeat(n)` تُكرِّر النص n مرة.
- والمزيد، يمكن الاطلاع عليها في موسوعة حسوب.

هنالك توابع أخرى للنصوص أيضًا تعمل على البحث/الاستبدال مع التعابير النمطية (regular expressions). لكن ذلك موضوع كبير، لذا فقد شُرِحَ في فصل مستقل، التعابير النمطية.

5.3.12 تمارين

ا. حول الحرف الأول إلى حرف كبير

الأهمية: ★★★★★

اكتب دالة باسم `ucFirst(str)` تُرجِع النص `str` مع تكبير أول حرف فيه، مثلًا:

```
ucFirst("john") == "Ahmad";
```

الحل:

لا يمكننا استبدال الحرف الأول، لأن النصوص في JavaScript غير قابلة للتعديل. لكن، يمكننا إنشاء نص جديد وفقًا للنص الموجود، مع تكبير الحرف الأول:

```
let newStr = str[0].toUpperCase() + str.slice(1)
```

لكن، يوجد مشكلة صغيرة، وهي إن كان `str` فارغًا، فسيصبح `str[0]` قيمة غير معرفة `undefined`، ولأن `undefined` لا يملك الدالة `toUpperCase()` فسيظهر خطأ.

يوجد طريقتين بديلتين هنا:

1. استخدام `str.charAt(0)`، لأنها تُرجِع نصًا دائمًا (ربما نصًا فارغًا).
2. إضافة اختبار في حال كان النص فارغًا.

هنا الخيار الثاني:

```
function ucFirst(str) {
  if (!str) return str;

  return str[0].toUpperCase() + str.slice(1);
}

alert( ucFirst("john") ); // Ahmad
```

ب. فحص وجود شيء مزعج

الأهمية: ★★★★★

اكتب دالة باسم `checkSpam(str)` تُرجِع `true` إن كان `str` يحوي 'viagra' أو 'XXX'، وإلا فتُرجِع `false`. يجب أن لا تكون الدالة حساسة لحالة الأحرف:

```
checkSpam('buy ViAgRA now') == true
checkSpam('free xxxxx') == true
checkSpam("innocent rabbit") == false
```

الحل:

لجعل البحث غير حساس لحالة الأحرف، نحوِّل النص إلى أحرف صغيرة ومن ثم نبحث فيه على النص المطلوب:

```
function checkSpam(str) {
  let lowerStr = str.toLowerCase();

  return lowerStr.includes('viagra') || lowerStr.includes('xxx');
}

alert( checkSpam('buy ViAgRA now') );
alert( checkSpam('free xxxxx') );
alert( checkSpam("innocent rabbit") );
```

ج. قص النص

الأهمية: ★★★★★

أنشئ دالة باسم `truncate(str, maxlength)` تفحص طول النص `str` وتستبدل نهايته التي تتجاوز الحد `maxlength` بالرمز "…" لجعل طولها يساوي `maxlength` بالضبط. يجب أن تكون مخرجات الدالة النص المقصوص (في حال حدث ذلك). مثلًا:

```
truncate("What I'd like to tell on this topic is:", 20) = "What I'd like to te…"

truncate("Hi everyone!", 20) = "Hi everyone!"
```

الحل:

الطول الكلي هو `maxlength`، لذا فإننا نحتاج لقص النص إلى أقصر من ذلك بقليل لإعطاء مساحة للنقـط "…". لاحظ أن هناك حرف يونيكود واحد للحرف "…". وليست ثلاث نقاط.

```
function truncate(str, maxlength) {
  return (str.length > maxlength) ?
    str.slice(0, maxlength - 1) + '…' : str;
}
```

د. استخراج المال

الأهمية: ★★★★☆

لـدينا قيمـة بالشـكل " $120 "، إذ علامـة الـدولار تـأتي أولًا ومن ثم العـدد. أنشِـئ دالـة باسـم `extractCurrencyValue(str)` تستخرج القيمة العددية من نصوص مشابهة وإرجاعها. مثال:

```
alert( extractCurrencyValue('$120') === 120 ); // true
```

الحل:

```
function extractCurrencyValue(str) {
  return +str.slice(1);
}
```

5.4 المصفوفات Arrays

تُتيح لك الكائنات تخزين القيم في مجموعات ذات مفـاتيح، وهـذا أمـر طيّـب. ولكنّ سـتحتاج دومًـا مـا في عملك إلى مجموعة مرتّبة، أي أنَّ العناصر مرتّبة: عنصر أوّل، عنصر ثانٍ، عنصر ثالث، وهكذا دواليك. تُفيدنا هـذه الطريقة في تخزين أمور مثل: المستخدمين والبضائع وعناصر HTML وغيرها.

هنا يكون استعمال الكائنات غير موفَّق، إذ أنَّها لا تقدّم لنا أيّ تابِع يتيح تحديـد تـرتيب العناصـر، فلا يمكننـا إضافة خاصيةً جديدةً تحلّ بين الخاصيات الموجودة. لم تُصنَع الكائنات لهذا الغرض بتاتًا.

توجد بنية بيانـات أخـرى باسـم `Array` (نـدعو هـذا النـوع بالمصـفوفة) وهي تـتيح لنـا تخـزين مجموعـات العناصر مرتّبةً.

5.4.1 التصريح

توجد صياغتان اثنتان لإنشاء مصفوفة فارغة:

```
let arr = new Array();
let arr = [];
```

تحتاج في عملك أغلب الوقت (ونقول أغلب الـوقت) الصـياغةَ الثانيـة. يمكننـا أيضًـا تقـديم عناصـر أوليّـة للمصفوفة نكتبها في أقواس:

```
let fruits = ["Apple", "Orange", "Plum"];
```

لاحظ أنَّ عناصر المصفوفات مُرقَّمة (مُفهرَسَة) بدءًا من الرقم صـفر. ويمكننـا أن نأخـذ عنصـرًا منهـا بكتابـة ترتيبه في أقواس معقوفة:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple/تفاحة
alert( fruits[1] ); // Orange/برتقالة
alert( fruits[2] ); // Plum/برقوق
```

يمكن أيضًا تعويض أحد العناصر بأخرى:

```
fruits[2] = 'Pear'; // ["Apple", "Orange", "Pear"] صارت
```

…أو إضافة أخرى جديدة إلى المصفوفة:

```
fruits[3] = 'Lemon'; // Apple", "Orange", "Pear", "] صارت (ليمون)
[""Lemon
```

نعرف باستعمال التابع length إجمالي العناصر في المصفوفة:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits.length ); // 3
```

يمكننا أيضًا استعمال alert لعرض المصفوفة كاملةً.

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits ); // Apple,Orange,Plum
```

كما يمكن للمصفوفات تخزين أيّ نوع من البيانات. مثلًا:

```
// قيم مختلفة الأنواع
let arr = [ 'Apple', { name: 'Ahmad' }, true, function() {
alert('hello'); } ];

// خُذ الكائن ذا الفهرس 1 ثمّ اعرض اسمه
alert( arr[1].name ); // Ahmad

// خُذ الدالة في الفهرس 3 ثمّ شغّلها
arr[3](); // hello
```

الفاصلة نهاية الجملة

كما الكائنات، يمكن أن نُنهي عناصر المصفوفات بفاصلة ,:

```
let fruits = [
  "Apple",
  "Orange",
  "Plum",
];
```

يُسهّل أسلوب الكتابة "بالفاصلة نهاية الجملة" إضافة العناصر وإزالتها، إذ أن الأسطر البرمجية كلها تصير متشابهة.

5.4.2 توابِع الدفع والجلب

من بين أنواع المصفوفات، تُعدّ الطوابير (queue) أكثرها استعمالًا. تعني الصفوف (في علـوم الحاسـوب) تجميعات العناصر المُرتَّبة والتي تدعم العمليتين هاتين:

- الدفع push: يُضيف عنصرًا نهاية الصفّ
- الأخذ `shift`: يأخذ عنصرًا من بداية الصفّ، فيتحرّك الصفّ ويصير العنصر الثاني هو الأول فيه.

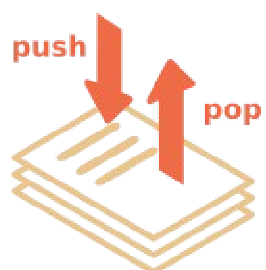
تدعم المصفوفات كلتا العمليتين هاتين. وفي الحياة العمليـة، اسـتعمال هـاتين العمليـتين ضـروري دومًـا. نأخذ مثلًا مجموعة رسائل مرتّبة يجب عرضها على الشاشة، أو "صف رسائل".

هناك طريقة أخرى لاستعمال المصفوفات، وهي بنية البيانات بالاسم "كومة" (stack).

تدعم الأكوام عمليتين أيضًا:

- الدفع push: يُضيف عنصرًا نهاية الكومة.
- السحب pop: يأخذ عنصرًا من نهاية الكومة.

أي أنّ العناصر الجديدة تُضاف دومًا إلى آخر الكومة، وتُزال أيضًا من نهايتها. عادةً ما نرسم هذه الأكـوام مثـل أكوام بطاقات اللعب: البطاقات الجديدة تُضَاف أعلى الكومة، وتُأخذ من أعلاها أيضًا:



في الأكوام، آخر عنصر ندفعه إليها يكون أوّل من يُأخَذ، ويسمَّى هـذا بمبـدأ "آخـر من يـدخل أول من يخـرج" (Last-In-First-Out تختصـر إلى LIFO). أمّـا في الطوابـير، فهي "أول من يـدخل أول من يخـرج" (First-In-First-Out تختصر إلى FIFO).

تعمـل المصـفوفات في JavaScript بـالطريقتين، صـفوف أو أكـوام. يمكنـك اسـتعمالها لإضـافة العناصـر وإزالتها من/إلى بداية المصفوفة ونهايتها.

تُسمّى بنية البيانات هذه (في علوم الحاسوب) باسم "الطوابير ذات الطرفين" (deque).

التوابع التي تؤثّر على نهاية المصفوفة:

- pop: يستخرج آخر عنصر من المصفوفة ويُعيده:

```
let fruits = ["Apple", "Orange", "Pear"];

alert( fruits.pop() ); // alert بتنبيه عبر الدالة "Pear" أزِل

alert( fruits ); // Apple, Orange
```

- push: يُضيف العنصر إلى آخر المصفوفة:

```
let fruits = ["Apple", "Orange"];

fruits.push("Pear");

alert( fruits ); // Apple, Orange, Pear
```

باستدعاء `fruits.push(...)` كأنّما استدعيت `fruits[fruits.length] = ...`.

التوابع التي تؤثّر على بداية المصفوفة:

- shift: يستخرج أوَّل عنصر من المصفوفة وتُعيده:

```
let fruits = ["Apple", "Orange", "Pear"];

alert( fruits.shift() ); // alert أزِل التفاحة واعرضها بِ

alert( fruits ); // Orange, Pear
```

- unshift: يُضيف العنصر إلى أوَّل المصفوفة:

```
let fruits = ["Orange", "Pear"];

fruits.unshift('Apple');

alert( fruits ); // Apple, Orange, Pear
```

يمكنك أيضًا إضافة أكثر من عنصر في استدعاء واحد من push وunshift:

```
let fruits = ["Apple"];
```

```
fruits.push("Orange", "Peach");
fruits.unshift("Pineapple", "Lemon");

// ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
alert( fruits );
```

5.4.3 داخليًا وخلف الكواليس

المصفوفات هي كائنات (النوع object)، كائنات من نوع خاص. القوسان المعقوفان المستعملان للـدخول إلى الخاصيات `arr[0]` هما فعليًا جزء من صياغة الكائنات. داخليًا، لا يفـرق ذاك عن `obj[key]` (إذ `arr` هـو الكائن والأرقام تلك مفاتيح).

ما تفعله المصفوفات هو "توسعة" الكائنات بتقديم توابِع خاصَّة تعمل مع البيانـات والمجموعـات المُرتَّبـة، إضافةً إلى تقديم خاصية `length`، ولكنّ أساسها ما زال الكائنات.

تذكَّر أنَّ هناك 7 أنواع أساسية في JavaScript، فقط. المصفوفة هي كائن، وتتصرَّف بناءً على ذلك، ككائن. فمثلًا، عند نسخها تُنسخ بالمرجع (By reference):

```
let fruits = ["Banana"]
let arr = fruits; // انسخها بالمرجع (متغيران اثنان يُشيران إلى نفس المصفوفة)

alert( arr === fruits ); // true
arr.push("Pear"); // عدّل المصفوفة "بالمرجع"
alert( fruits ); // Banana, Pear :صاروا الآن عنصرين
```

...إلا أنّ المميز حقًا في المصفوفات هي آلية تمثيلها داخليًا، إذ يحاول المُحرِّك تخزين عناصرها متتابعـةً في مساحة الذاكرة، أي واحدة بعد الأخرى، تمامًا مثلما وضَّحت الرسوم في هذا الفصـل. هنـاك أيضًـا طرائـق أخـرى لتحسين (optimization) المصفوفات فتعمل بسرعة كبيرة حقًا.

ولكن، لو لم نعمل مع المصفوفة على أنّها "تجميعة مُرتَّبة" بل وكأنَّها كائن مثل غيرها، فسينهار هذا كله.

يمكننا (تقنيًا) كتابة هذا:

```
let fruits = []; // نصنع مصفوفة
fruits[99999] = 5; // نُسند خاصيةً لها فهرس أكبر من طول المصفوفة بكثير
fruits.age = 25; // نُنشئ خاصيةً لها أيّ اسم
```

أجل، يمكننا فعل هذا، فالمصفوفات في أساسها كائنات، ويمكننا إضافة مـا نريـد من خاصـيات لهـا. ولكن المُحرِّك هنا سيرى بأنَّا نُعامل المصفوفة معاملة الكائن العادي. وبهذا -في هذه الحالـة- لا تنفـع أنـواع التحسـين المخصّصة للكائنات، وسيُعطِّلها المحرّك، وتضيع كل فوائد المصفوفات.

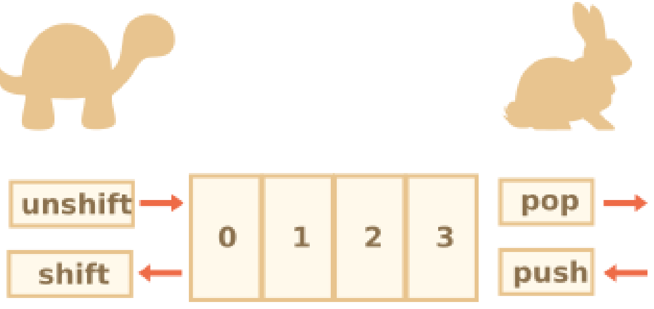
هذه طرائق يمكنك فيها إساءة استعمال المصفوفات:

- إضافة خاصيات ليست عددية مثل `arr.test = 5`.
- الفراغات، أي تُضيف `arr[0]` وبعدها `arr[1000]` (دون عناصر بينها).
- ملء المصفوفة بالعكس، أي `arr[1000]` ثم `arr[999]` وهكذا.

نرجوك هنا أن تعتبر المصفوفات بنًى خاصَّة تتعامل مع "البيانات المرتّبـة" (ordered data)، فهي تقـدِّم لك توابِع خاصَّة لهذا بالذات. يتغيّـر تعامـل محرّكـات JavaScript حين تتعامـل مـع المصـفوفات، فتعمـل مـع البيانات المُرتَّبة المتتابعة، فمن فضلك استعمِلها بهذه الطريقة. لو أردت مفاتيح لا عددية، أو مثلما في الحـالات الثلاث أعلاه، فغالبًا لا تكون المصفوفة ما تبحث عنه، بل الكائنات العادية `{}`.

5.4.4 الأداء

يعمل التابعان `push/pop` بسرعة، بينما `shift/unshift` بطيئان.



لماذا يكون التعامل مع نهاية المصفوفة أسرع من التعامل مع بدايتها؟ لنأخذ نظرة عمًا يحـدث أثنـاء تنفيـذ الشيفرة:

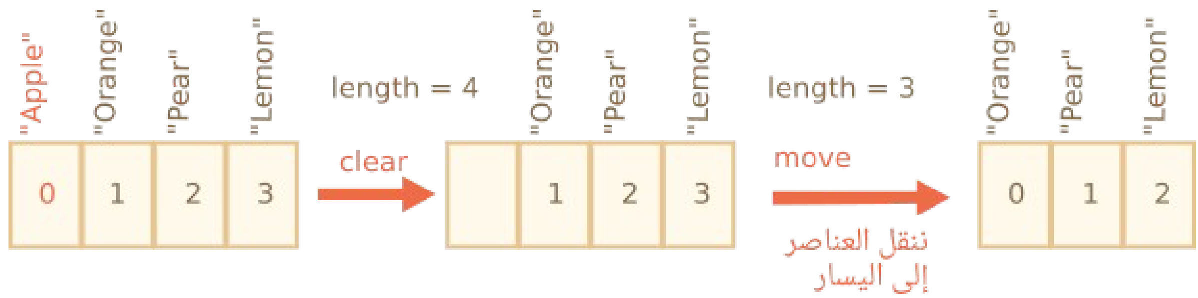
```
fruits.shift(); // خُذ عنصرًا واحدًا من الأوّل
```

لا يكفي أن تأخذ العنصر ذا الفهرس `0` وتُزيله، بل عليك أيضًا إعادة ترقيم بقية العناصر وفقًا لذلك.

ما تفعله عملية `shift` هي ثلاث أمور:

1. إزالة العنصر ذا الفهرس `0`.
2. تحريك كل العناصر الأخرى إلى يسار المصفوفة، وإعـادة ترقيمهـا من الفهـرس رقم `1` إلى `0`، ومن `2` إلى `1`، وهكذا.

3. تحديث خاصية الطول `length`.



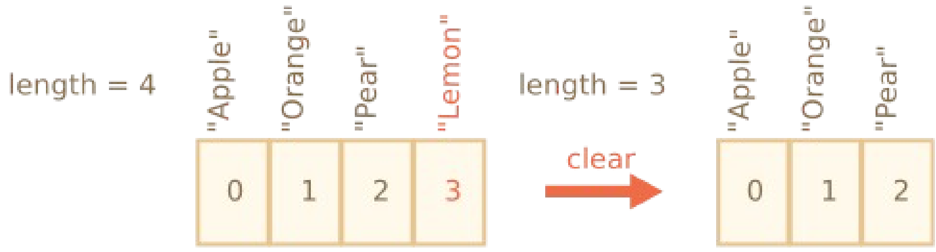
زِد العناصر في المصفوفات، تزيد الوقت اللازم لتحريكها، وتزيد عدد العمليات داخل الذاكرة.

مثل `shift`، تفعل `unshift` نفس الأمور: فلنُضيف عنصرًا إلى بداية المصفوفة، علينا أولًا تحريك كل العناصر إلى اليمين، أي نزيد فهارسها كلها.

وماذا عن `push/pop`؟ ليس عليها تحريك أيّ عنصر. فلاستخراج عنصر من النهاية، يمحي التابع `pop` الفهرس ويعدِّل الطول `length` فيقصِّره.

إجراءات عملية `pop`:

```
fruits.pop(); // خُذ عنصرًا واحدًا من الآخر
```



لا تحتاج عملية `pop` إلى تحريك ولا مقدار ذرة، لأنّ العناصر تبقى كما هي مع فهارسها. لهذا السبب سرعتها تفوق سرعة البرق، أي أقصى سرعة ممكنة.

ذات الأمر للتابِع `push`.

5.4.5 الحلقات

هذه إحدى أقدم الطرق للمرور على عناصر المصفوفات، استعمال حلقة `for` بالمرور على فهارس المصفوفة:

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
```

```
  alert( arr[i] );
}
```

ولكن المصفوفات تسمح بطريقة أخرى للمرور عليها، `for..of`:

```
let fruits = ["Apple", "Orange", "Plum"];

// المرور على عناصر المصفوفة
for (let fruit of fruits) {
  alert( fruit );
}
```

لا تتيح لك حلقة `for..of` الوصول إلى فهرس العنصر الحالي في الحلقة، بل قيمة العنصر فقط، وفي أغلب الأحيان هذا ما تحتاج، كما وأنَّ الشيفرة أقصر.

طالما المصفوفات كائنات، فيمكننا (نظريًا) استعمال `for..in`:

```
let arr = ["Apple", "Orange", "Pear"];

for (let key in arr) {
  alert( arr[key] ); // Apple, Orange, Pear
}
```

ولكن الواقع أنّ الطريقة هذه سيئة، ففيها عدد من المشاكل:

1. تمرّ الحلقة `for..in` على كل الخاصيات مجتمعةً، وليس العددية منها فقط. توجد في المتصفّح وغيرها من بيئات كائنات "شبيهة بالمصفوفات". أي أن لها خاصية الطول `length` وخاصيات الفهارس، ولكن لها أيضًا توابِع وخاصيات لا عددية أخرى لا نحتاج إليها أغلب الأحيان، إلّا أنَّ حلقة `for..in` ستَمرُّ عليها هي أيضًا. لذا لو اضطررت للعمل مع الكائنات الشبيهة بالمصفوفات، فهذه الخاصيات "الأخرى" ستَتسبّب بالمتاعب بلا شك.
2. أداء حلقة `for..in` يكون بالنحو الأمثل على الكائنات العامة لا المصفوفات، ولهذا سيكون أبطأ 10 أو 100 مرة. طبعًا فالأداء سريع جدًا مع ذلك. هذه السرعة الإضافية ستنفع غالبًا في الحالات الحرجة (أي حين يجب أن يكون تنفيذ الحلقة بأسرع وقت ممكن). مع ذلك، الحرس واجب والاهتمام بهذا الاختلاف مهم.

لكن في أغلب الأحيان، استعمال `for..in` للمصفوفات فكرة سيئة.

5.4.6 كلمتان حول "الطول"

تَتحدَّث خاصية الطول length تلقائيًا متى ما عدَّلنا المصفوفة. وللدّقة، فهي ليست عدد القيم في المصفوفة، بل أكبر فهرس موجود زائدًا واحد.

فمثلًا، لو كان لعنصر واحد فهرس كبير، فسيكون الطول كبيرًا أيضًا:

```
let fruits = [];
fruits[123] = "Apple";

alert( fruits.length ); // 124
```

لكنّنا لا نستعمل المصفوفات هكذا، سجِّلها عندك.

هناك ما هو عجيب حول خاصية length، ألا وهي أنّها تقبل الكتابة. لو زِدنا قيمتها يدويًا، لا نرى شيئًا تغيّر، ولكن لو أنقصناها، تُبتر المصفوفة حسب الطول، ولا يمكن العودة عن هذه العملية. طالِع هذا المثال:

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // نبتر المصفوفة ونُبقي عنصرين فقط
alert( arr ); // [1, 2]

arr.length = 5; // نُعيد الطول الذي كان في الأوّل
alert( arr[3] ); // undefined القيم المبتورة لا تعود، وإنما تصبح
```

إذًا، فالطريقة الأسهل والأبسط لمسح المصفوفة هي: `arr.length = 0;`.

5.4.7 new Array()

هناك صياغة أخرى يمكن استعمالها لإنشاء المصفوفات:

```
let arr = new Array("Apple", "Pear", "etc");
```

ولكنّها نادرًا ما تُستعمَل، فالأقواس المعقوفة [] أقصر. كما وأنّ هذه الطريقة تقدّم ميزة... مخادعة، إن صحّ التعبير. إن استدعيت new Array وفيها مُعامل واحد فقط (عددي)، فستُنشأ مصفوفة لا عناصر فيها، ولكن بالطول المحدّد.

هاك طريقة يمكنك بها تخريب حياتك، لو أردت:

```
let arr = new Array(2); // هل ستكون المصفوفة [2]؟
```

```
alert( arr[0] ); // غير معرّفة! ليس فيها عناصر.

alert( arr.length ); // طولها 2
```

في هذا الشيفرة، كل عناصر `new Array(number)` لها القيمة `undefined`. ولهذا نستعمل الأقواس المعقوفة غالبًا، لنتجنّب هذه المفاجئات السارّة، إلّا لو كنت تعي حقًا ما تفعله.

5.4.8 المصفوفات متعدّدة الأبعاد

يمكن أن تكون عناصر المصفوفات مصفوفات أخرى أيضًا. نستغلّ هذه الميزة فنعمل مصفوفات متعدّدة الأبعاد لتخزين المصفوفات الرياضية مثلًا:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

alert( matrix[1][1] ); // 5، العنصر في الوسط
```

5.4.9 تحويل المصفوفات إلى سلاسل نصية

تُنفِّذ المصفوفات تابِع `toString` خاصّ بها، فيُعيد قائمة من العناصر مفصولة بفواصل. خُذ هذا المثال:

```
let arr = [1, 2, 3];

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

جرّب هذه، أيضًا:

```
alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
```

ليس للمصفوفات `Symbol.toPrimitive` ولا دالة `valueOf`، بل تُنفِّذ التحويل `toString` فقط لا غير. هكذا تصير `[]` سلسلة نصية فارغة، و`[1]` تصير `"1"` و`[1,2]` تصير `"1,2"`.

متى ما أضاف عامل الجمع الثنائي `"+"` شيئًا إلى السلسلة النصية، حَوَّله إلى سلسلة نصية هو الآخر. هكذا هي الخطوة التالية:

```
alert( "" + 1 ); // "1"
alert( "1" + 1 ); // "11"
alert( "1,2" + 1 ); // "1,21"
```

5.4.10 الخلاصة

المصفوفات نوع خاصّ من الكائنات، وهي مُخصَّصة لتخزين البيانات عناصر مُرتَّبة، كما وإدارتها أيضًا.

- التصريح

```
// الأقواس المعقوفة (طبيعية)
let arr = [item1, item2...];

// new Array (نادرة جدًا)
let arr = new Array(item1, item2...);
```

باستدعاء `new Array(number)` تُنشئ مصفوفة بالطول المُحدَّد، ولكن بلا أيّ عنصر.

- خاصية الطول `length` هي طول المصفوفة، أو للدّقة، آخر فهرس عددي زائدًا واحد. التوابع المختلفة على المصفوفات تعدّل هذه الخاصية تلقائيًا.

- إن قصَّرنا خاصية `length` يدويًا، فنحن نبتر المصفوفة حسب القيمة الجديدة.

يمكننا استعمال المصفوفة كما الصفوف ذات الطرفين، بالعمليات الآتية:

- `push(...items)`: تُضيف `items` إلى النهاية.
- `pop()`: تُزيل العنصر من النهاية وتُعيده.
- `shift()`: تُزيل العنصر من البداية وتُعيده.
- `unshift(...items)`: تُضيف `items` إلى البداية.

لتمرّ على عناصر المصفوفة:

- `for (let i=0; i<arr.length; i++)`: تُنفَّذ بسرعة، ومتوافقة مع المتصفحات القديمة.
- `for (let item of arr)`: الصياغة الحديثة للعناصر فقط.
- `for (let i in arr)`: إيّاك واستعمالها.

سنرجع إلى المصفوفات لاحقًا ونتعلّم توابِع أخرى لإضافة العناصر وإزالتها واستخراجها، كما وترتيب المصفوفات. هذا كله في الفصل التالي، توابع المصفوفات.

5.4.11 تمارين

ا. هل تُنسخ المصفوفات؟

الأهمية: ★★★☆☆

ما ناتج هذه الشيفرة؟

```
let fruits = ["Apples", "Pear", "Orange"];

// ادفع عنصرًا جديدًا داخل "النسخة"
let shoppingCart = fruits;
shoppingCart.push("Banana");

// ماذا في fruits؟
alert( fruits.length ); // ?
```

الحل:

الناتج هو 4:

```
let fruits = ["Apples", "Pear", "Orange"];

let shoppingCart = fruits;

shoppingCart.push("Banana");

alert( fruits.length ); // 4
```

هذا لأنّ المصفوفات كائنات. فكِلا `shoppingCart` و`fruits` يُشيران إلى نفس المصفوفة ذاتها.

ب. العمليات على المصفوفات

الأهمية: ★★★★★

فلنجرِّب خمس عمليات على المصفوفات.

1. أنشِئ مصفوفة باسم `styles` تحوي العنصرين "Jazz" و"Blues".
2. أضِف "Rock-n-Roll" إلى نهايتها.

3. استبدِل القيمة في الوسط بالقيمة "Classics". يجب أن تعمل الشيفرة الذي ستكتبه ليجد القيمة في الوسط مع أيّ مصفوفة كانت لو كان طولها عدد فردي.

4. أزِل القيمة الأولى من المصفوفة واعرضها.

5. أضِف "Rap" و"Reggae" إلى بداية المصفوفة.

المصفوفة خلال العمليات هذه:

```
Jazz, Blues
Jazz, Blues, Rock-n-Roll
Jazz, Classics, Rock-n-Roll
Classics, Rock-n-Roll
Rap, Reggae, Classics, Rock-n-Roll
```

الحل:

```
let styles = ["Jazz", "Blues"];
styles.push("Rock-n-Roll");
styles[Math.floor((styles.length - 1) / 2)] = "Classics";
alert( styles.shift() );
styles.unshift("Rap", "Reggae");
```

ج. النداء داخل سياق المصفوفة

الأهمية: ★★★★★

ما الناتج؟ لماذا؟

```
let arr = ["a", "b"];

arr.push(function() {
  alert( this );
})

arr[2](); // ?
```

الحل:

من ناحية الصياغة، فالاستدعاء `arr[2]()` هو نفسه النداء القديم `obj[method]()`، فبدل `obj` هناك `arr`، وبدل `method` هناك 2.

إذًا فما أمامنا هو نداء الدالة `arr[2]` وكأنّها تابِع لكائن. وبالطبيعة، فهي تستلم `this` الذي يُشير إلى الكائن `arr` وتكتب المصفوفة ناتجًا:

```
let arr = ["a", "b"];

arr.push(function() {
  alert( this );
})

arr[2](); // "a","b",function
```

للمصفوفة ثلاث قيم: الاثنين من البداية، مع الدالة.

د. جمع الأعداد المُدخَلة

الأهمية: ★★★★☆

اكتب دالة `sumInput()` تؤدّي الآتي:

- طلب القيم من المستخدم باستعمال `prompt` وتخزينها في مصفوفة.
- أن ينتهي الطلب لو أدخل المستخدم قيمة غير عددية، أو سلسلة نصية فارغة، أو ضغطَ "ألغِ".
- حساب مجموع عناصر المصفوفة وإعادتها.

ملاحظة: الصفر `0` عدد مسموح، لذا لا تُوقف الطلب لو رأيته.

الحل:

انتبه هنا على التفصيل الصغير في الحل، صغير ولكن مهمّ: لا يمكننا تحويل قيمة المتغير `value` إلى عدد مباشرةً بعد `prompt`، لأنّه بعدما نُجري `value = +value`، لن نفرِّق بين السلسلة النصية الفارغة (أي علينا إيقاف الطلب) من الصفر (قيمة صالحة). عوض ذلك نؤجّل ذلك لما بعد.

```
function sumInput() {

  let numbers = [];

  while (true) {

    let value = prompt("A number please?", 0);
```

```
    // هل نلغي الطلب؟
    if (value === "" || value === null || !isFinite(value)) break;

    numbers.push(+value);
  }

  let sum = 0;
  for (let number of numbers) {
    sum += number;
  }
  return sum;
}

alert( sumInput() );
```

ه. أكبر مصفوفة فرعية

الأهمية: ★★☆☆☆

البيانات المُدخلة هي مصفوفة من الأعداد، مثل `arr = [1, -2, 3, 4, -9, 6]`. والمهمة هي: البحث عن مصفوفة فرعية متتابعة في `arr` لها أكبر ناتج جمع.

اكتب دالة `getMaxSubSum(arr)` لتُعيد ذلك الناتج.

مثال:

```
getMaxSubSum([-1, 2, 3, -9]) = 5 (3+2 مجموع)
getMaxSubSum([2, -1, 2, 3, -9]) = 6 (3+2+(1-)+2 مجموع)
getMaxSubSum([-1, 2, 3, -9, 11]) = 11 (...وهكذا)
getMaxSubSum([-2, -1, 1, 2]) = 3
getMaxSubSum([100, -9, 2, -3, 5]) = 100
getMaxSubSum([1, 2, 3]) = 6 (نأخذها كلها)
```

إن كانت القيم كلها سالبة فيعني هذا ألا نأخذ شيئًا (المصفوفة الفرعية فارغة)، وبهذا يكون الناتج صفرًا:

```
getMaxSubSum([-1, -2, -3]) = 0
```

يُحبَّذ لو تُفكِّر -رجاءً- بحلّ سريع: O(n2) أو حتّى O(n) لو أمكنك.

الحل:

- النسخة البطيئة

يمكننا حساب كلّ ناتج جمع فرعي ممكن.

أبسط طريقة هي أخذ كلّ عنصر وحساب مجموع المصفوفات الفرعية بدءًا من مكان العنصر.

فمثلًا إن كان لدينا `[-1, 2, 3, -9, 11]`:

```
// :-1 نبدأ بِـ
-1
-1 + 2
-1 + 2 + 3
-1 + 2 + 3 + (-9)
-1 + 2 + 3 + (-9) + 11

// :2 نبدأ بِـ
2
+ 3
+ 3 + (-9)
+ 3 + (-9) + 11

// :3 نبدأ بِـ
3
+ (-9)
+ (-9) + 11

// :-9 نبدأ بِـ
-9
-9 + 11

// :11 نبدأ بِـ
11
```

في الواقع فالشيفرة هي حلقات متداخلة، تمرّ الحلقة العلوية على عناصر المصفوفة، والسفلية تعـدّ النـواتج الفرعية بدءًا من العنصر الحالي.

```
function getMaxSubSum(arr) {
  let maxSum = 0; // إن لم نأخذ أيّ عنصر، فسنُرجع الصفر 0

  for (let i = 0; i < arr.length; i++) {
    let sumFixedStart = 0;
    for (let j = i; j < arr.length; j++) {
      sumFixedStart += arr[j];
      maxSum = Math.max(maxSum, sumFixedStart);
    }
  }

  return maxSum;
}

alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5
alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11
alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3
alert( getMaxSubSum([1, 2, 3]) ); // 6
alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100
```

مدى التعقيد الحسابي لهـذا الحـل هـو O(n2). أي بعبـارة أخـرى، لـو زِدت حجم المصـفوفة مـرتين اثنـتين، فسيزيد وقت عمل الخوارزمية أربع مرات أكثر.

يمكن أن تؤدِّي هذه الخوارزميات للمصفوفات الكبـيرة (نَتحـدَّث عن 1000 و10000 وأكـثر) إلى بطء شـديد في التنفيذ.

- النسخة السريعة

لنمرّ على عناصر المصفوفة ونحفظ ناتج جمع العناصر الحالي في المتغير s. مـتى مـا صـار s سـالبًا، نعيّنـه صفرًا s=0. إجابتنا على هذا هي أكبر قيمة من هذا المتغير s.

لو لم يكن هذا الوصف منطقيًا، فيمكنك مطالعة الشيفرة، قصيرة للغاية:

```
function getMaxSubSum(arr) {
  let maxSum = 0;
  let partialSum = 0;

  for (let item of arr) { // لكلّ item في arr
```

```
      partialSum += item; // partialSum نُضيفه إلى
      maxSum = Math.max(maxSum, partialSum); // نتذكّر أكبر قيمة
      if (partialSum < 0) partialSum = 0; // لو كانت سالبة فالصفر
    }

    return maxSum;
  }

  alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5
  alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11
  alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3
  alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100
  alert( getMaxSubSum([1, 2, 3]) ); // 6
  alert( getMaxSubSum([-1, -2, -3]) ); // 0
```

على الخوارزمية هنا أن تمرّ مرورًا واحدًا فقط على المصفوفة، أي أن التعقيد الحسابي هو O(n).

يمكنك أن تجد معلومات مفصّلة أكثر عن الخوارزمية هنا: Maximum subarray problem. لو لم يكن هذا واضحًا بعد، فالأفضل لو تتعقّب ما تفعل الخوارزمية في الأمثلة أعلاه، وترى ما تفعله من حسابات. "التعقّب يغني عن ألف كلمة"... ربّما.

5.5 توابِع المصفوفات

تقدِّم المصفوفات توابِع عديدة تُسهِّل التعامل معها. ولتبسيطها سنقسّمها إلى مجموعات بحسب الوظيفة في هذا الفصل ونشرح كلٌّ منها على حدة.

5.5.1 إضافة العناصر وإزالتها

عرفنا من الفصل الماضي بالتوابِع التي تُضيف العناصر وتُزيلها من بداية أو نهاية المصفوفة:

- `arr.push(...items)`: يُضيف العناصر إلى النهاية،
- `arr.pop()`: يستخرج عنصرًا من النهاية،
- `arr.shift()`: يستخرج عنصرًا من البداية،
- `arr.unshift(...items)`: يُضيف العناصر إلى البداية.

وهذه أخرى غيرها.

ا. الوصل splice

يا ترى كيف نحذف أحد عناصر المصفوفة؟ المصفوفات كائنات، يمكننا تجربة `delete` وربما تنجح:

```
let arr = ["I", "go", "home"];

delete arr[1]; // "go" أزِل

alert( arr[1] ); // undefined

// arr = ["I",  , "home"]; صارت المصفوفة الآن
alert( arr.length ); // 3
```

أُزيل العنصر صحيح، ولكنّ ما زال في المصفوفة ثلاثة عناصر، كما نرى في `arr.length == 3`.

هذا طبيعي، إذ يُزيل `delete obj.key` القيمة بمفتاحها `key`... وهذا فقط. ينفع للكائنات ربّما، لكنّا نريدها للمصفوفات أن تنتقل كل العناصر على اليمين وتأخذ الفراغ الجديد. أي أننا نتوقع أن تصغر المصفوفة الآن، لهذا السبب علينا استعمال توابِع خاصّة لذلك.

يمكننا تشبيه التابع arr.splice(start) بالتابع "بتاع كُلّو" للمصفوفات (كما يُقال بالعامية). يمكنه أن يُجري ما تريد للعناصر: إدراج، إزالة، استبدال.

هذه صياغته:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

يبدأ التابع من عند العنصر ذي الفهرس `index`، فيُزيل `deleteCount` من العناصر ويُدرج العناصر `elem1, ..., elemN` المُمرّرة إليه مكانها. أخيرًا يُعيد المصفوفة بالعناصر المُزالة.

فهم هذا التابع بالأمثلة أبسط. فلنبدأ أولًا بالحذف:

```
let arr = ["I", "study", "JavaScript"];

arr.splice(1, 1); // أزِل من العنصر ذا الفهرس 1 عنصرًا واحدًا (1)

alert( arr ); // ["I", "JavaScript"]
```

رأيت؟ سهلة. نبدأ من العنصر ذي الفهرس `1` ونُزيل عنصرًا واحدًا (`1`).

الآن، نُزيل ثلاثة عناصر ونستبدلها بعنصرين آخرين:

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// أزِل الثلاث عناصر الأولى وعوّضها بتلك الأخرى
arr.splice(0, 3, "Let's", "dance");

alert( arr ) // ["Let's", "dance", "right", "now"] صارت الآن
```

أمّا هنا فكيف يُعيد `splice` مصفوفةً بالعناصر المُزالة.

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// أزِل أوّل عنصرين
let removed = arr.splice(0, 2);

alert( removed ); // "I", "study" <-- قائمة بالعناصر المُزالة
```

يمكن أن يُدرج تابِع `splice` العناصر دون إزالة أيّ شيء أيضًا. كيف؟ نضع `deleteCount` يساوي الصفر `0`:

```
let arr = ["I", "study", "JavaScript"];

```

```
// من الفهرس 2
// احذف 0
// ثمّ أرجِع "complex" و "language"
arr.splice(2, 0, "complex", "language");

alert( arr ); // "I", "study", "complex", "language", "JavaScript"
```

الفهارس السالبة ممكنة أيضًا

يمكننا هنا وفي توابِع المصفوفات الأخرى استعمال الفهارس السالبة. وظيفتها تحديد المكان بدءًا من نهاية المصفوفة، هكذا:

```
let arr = [1, 2, 5];

// من الفهرس ‎-1 (العنصر قبل الأخير) //
// احذف 0 عنصر (لا شيء)،
// ثمّ أدرج 3 و 4
arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5
```

ب. القطع slice

التابِع arr.slice أبسط بكثير من شبيهه `arr.splice`.

صياغته هي:

```
arr.slice([start], [end])
```

وهو يُعيد مصفوفة جديدةً بنسخ العناصر من الفهرس `start` إلى `end` (باستثناء `end`). يمكن أن تكون `start` وحتّى `end` سالبتان، بهذا يُعدّ المحرّك القيمتان أماكن بدءًا من نهاية المصفوفة.

هذا التابِع يشبه تابِع السلاسل النصية `str.slice`، ولكن بدل السلاسل النصية الفرعية، يُعيد المصفوفات الفرعية. إليك المثال الآتي:

```
let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s (نسخة تبدأ من 1 وتنتهي عند 3)
alert( arr.slice(-2) ); // s,t (نسخة تبدأ من -2 وتنتهي في النهاية)
```

يمكننا أيضًا استدعائها بلا وُسطاء: يُنشئ `arr.slice()` نسخة عن `arr`. نستعمل هذا غالبًا لأخذ نسخة وإجراء تعديلات عليها دون تعديل المصفوفة الأصلية، وتركها كما هي.

ج. الربط concat

يُنشئ التابع arr.concat مصفوفةً جديدة فيها القيم الموجودة في المصفوفات والعناصر الأخرى.

صياغته هي:

```
arr.concat(arg1, arg2...)
```

وهو يقبل أيّ عدد من الوُسطاء، أكانت مصفوفات أو قيم. أمّا ناتجه هو مصفوفة جديدة تحوي العناصر من `arr`، ثم `arg1` فَـ `arg2` وهكذا دواليك. لـو كـان الوسـيط `argN` نفسـه مصـفوفة، فستُنسَـخ كـل عناصـره، وإلّا فسيُنسخ الوسيط نفسه. لاحِظ هذا المثال:

```
let arr = [1, 2];

// اصنع مصفوفة فيها العنصرين: arr و [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4

// اصنع مصفوفة فيها العناصر: arr و[3,4] و[5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// اصنع مصفوفة فيها العنصرين: arr و[3,4]، بعدها أضِف القيمتين 5 و 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

عادةً تنسخ المصفوفة عناصر المصفوفات الأخرى. بينما الكائنات الأخرى (حتّى لو كانت مثل المصفوفات) فستُضاف كتلة كاملة.

```
let arr = [1, 2];

let arrayLike = {
"something",
  length: 1
};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]
```

... ولكن لو كان للكائن الشبيه بالمصفوفات خاصية `Symbol.isConcatSpreadable`، فستعامل معه `concat` مثلما تتعامل مع المصفوفات: ستُضاف عناصره بدل كيانه:

```
let arr = [1, 2];

let arrayLike = {
"something",
"else",
  [Symbol.isConcatSpreadable]: true,
  length: 2
};

alert( arr.concat(arrayLike) ); // 1,2,something,else
```

5.5.2 التكرار: لكلّ forEach

يتيح لنا التابع arr.forEach تشغيل إحدى الدوال على كلّ عنصر من عناصر المصفوفة.

الصياغة:

```
arr.forEach(function(item, index, array) {
  // ... استعملها فيما تريد
});
```

مثال على عرض كلّ عنصر من عناصر المصفوفة:

```
// لكلّ عنصر، استدعِ دالة التنبيه alert
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

بينما هذه الشيفرة تحبّ الكلام الزائد ومكانها في المصفوفة المحدّدة:

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  alert(`${item} is at index ${index} in ${array}`);
});
```

ناتج التابع (لو أعادَ شيئًا أصلًا) يُهمل ويُرمى.

5.5.3 البحث في المصفوفات

أمّا الآن لنرى التوابع التي تبحث في المصفوفة.

أ. التوابع indexOf و lastIndexOf و includes

للتوابع arr.indexOf و arr.lastIndexOf و arr.includes نفس الصياغة ووظيفتها هي ذات وظيفة تلك بنسخة النصوص النصية، الفرق أنها هنا تتعامل مع العناصر بدل المحارف:

- `arr.indexOf(item, from)`: يبحث عن العنصر `item` بدءًا من الفهرس `from`، ويُعيد فهرسه حيث وجده. ولو لم يجده، يُعيد ‎-1.
- `arr.lastIndexOf(item, from)`: نفسه، ولكن البحث يبدأ من اليمين وينتهي في اليسار.
- `arr.includes(item, from)`: يبحث عن العنصر `item` بدءًا من الفهرس `from`، ويُعيد `true` إن وجدته.

مثال:

```
let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true
```

لاحظ أنّ التوابع تستعمل الموازنة بِـ `===`. لذا لو كنّا نبحث عن `false`، فستبحث هي عن `false` نفسها وليس الصفر.

لو أردت معرفة فيما كانت تحتوي المصفوفة على عنصر معيّن، ولا تريد معرفة فهرسه، فدالة `arr.includes` مناسبة لك. وهناك أيضًا أمر، تختلف `includes` عن سابقاتها `indexOf/lastIndexOf` بأنّها تتعامل مع NaN كما ينبغي:

```
const arr = [NaN];
alert( arr.indexOf(NaN) ); // يُعيد ‎-1 (الصحيح هو 0 إلّا أنّ الموازنة === لا تعمل مع NaN)
alert( arr.includes(NaN) );// true (الآن صحيح)
```

ب. البحث عبر find و findIndex

لنقل أنّ لدينا مصفوفة من الكائنات، كيف نجد الكائن حسب شرط معيّن؟

هنا يمكننا استغلال التابع arr.find(fn).

صياغته هي:

```
let result = arr.find(function(item, index, array) {
  // لو أُعيدت القيمة true، فيُعاد العنصر ويتوقّف التعداد
  // لو لم نجد ما نريد نُعيد undefined
});
```

تُستدعَى الدالة على كل عنصر من عناصر المصفوفة، واحدًا بعد الآخر:

- `item`: العنصر.
- `index`: الفهرس.
- `array`: المصفوفة نفسها.

لو أعادت `true`، يتوقّف البحث ويُعاد العنصر `item`. إن لم يوجد شيء فيُعاد `undefined`.

نرى في هذا المثال مصفوفة من المستخدمين، لكلّ مستخدم حقلان `id` و`name`. نريد الذي يتوافق مع الشرط `id == 1`:

```
let users = [
  {id: 1, name: "Ahmad"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

let user = users.find(item => item.id == 1);

alert(user.name); // Ahmad
```

في الحياة العملية، يكثُر استعمال الكائنات في المصفوفات، ولهذا فالتابع `find` مفيد جدًا لنا.

يمكنك ملاحظة بأنّا في المثال مرّرنا للتابِع `find` الدالة `item => item.id == 1` وفيها وسيط واحد. هذا طبيعي فنادرًا ما نستعمل الوُسطاء البقية في هذه الدالة.

يتشابه التابع arr.findIndex كثيرًا مع هذا، عدا على أنّه يُعيد فهرس العنصر الذي وجده بدل العنصر نفسه، ويُعيد `-1` لو لم يجد شيئًا.

ج. الترشيح filter

يبحث التابع `find` عن أوّل عنصر (واحد فقط) يُحقّق للدالة شرطها فتُعيد `true`.

لو أردت إعادة أكثر من واحد فيمكن استعمال arr.filter(fn).

تشبه صياغة `filter` التابع `find`، الفرق هو إعادته لمصفوفة بكلّ العناصر المتطابقة:

```
let results = arr.filter(function(item, index, array) {
  // لو كانت true فتُضاف القائمة إلى مصفوفة النتائج ويتواصل التكرار
  // يُعيد مصفوفة فارغة إن لم يجد شيئًا
});
```

مثال:

```
let users = [
  {id: 1, name: "Ahmad"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];
// يُعيد مصفوفة تحتوي على أوّل مستخدمَين اثنين
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

5.5.4 التعديل على عناصر المصفوفات

لنرى الآن التوابع التي تُعدّل المصفوفة وتُعيد ترتيبها.

ا. التابع map

يُعدّ التابع arr.map أكثرها استخدامًا وفائدةً أيضًا. ما يفعله هو استدعاء الدالة على كلّ عنصر من المصفوفة وإعادة مصفوفة بالنتائج.

صياغته هي:

```
let result = arr.map(function(item, index, array) {
  // يُعيد القيمة الجديدة عوض العنصر
});
```

مثلًا، هنا نعدّل كل عنصر فنحوِّله إلى طوله:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

ب. التابع sort(fn)

نُـرتّب باسـتدعاء arr.sort() المصـفوفة كمـا هي دون نسـخها فنغيّـر تـرتيب عناصـرها. هي الأخـرى تُعيـد المصفوفة المُرتّبة، ولكن غالبًا ما نُهمل القيمة المُعادة فالمصفوفة arr هي التي تتغيّر.

مثال:

```
let arr = [ 1, 2, 15 ];

// the method reorders the content of arr
arr.sort();

alert( arr );  // 1, 15, 2
```

هل لاحظت بأنّ الناتج غريب؟ صار 1, 15, 2. ليس هذا ما نريد. ولكن، لماذا؟

مبدئيًا، تُرتّب العناصر وكأنها سلاسل نصية.

بالمعنى الحرفي للكلمة: تُحوّل كل العناصر إلى سلاسل نصية عند الموازنة. والترتيب المعجماتي هو المتّبـع لترتيب السلاسل النصية، "2" > "15" صحيحة حقًا.

علينا لاستعمال الترتيب الذي نريده تمريرَ دالة تكون وسيطًا للتابِع arr.sort().

على الدالة موازنة قيمتين اثنتين (أيًا كانتا) وإعادة الناتج:

```
function compare(a, b) {
  if (a > b) return 1; // لو كانت القيمة الأولى أكبر من الثانية
  if (a == b) return 0; // لو تساوت القيمتين
  if (a < b) return -1; // لو كانت القيمة الأولى أصغر من الثانية
}
```

مثال عن الترتيب لو كانت القيم أعدادًا:

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}

let arr = [ 1, 2, 15 ];
```

```
arr.sort(compareNumeric);

alert(arr);  // 1, 2, 15
```

الآن صارت تعمل كما نريد.

لنتوقف لحظة ونفكّر فيما يحدث تمامًا. أنتّفق بأنّ المصفوفة `arr` يمكن أن تحتوي أيّ شيء؟ أيّ شيء من الأعداد أو السلاسل النصية أو الكائنات أو غيرها. كلّ ما لدينا هو "مجموعة من العناصر". لترتيبها نحتاج دالة ترتيب تعرف طريقة الموازنة بين عناصر المصفوفة. مبدئيًا، الترتيب يكون بالسلاسل النصية.

يُنفِّذ التابع `arr.sort(fn)` في طيّاته خوارزمية فرز عامّة. لا نكترث كيف تعمل هذه الخوارزمية خلف الكواليس (وهي غالبًا quicksort محسّنة)، بل نكترث بأنّها ستمرّ على المصفوفة، تُوازن عناصرها باستعمال الدالة المقدّمة أعلاه وتُعيد ترتيبها. نكترث بأن نقدّم دالة `fn` التي ستؤدّي الموازنة.

بالمناسبة، لو أردت معرفة العناصر التي تُوازنها الدالة حاليًا، فلا بأس. لن يقتلك أحد لو عرضتها:

```
[1, -2, 15, 2, 0, 8].sort(function(a, b) {
  alert( a + " <> " + b );
});
```

يمكن أن توازن الخوارزمية العنصر مع غيره من العناصر، ولكنّها تحاول قدر الإمكان تقليص عدد الموازنات.

يمكن أن تُعيد دالة الموازنة أيّ عدد

في الواقع، ليس على دالة الموازنة إلّا إعادة عدد موجب بدلالة "هذا أكبر من ذاك" وسالب بدلالة "هذا أصغر من ذاك".

يمكننا هكذا كتابة الدوال بأسطر أقل:

```
let arr = [ 1, 2, 15 ];

arr.sort(function(a, b) { return a - b; });

alert(arr);  // 1, 2, 15
```

تحيا الدوال السهمية

أتذكر الدوال السهمية من فصل تعابير الدوال والدوال السهمية؟ يمكننا استعمالها أيضًا لتبسيط شيفرة الفرز:

```
arr.sort( (a, b) => a - b );
```

لا تفرق هذه عن تلك الطويلة بشيء، البتة.

ج. العكس reverse

يعكس التابع arr.reverse ترتيب العناصر في المصفوفة arr.

مثال:

```
let arr = [1, 2, 3, 4, 5];
arr.reverse();

alert( arr ); // 5,4,3,2,1
```

كما ويُعيد المصفوفة arr بعد عكسها.

د. التقسيم split والدمج join

إليك موقفًا من الحياة العملية. تحاول الآن برمجة تطبيق مراسلة، ويُدخل المستخدم قائمة المستلمين بفاصلة بين كلّ واحد: Ahmad, Pete, Mary. ولكن لنا نحن المبرمجين، فالمصفوفة التي تحتوي الأسماء أسهل بكثير من السلسلة النصية. كيف السبيل إذًا؟

هذا ما يفعله التابع str.split(delim). يأخذ السلسلة النصية ويقسمها إلى مصفوفة حسب محرف القاسِم delim المقدّم.

في المثال أعلاه نقسم حسب "فاصلة بعدها مسافة":

```
let names = 'Bilbo, Gandalf, Nazgul';

let arr = names.split(', ');

for (let name of arr) {
  alert( `A message to ${name}.` ); // A message to Bilbo  (والبقية)
}
```

للتابِع split وسيطًا عدديًا اختياريًا أيضًا، وهو يحدّ طول المصفوفة. لو قدّمته فستُهمل العناصر الأخرى. ولكن في الواقع العملي، نادرًا ما ستفيدك هذا:

```
let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(', ', 2);

alert(arr); // Bilbo, Gandalf
```

التقسيم إلى أحرف

لو ناديت `split(s)` وتركت s فارغًا فستُقسم السلسلة النصية إلى مصفوفة من الأحرف:

```
let str = "test";

alert( str.split('') ); // t,e,s,t
```

على العكس من `split` فنداء arr.join(glue) يُنشئ سلسلة نصية من عناصر `arr` مجموعةً معًا "باللاصق". مثال:

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];

// جمع عناصر المصفوفة في سلسلة نصية بإضافة ؛ بينها
let str = arr.join(';');

alert( str ); // Bilbo;Gandalf;Nazgul
```

ه. التابعان reduce و reduceRight

متى ما أردنا أن نمرّ على عناصر المصفوفة، استعملنا `forEach` أو `for` أو `for..of`. ومتى ما أردنا أن نمرّ ونُعيد بيانات كلّ عنصر، استعملنا `map`. نفس الحال مع التابعين arr.reduce وarr.reduceRight، إلّا أنهما ليسا بالسهولة نفسها. يُستعمل هذان التابعان لحساب قيمة واحدة بناءً على عناصر المصفوفة.

هذه الصياغة:

```
let value = arr.reduce(function(previousValue, item, index, array) {
  // ...
}, [initial]);
```

تُطبّق الدالة على كل عناصر المصفوفة واحدًا بعد الآخر، و"تنقل" النتيجة إلى النداء التالي لها:

وُسطاء الدالة:

- `previousValue` - نتيجة النداء السابق للدالة. يُساوي قيمة `initial` في أوّل نداء (لو قُدّمت أصلًا).
- `item`: العنصر الحالي في المصفوفة.
- `Index`: مكان العنصر.
- `array`: المصفوفة نفسها.

حين تُطبّق الدالة، تُمرّر إليها نتيجة النداء السابق في أوّل وسيط. أجل، معقّد قليلًا، لكن ليس كما تتخيّل لـو قلنا أنّ الوسيط الأول بمثابة "ذاكرة" تخزّن النتيجة النهائية من إجراءات التنفيذ التي سبقتها، وفي آخر نداء تصير نتيجة التابع reduce.

ربّما نقدّم مثالًا لتسهيل المسألة. هنا نعرف مجموعة عناصر المصفوفة في سطر برمجي واحد:

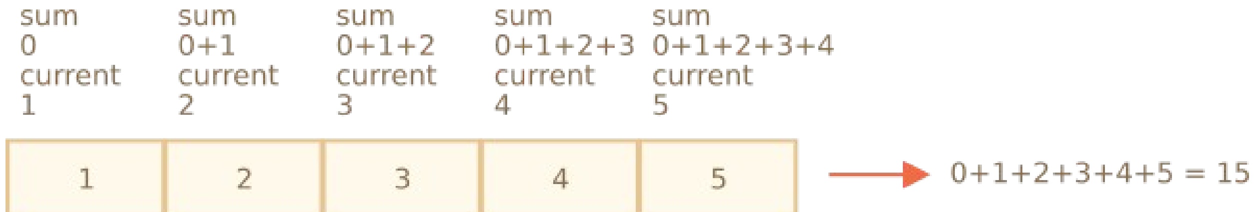
```
let arr = [1, 2, 3, 4, 5];

let result = arr.reduce((sum, current) => sum + current, 0);

alert(result); // 15
```

الدالة المُمرّرة إلى reduce تستعمل وسيطين اثنين فقط، وهذا كافٍ عادةً.

لنرى تفاصيل النداءات.

1. في أوّل مرّة، قيمة sum هي قيمة initial (آخـر وسـيط في reduce) وتسـاوي 0، وcurrent هي أوّل عنصر في المصفوفة وتساوي 1. إذًا فناتج الدالة هو 1.
2. في النداء التالي، sum = 1 ونُضيف العنصر الثاني في المصفوفة (2) ونُعيد القيمة.
3. في النداء الثالث، sum = 3، ونُضيف العنصر التالي في المصفوفة، وهكذا دواليك إلى آخر نداء…

هذا سير العملية الحسابية:



وهكذا نمثّلها في جدول (كلّ صف يساوي نداء واحد للدالة على العنصر التالي في المصفوفة):

	sum	current	الناتج
أوّل نداء	0	1	1
ثاني نداء	1	2	3
ثالث نداء	3	3	6
رابع نداء	6	4	10
خامس نداء	10	5	15

هكذا نرى بوضوح شديد كيف يصير ناتج النداء السابق أوّل وسيط في النداء الذي يلحقه.

يمكننا أيضًا حذف القيمة الأولية:

```
let arr = [1, 2, 3, 4, 5];

// (0 القيمة اختفت) reduce أزلنا القيمة الأولية من التابع
let result = arr.reduce((sum, current) => sum + current);

alert( result ); // 15
```

وستكون النتيجة متطابقة، إذ أنّ `reduce` تأخذ أول عنصر من المصفوفة على أنّه قيمة أولية (لو لم نقدّم نحن قيمة أولية) وتبدأ العملية من العنصر الثاني.

جدول العملية الحسابية مُطابق للجدول أعلاه، لو حذفنا أول سطر فيه. ولكن عليك أن تحترس حين لا تقدّم تلك القيمة. لو كانت المصفوفة فارغة فنداء `reduce` بدون القيمة الأولية سيعطيك خطأً.

مثال على ذلك:

```
let arr = [];

arr.reduce((sum, current) => sum + current);
```

الشيفرة السابقة ستطلق خطأً، إذ لا يمكن استدعاء `reduce` مع مصفوفة فارغة دون قيمة أولية، وتحل المشكلة بتوفير قيمة أولية، وستعاد آنذاك. لذا خُذ هذه النصيحة وحدّد قيمة أولية دومًا.

لا يختلف التابع arr.reduceRight عن هذا أعلاه إلا بأنّه يبدأ من اليمين وينتهي على اليسار.

5.5.5 التابع Array.isArray

المصفوفات ليست نوعًا منفصلًا في اللغة، بل هي مبنيّة على الكائنات. لذا `typeof` لن تفيدك في التفريق بين الكائن العادي والمصفوفة:

```
alert(typeof {}); // object كائن
alert(typeof []); // كائن أيضًا
```

...ولكن، المصفوفات تستعمل كثيرًا جدًا لدرجة تقديم تابِع خاص لهذا الغرض: Array.isArray(value). يُعيد هذا التابع `true` لو كانت `value` مصفوفة حقًا، و`false` لو لم تكن.

```
alert(Array.isArray({})); // false

alert(Array.isArray([])); // true
```

5.5.6 تدعم أغلب التوابع thisArg

تقبل أغلب توابِع المصفوفات تقريبًا، التوابع التي تستدعي دوال (مثل `find` و`filter` و`map`، عدا `sort`) - تقبل المُعامل الاختياري `thisArg`. لم نشرح هذا المُعامل في الأقسام أعلاه إذ أنّه نادرًا ما يُستعمل. ولكن علينا الحديث عنه لألا يكون الشرح ناقصًا.

هذه الصياغة الكاملة لهذه التوابع:

```
arr.find(func, thisArg);
arr.filter(func, thisArg);
arr.map(func, thisArg);
// ...
// الوسيط thisArg هو آخر وسيط اختياري
```

تكون قيمة المُعامل `thisArg` للدالة `func` تساوي `this`. هنا مثلًا نستعمل تابِع كائن `army` على أنّه مرشّح، والوسيط `thisArg` يمرّر سياق التنفيذ وذلك لإيجاد المستخدمين الذين يعيد التابع `army.canJoin` القيمة `true`:

```
let army = {
  minAge: 18,
  maxAge: 27,
  canJoin(user) {
    return user.age >= this.minAge && user.age < this.maxAge;
  }
};

let users = [
  {age: 16},
  {age: 20},
  {age: 23},
  {age: 30}
];
let soldiers = users.filter(army.canJoin, army);

alert(soldiers.length); // 2
alert(soldiers[0].age); // 20
alert(soldiers[1].age); // 23
```

لو استعملنا في المثال أعلاه `users.filter(army.canJoin)` فسيُستدعى التابع `army.canJoin` كدالة مستقلة بذاتها حيث `this=undefined`، ما سيؤدي إلى خطأ.

يمكن استبدال استدعاء `users.filter(army.canJoin, army)` بالتعليمة التي تُؤدّي ذات الغرض `users.filter(user => army.canJoin(user))`. نستعمل الأولى أكثر من الثانية إذ أنّ الناس تفهمها أكثر من تلك.

5.5.7 الخلاصة

ورقة فيها كل توابِع المصفوفات (غُشّ منها):

- لإضافة العناصر وإزالتها:
  - `push(...items)` - تُضيف العناصر `items` إلى النهاية،
- مصفوفة جديدة: انسخ كل عناصر المصفوفة الحالية وأضِف إليها العناصر `items`. لو كانت واحدة من عناصر `items` مصفوفة أيضًا، فستُنسخ عناصرها بدل.
- لتبحث عن العناصر:
  - `indexOf/lastIndexOf(item, pos)` - ابحث عن العنصر `item` بدءًا من العنصر ذي الفهرس `pos` وأعِد فهرسه أو أعِد `-1` لو لم تجده.
  - `includes(value)` - أعِد القيمة `true` لو كان العنصر `value` في المصفوفة، وإلا أعِد `false`.
  - `find/filter(func)` - رشّح العناصر عبر دالة وأعِد أوّل قيمة (أو كل القيم) التي تُعيد الدالة قيمة `true` لو مُرّر ذلك العنصر لها.
  - `findIndex` - يشبه `find`، ولكن يُعيد الفهرس بدل القيمة.
- للمرور على عناصر المصفوفة:
  - `forEach(func)` - يستدعي `func` لكلّ عنصر ولا يُعيد أيّ شيء.
- لتعديل عناصر المصفوفة:
  - `map(func)` - أنشِئ مصفوفة جديدة من نتائج استدعاء `func` لكلّ من عناصر المصفوفة.
  - `sort(func)` - افرز المصفوفة كما هي وأعِد ناتج الفرز.
  - `reverse()` - اعكس عناصر المصفوفة كما هي وأعِد ناتج العكس.
  - `split/join` - حوّل المصفوفة إلى سلسلة نصية، والعكس أيضًا.

- `reduce(func, initial)` - احسب قيمة من المصفوفة باستدعاء `func` على كلّ عنصر فيها وتمرير الناتج بين كلّ استدعاء وآخر.

- وأيضًا:
  - `Array.isArray(arr)` - يفحص لو كانت `arr` مصفوفة أم لا.

لاحظ أنّ التوابع `sort` و`reverse` و`splice` تُعدّل المصفوفة نفسها.

هذه التوابع أعلاه هي أغلب ما تحتاج وما تريد أغلب الوقت (99.99%). ولكن هناك طبعًا غيرها:

- arr.some(fn)/arr.every(fn) - تفحص المصفوفة. تُنادى الدالة `fn` على كلّ عنصر من المصفوفة (مثل `map`). لو كانت أيًا من (أو كل) النتائج `true`، فيُعيد `true`، وإلا يُعيد `false`.
- arr.fill(value, start, end) - يملأ المصفوفة بالقيمة المتكرّرة `value` من الفهرس `start` إلى الفهرس `end`.
- arr.copyWithin(target, start, end) - ينسخ العناصر من العنصر ذا الفهرس `start` إلى ذا الفهرس `end` ويلصقها "داخلها" عند الفهرس `target` (تعوّض ما هو موجود مكانها في المصفوفة).

طالِع هذه الصفحة ففيها كل التوابع المتعلقة بالمصفوفات. من أول وهلة سترى بأنّ عدد التوابع لا ينتهي ومهمة حفظها مستحيلة، ولكن الواقع هي أنّها بسيطة جدًا.

طالِع "ورقة الغشّ" لتعرف ما تفعل كلًا منها، ثمّ حُلّ مهام هذا الفصل لتتدرّب عليها وتكون خبيرًا كفاية بتوابِع الدوال. بعدها، لو احتجت التعامل مع المصفوفات ولا تدري ما تفعل، تعال هنا وابحث في ورقة الغشّ عن التابع المناسب لحالتك. الأمثلة الموجودة ستفيدك فتكتبها كما ينبغي. وسريعًا ما ستتذكّر كل التوابع من تلقاء نفسك ودون بذل أيّ جهد.

5.5.8 تمارين

ا. حول border-left-width إلى borderLeftWidth

الأهمية: ★★★★★

اكتب دالة `camelize(str)` تغيّر الكلمات المقسومة بِشَرطات مثل "my-short-string" إلى عبارات بتنسيق "سنام الجمل": "myShortString".

بعبارة أخرى: أزِل كلّ الشرطات وحوّل أوّل حرف من كلّ كلمة بعدها إلى الحالة الكبيرة.

أمثلة:

```
camelize("background-color") == 'backgroundColor';
```

```
camelize("list-style-image") == 'listStyleImage';
camelize("-webkit-transition") == 'WebkitTransition';
```

تلميح: استعمل split لتقسيم السلسلة النصية إلى مصفوفة، ثمّ عدّل عناصرها وأعِد ربطها بتابِع join.

(إليك الشيفرة البدائية)

الحل:

```
function camelize(str) {
  return str
    .split('-') // splits 'my-long-word' into array ['my', 'long', 'word']
    .map(
      // كبر الحروف الأولى لجميع عناصر المصفوفة باستثناء أول عنصر
      // ['my', 'long', 'word'] --> ['my', 'Long', 'Word']
      (word, index) => index == 0 ? word : word[0].toUpperCase() + word.slice(1)
    )
    .join(''); // joins ['my', 'Long', 'Word'] into 'myLongWord'
}
```

(إليك الحل في بيئة تجريبية)

ب. نطاق ترشيح

الأهمية: ★★★★☆

اكتب دالة filterRange(arr, a, b) تأخذ المصفوفة arr، وتبحث في عناصرها بين a وb وتُعيد مصفوفة بها. يجب ألّا تُعدّل الدالة المصفوفة، بل إعادة مصفوفة جديدة.

مثال:

```
let arr = [5, 3, 8, 1];

let filtered = filterRange(arr, 1, 4);

alert( filtered ); // 3,1

alert( arr ); // 5,3,8,1
```

(إليك الشيفرة البدائية)

الحل:

```
function filterRange(arr, a, b) {
  // أضفنا الأقواس حول التعبير لتسهيل القراءة
  return arr.filter(item => (a <= item && item <= b));
}

let arr = [5, 3, 8, 1];

let filtered = filterRange(arr, 1, 4);

alert( filtered ); // 3,1

alert( arr ); // 5,3,8,1
```

(إليك الحل في بيئة تجريبية)

ج. نطاق ترشيح "كما هو"

الأهمية: ★★★★☆

اكتب دالة `filterRangeInPlace(arr, a, b)` تأخذ المصفوفة `arr` وتُزيل منها كل القيم عدا تلك بين `a` و`b`. الشرط هو: `a ≤ arr[i] ≤ b`.

يجب أن تُعدّل الدالة المصفوفة، ولا تُعيد شيئًا.

مثال:

```
let arr = [5, 3, 8, 1];

// حذف جميع الأعداد باستثناء الواقعة بين 1 و 4
filterRangeInPlace(arr, 1, 4);

alert( arr ); // [3, 1]
```

(إليك الشيفرة البدائية)

الحل:

```
function filterRangeInPlace(arr, a, b) {
```

```
  for (let i = 0; i < arr.length; i++) {
    let val = arr[i];

    // إزالة إن كانت خارج النطاق
    if (val < a || val > b) {
      arr.splice(i, 1);
      i--;
    }
  }

}

let arr = [5, 3, 8, 1];

filterRangeInPlace(arr, 1, 4); // removed the numbers except from 1 to
4

alert( arr ); // [3, 1]
```

(إليك الحل في بيئة تجريبية)

د. الفرز بالترتيب التنازلي

الأهمية: ★★★★☆

```
let arr = [5, 2, 1, -10, 8];

// ... شيفرة ترتيب العناصر تنازليًا

alert( arr ); // 8, 5, 2, 1, -10
```

الحل:

```
let arr = [5, 2, 1, -10, 8];

arr.sort((a, b) => b - a);

alert( arr );
```

ه. نسخ المصفوفة وفرزها

الأهمية: ★★★★★

في يدنا مصفوفة من السلاسل النصية arr. نريد نسخة مرتّبة عنها وترك arr بلا تعديل.

أنشِئ دالة copySorted(arr) تُعيد هذه النسخة.

```
let arr = ["HTML", "JavaScript", "CSS"];

let sorted = copySorted(arr);

alert( sorted ); // CSS, HTML, JavaScript
alert( arr ); // HTML, JavaScript, CSS
```

الحل:

يمكن أن نستعمل slice() لأخذ نسخة ونفرز المصفوفة:

```
function copySorted(arr) {
  return arr.slice().sort();
}

let arr = ["HTML", "JavaScript", "CSS"];

let sorted = copySorted(arr);

alert( sorted );
alert( arr );
```

و. خارطة بالأسماء

الأهمية: ★★★★★

لدينا مصفوفة من كائنات user، لكلّ منها صفة user.name. اكتب شيئًا يحوّلها إلى مصفوفة من الأسماء.

مثال:

```
let john = { name: "Ahmad", age: 25 };
let pete = { name: "Pete", age: 30 };
```

```
let mary = { name: "Mary", age: 28 };

let users = [ john, pete, mary ];

let names = /* شيفرتك هنا */

alert( names ); // Ahmad, Pete, Mary
```

الحل:

```
let john = { name: "Ahmad", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let users = [ john, pete, mary ];

let names = users.map(item => item.name);

alert( names ); // Ahmad, Pete, Mary
```

ز. أنشِئ آلة حاسبة يمكن توسعتها لاحقًا

الأهمية: ★★★★★

أنشِئ دالة إنشاء باني "constructor" `Calculator` تُنشئ كائنات من نوع "آلة حاسبة" يمكن لنا "توسعتها".

تنقسم هذه المهمة إلى جزأين اثنين:

أولًا، نفّذ تابِع `calculate(str)` يأخذ سلسلة نصية (مثل `"1 + 2"`) بالتنسيق "عدد مُعامل عدد" (أي مقسومة بمسافات) ويُعيد الناتج. يجب أن يفهم التابع الجمع `+` والطرح `-`.

مثال عن الاستعمال:

```
let calc = new Calculator;

alert( calc.calculate("3 + 7") ); // 10
```

بعدها أضِف تابِع `addMethod(name, func)` يُعلّم الآلة الحاسبة عمليّة جديدة. يأخذ التابع المُعامل `name` ودالة `func(a,b)` بوسيطين تُنفّذ هذه العملية.

كمثال على ذلك سنُضيف عمليات الضرب `*` والقسمة `/` والأُسّ `**`:

```
let powerCalc = new Calculator;
powerCalc.addMethod("*", (a, b) => a * b);
powerCalc.addMethod("/", (a, b) => a / b);
powerCalc.addMethod("**", (a, b) => a ** b);

let result = powerCalc.calculate("2 ** 3");
alert( result ); // 8
```

- في هذه المهمة ليس هناك أقواس رياضية أو تعابير معقّدة.
- تفصل الأعداد والمُعامل مسافة واحدة فقط.
- يمكنك التعامل مع الأخطاء لو أردت.

(إليك الشيفرة البدائية)

الحل:

- لاحظ طريقة تخزين التوابع، حيث تُضاف إلى صفة `this.methods` فقط.
- كلّ الشروط والتحويلات العددية موجودة في التابع `calculate`. يمكننا في المستقبل توسيعه ليدعم تعابير أكثر تعقيدًا.

```
function Calculator() {

  this.methods = {
    "-": (a, b) => a - b,
    "+": (a, b) => a + b
  };

  this.calculate = function(str) {

    let split = str.split(' '),
      a = +split[0],
      op = split[1],
```

```
      b = +split[2]

    if (!this.methods[op] || isNaN(a) || isNaN(b)) {
      return NaN;
    }

    return this.methods[op](a, b);
  }

  this.addMethod = function(name, func) {
    this.methods[name] = func;
  };
}
```

(إليك الحل في بيئة تجريبية)

ح. المرور على خاصيات كائن

الأهمية: ★★★★★

لدينا مصفوفة من كائنات تمثل مستخدمين، ولكل كائن اسمًا (الخاصية name)، ومهمتك هي تحويل تلك الكائنات التي تمثِّل المستخدمين إلى الأسماء المقابلة لها. مثال:

```
let john = { name: "Ahmad", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let users = [ john, pete, mary ];

let names = /* ... your code */

alert( names ); // Ahmad, Pete, Mary
```

الحل:

```
let john = { name: "Ahmad", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };
let users = [ john, pete, mary ];
```

```
let names = users.map(item => item.name);

alert( names ); // Ahmad, Pete, Mary
```

ط. تحويل مصفوفة إلى شكل آخر

الأهمية: ★★★★★

لدينا مصفوفة من كائنات تمثل مستخدمين، ولكل كائن الخاصيات التالية: name و surname و id، ومهمتك هي تحويل تلك الكائنات التي تمثِّل المستخدمين إلى كائنات مقابلة لها تملك خاصيات فقط هما id و fullName حيث الأخيرة مولدة من دمج الخاصية name مع surname. مثال:

```
let john = { name: "Ahmad", surname: "Smith", id: 1 };
let pete = { name: "Pete", surname: "Hunt", id: 2 };
let mary = { name: "Mary", surname: "Key", id: 3 };

let users = [ john, pete, mary ];

let usersMapped = /* ... your code ... */

/*
usersMapped = [
  { fullName: "Ahmad Smith", id: 1 },
  { fullName: "Pete Hunt", id: 2 },
  { fullName: "Mary Key", id: 3 }
]
*/

alert( usersMapped[0].id ) // 1
alert( usersMapped[0].fullName ) // Ahmad Smith
```

الحل:

```
let john = { name: "Ahmad", surname: "Smith", id: 1 };
let pete = { name: "Pete", surname: "Hunt", id: 2 };
let mary = { name: "Mary", surname: "Key", id: 3 };
let users = [ john, pete, mary ];
```

```
let usersMapped = users.map(user => ({
  fullName: `${user.name} ${user.surname}`,
  id: user.id
}));

/*
usersMapped = [
  { fullName: "Ahmad Smith", id: 1 },
  { fullName: "Pete Hunt", id: 2 },
  { fullName: "Mary Key", id: 3 }
]
*/

alert( usersMapped[0].id ); // 1
alert( usersMapped[0].fullName ); // Ahmad Smith
```

انتبه إلى الأقواس المستعملة مع الدالة السهمية في map، فلا يمكننا مثلًا كتابة:

```
let usersMapped = users.map(user => {
  fullName: `${user.name} ${user.surname}`,
  id: user.id
});
```

فكما تذكر، هنالك صياغتان للدوال السهمية، الأولى دون أقواس value => expr والثانية مع أقواس معقوصة {...} <= value، فإن اقتصر الحل على الصياغة الثانية، فستعد JavaScript القوس } الأول عندما تراه أنه بداية جسم الدالة السهمية وليس بداية الكائن المراد إعادته، لذا وجب تغليف الكائن بالأقواس الهلالية () لحل هذه المشكلة.

```
let usersMapped = users.map(user => ({
  fullName: `${user.name} ${user.surname}`,
  id: user.id
}));
```

ي. فرز المستخدمين حسب أعمارهم

الأهمية: ★★★★★

اكتب دالة `sortByAge(users)` تأخذ مصفوفة من الكائنات بالصفة `age` وتُرتبّها حسب أعمارهم `age`.

مثال:

```
let john = { name: "Ahmad", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [ pete, john, mary ];

sortByAge(arr);

// [john, mary, pete]
alert(arr[0].name); // Ahmad
alert(arr[1].name); // Mary
alert(arr[2].name); // Pete
```

الحل:

```
function sortByAge(arr) {
  arr.sort((a, b) => a.age > b.age ? 1 : -1);
}

let john = { name: "Ahmad", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [ pete, john, mary ];

sortByAge(arr);

// [john, mary, pete]
alert(arr[0].name); // Ahmad
alert(arr[1].name); // Mary
alert(arr[2].name); // Pete
```

ك. خلط المصفوفات

الأهمية: ★★★☆☆

اكتب دالة `shuffle(array)` تخلط عناصر المصفوفة (أي ترتّبها عشوائيًا).

يمكن بتكرار نداء `shuffle` إعادة العناصر بترتيب مختلف. مثال:

```
let arr = [1, 2, 3];

shuffle(arr);
// arr = [3, 2, 1]

shuffle(arr);
// arr = [2, 1, 3]

shuffle(arr);
// arr = [3, 1, 2]
// ...
```

يجب أن تكون جميع احتمالات ترتيب العناصر متساوية. فمثلًا يمكن إعادة ترتيب `[1,2,3]` لتكون `[1,2,3]` أو `[1,3,2]` أو `[3,1,2]` أو ...إلخ، واحتمال حدوث كلّ حالة متساوٍ.

الحل:

هذا هو الحل البسيط:

```
function shuffle(array) {
  array.sort(() => Math.random() - 0.5);
}

let arr = [1, 2, 3];
shuffle(arr);
alert(arr);
```

تعمل هذه الشيفرة (نوعًا ما) إذ أنّ `Math.random() - 0.5` عددٌ عشوائي ويمكن أن يكون موجبًا أم سالبًا، بذلك تُعيد دالة الفرز ترتيب العناصر عشوائيًا.

ولكن ليس هذه الطريقة التي تعمل فيها دوال الفرز، إذ ليس لكلّ حالات التبديل الاحتمال نفسه. فمثلًا في الشيفرة أعلاه، تُنفّذ `shuffle` 1000000 مرّة وتعدّ مرّات ظهور النتائج الممكنة كلّها:

```
function shuffle(array) {
  array.sort(() => Math.random() - 0.5);
}

// نعدّ مرّات ظهور كلّ عمليات التبديل الممكنة
let count = {
  '123': 0,
  '132': 0,
  '213': 0,
  '231': 0,
  '321': 0,
  '312': 0
};

for (let i = 0; i < 1000000; i++) {
  let array = [1, 2, 3];
  shuffle(array);
  count[array.join('')]++;
}

// نعرض عدد عمليات التبديل الممكنة
for (let key in count) {
  alert(`${key}: ${count[key]}`);
}
```

إليك عيّنة عن الناتج (إذ يعتمد على محرّك جافاسكربت):

```
250706
124425
249618
124880
125148
125223
```

نرى تحيّز الشيفرة بوضوح شديد، إذ تظهر `123` و`213` أكثر بكثير من البقية. يختلف ناتج هذه الشيفرة حسب محرّكات JavaScript ولكن هذا يكفي لنقول بأنّ هذه الطريقة ليست موثوقة.

ولكن لمَ لا تعمل الشيفرة؟ بشكل عام فتابِع `sort` أشبه "بالصندوق الأسود": نرمي فيه مصفوفة ودالة موازنة وننتظر أن تفرز لنا المصفوفة. ولكن بسبب عشوائية الموازنة يختلّ ذكاء الصندوق الأسود، وهذا الاختلال يعتمد على طريقة كتابة كلّ محرّك للشيفرة الخاصة به.

ثمّة طرق أخرى أفضل لهذه المهمّة، مثل الخوارزمية خلّاط فِشر ييتس الرائعة. فكرتها هي المرور على عناصر المصفوفة بالعكس وتبديل كلّ واحد بآخر قبله عشوائيًا:

```
function shuffle(array) {
  for (let i = array.length - 1; i > 0; i--) {
   let j = Math.floor(Math.random() * (i + 1)); // رقم عشوائي من 0 إلى i

   // let t = array[i]; array[i] = array[j]; array[j] = t
   [array[i], array[j]] = [array[j], array[i]];
  }
}
```

بدلنا بالمثال هذا العنصرين `array[i]` و `array[j]` وذلك نستعمل صياغة "الإسناد بالتفكيك" وستجد تفاصيل أكثر عن هذه الصياغة في فصول لاحقة.

لنختبر الطريقة هذه بنفس ما اختبرنا تلك:

```
function shuffle(array) {
  for (let i = array.length - 1; i > 0; i--) {
   let j = Math.floor(Math.random() * (i + 1));
   [array[i], array[j]] = [array[j], array[i]];
  }
}

// نعدّ مرّات ظهور كلّ عمليات التبديل الممكنة
let count = {
  '123': 0,
  '132': 0,
  '213': 0,
  '231': 0,
  '321': 0,
  '312': 0
};
```

```
for (let i = 0; i < 1000000; i++) {
  let array = [1, 2, 3];
  shuffle(array);
  count[array.join('')]++;
}

// نعرض عدد عمليات التبديل الممكنة
for (let key in count) {
  alert(`${key}: ${count[key]}`);
}
```

عيّنة عن الناتج:

```
166693
166647
166628
167517
166199
166316
```

الآن كل شيء سليم: لكلّ عمليات التبديل ذات الاحتمال. كما أنّ خوارزمية "فِشر ييتس" أفضل من ناحية الأداء إذ ليس علينا تخصيص الموارد لعملية "الفرز".

ل. ما متوسّط الأعمار؟

الأهمية: ★★★★☆

اكتب دالة `getAverageAge(users)` تأخذ مصفوفة من كائنات لها الصفة `age` وتُعيد متوسّط الأعمار.

معادلة المتوسّط: `(age1 + age2 + ... + ageN) / N`.

مثال:

```
let john = { name: "Ahmad", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 29 };

let arr = [ john, pete, mary ];

alert( getAverageAge(arr) ); // (25 + 30 + 29) / 3 = 28
```

الحل:

```
function getAverageAge(users) {
  return users.reduce((prev, user) => prev + user.age, 0) / users.length;
}

let john = { name: "Ahmad", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 29 };

let arr = [ john, pete, mary ];

alert( getAverageAge(arr) ); // 28
```

م. ترشيح العناصر الفريدة في المصفوفة

الأهمية: ★★★★☆

لمّا أنّ `arr` مصفوفة، أنشِئ دالة `unique(arr)` تُعيد مصفوفة فيها عناصر `arr` غير مكرّرة.

مثال:

```
function unique(arr) {
  /* شيفرة هنا */
}

let strings = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-O"
];

alert( unique(strings) ); // Hare, Krishna, :-O
```

(إليك الشيفرة البدائية)

الحل:

ما سنفعل هو المرور على عناصر المصفوفة:

- سنفحص كلّ عنصر ونرى إن كان في المصفوفة الناتجة.
- إن كان كذلك... نُهمله، وإن لم يكن، نُضيفه إلى المصفوفة.

```
function unique(arr) {
  let result = [];

  for (let str of arr) {
    if (!result.includes(str)) {
      result.push(str);
    }
  }

  return result;
}

let strings = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-O"
];

alert( unique(strings) ); // Hare, Krishna, :-O
```

صحيح أنّ الكود يعمل، إلّا أنّ فيه مشكلة أداء محتملة. خلف الكواليس، يمرّ التابع `result.includes(str)` على المصفوفة `result` ويوازن كلّ عنصر مع `str` ليجد المطابقة المنشودة. لذا لو كان في `result` مئة `100` عنصر وما من أيّ مطابقة مع `str`، فعليها المرور على جُلّ `result` وإجراء `100` حالة مقارنة كاملة. ولو كانت `result` كبيرة مثل `10000` فيعني ذلك `10000` حالة موازنة.

إلى هنا لا مشكلة، لأنّ محرّكات JavaScript سريعة جدًا، والمرور على `1000` عنصر في المصفوفة يحدث في بضعة ميكروثوان. ولكنّنا هنا في حلقة `for` نُجري هذه الشروط لكلّ عنصر من `arr`. فإن كانت `arr.length` تساوي `10000` فيعني أنّا سنُجري `10000*10000` = مئة مليون حالة مقارنة. كثير جدًا.

إذًا، فهذا الحل ينفع للمصفوفات الصغيرة فقط. سنرى لاحقًا في فصل النوع Map والنوع Set كيف نحسّن هذا الكود.

(إليك الحل في بيئة تجريبية)

ن. إنشاء كائن من مصفوفة

الأهمية: ★★★★☆

لنفترض استلام مصفوفة من مستخدمين (كائنات) بالشكل: {id:..., name:..., age...}. أنشئ دالة باسم groupById(arr) تعمل على توليد كائن من تلك المصفوفة بحيث تكون المفـاتيح فيـه هي قيمـة الخاصـية id والقيمة المرتبطة فيها هي الكائن المقابل. انظر مثلًا لتوضيح ذلك:

```
let users = [
  {id: 'john', name: "Ahmad Smith", age: 20},
  {id: 'ann', name: "Ann Smith", age: 24},
  {id: 'pete', name: "Pete Peterson", age: 31},
];

let usersById = groupById(users);
/*
// ستكون بعد الاستدعاء
usersById = {
  john: {id: 'john', name: "Ahmad Smith", age: 20},
  ann: {id: 'ann', name: "Ann Smith", age: 24},
  pete: {id: 'pete', name: "Pete Peterson", age: 31},
}
*/
```

سترى مثل هذه الدالة عند العمل على البيانات القادمة من الخادم.

سنفترض في حالتنا هذه أن المفاتيح id فريدة وغير مكررة. استعمل التابع reduce في الحل.

(إليك الشيفرة البدائية)

الحل:

```
function groupById(array) {
  return array.reduce((obj, value) => {
    obj[value.id] = value;
    return obj;
  }, {})
}
```

(إليك الحل في بيئة تجريبية)

5.6 المُكرَّرات Iterables

الكائنات المُكرَّرة Iterables هي مفهوم أعمّ من المصفوفات. تتيح لنا هذه الكائنات تحويل أيّ كائن إلى "كائن يمكن تكرار عملية على عناصره" أي يمكن المرور على عناصره واحدًا تلو الآخر فيمكننا استعماله في حلقة `for..of`. بالطبع فالمصفوفات يمكن تكرار عملية على عناصرها، ولكن هناك كائنات أخرى (مضمّنة في أصل اللغة) يمكن تكرار عملية عليها أيضًا، مثل السلاسل النصية.

لو لم يكن الكائن مصفوفة تقنيًا، ولكن يمكننا تمثيله على أنّه تجميعة من العناصر (النوع `list`، والنوع `set`)، فصياغة `for..of` ممتازة لنمرّ على عناصره. لذا دعنا نرى كيف يمكن توظيف هذه "المُكرَّرات".

5.6.1 Symbol.iterator

يمكن لنا أن نُدرك هذا المفهوم -مفهوم المُكرَّرات أعني- بأن نصنع واحدًا بنفسنا. لنقل أنّ لدينا كائن وهو ليس بمصفوفة بأيّ شكل، ولكن يمكن توظيفه لحلقة `for..of`. مثلًا كائن المدى هذا `range` يُمثّل مجموعة متتالية من الأعداد.

```
let range = {
  from: 1,
  to: 5
};

// نريد أن تعمل for..of هكذا:
// for(let num of range) ... num=1,2,3,4,5
```

لنُضيف خاصية التكرار إلى `range` (فتعمل بهذا `for..of`)، علينا إضافة تابِع إلى الكائن بالاسم `Symbol.iterator` (وهو رمز خاصّ في اللغة يتيح لنا هذه الميزة).

1. حين تبدأ `for..of`، تنادي ذلك التابع مرة واحدة (أو تعرض الأخطاء المعروفة لو لم تجدها). على هذا التابع إعادة مُكرِّر/iterator، أي كائنًا له التابع `next`.
2. بعدها، تعمل `for..of` مع ذلك الكائن المُعاد فقط لا غير.
3. حين تحتاج `for..of` القيمة التالية، تستدعي `next()` لذاك الكائن.
4. يجب أن يكون ناتج `next()` بالشكل هذا `{done: Boolean, value: any}`، حيث لو كانت `done=true` فيعني أن التكرار اكتمل، وإلّا فقيمة `value` هي التالية.

إليك النص الكامل لتنفيذ كائن `range` (مع الملاحظات):

```
let range = {
  from: 1,
  to: 5
};

// 1. حين نادي for..of فهي تنادي هذه
range[Symbol.iterator] = function() {

  // ...وتُعيد الكائن المُكرِّر:
  // 2. بعد ذلك تعمل for..of مع هذا المُكرِّر، طالبةً منه القيم التالية
  return {
    current: this.from,
    last: this.to,

    // 3. يُستدعى next() في كلّ كرّة في حلقة for..of
    next() {

      // 4. يجب أن يُعيد القيمة كائنًا كهذا {done:.., value :...}
      if (this.current <= this.last) {
        return { done: false, value: this.current++ };
      } else {
        return { done: true };
      }
    }
  };
};

// والآن تعمل!
for (let num of range) {
  alert(num); // 1، ثمّ 2 فَ 3 فَ 4 فَ 5
}
```

لاحظ الميزة الأساس للمُكرَّرات: فصل الاهتمامات.

- عنصر `range` ذاته ليس له التابع `next()`.
- بدل ذلك، يُنشأ كائن آخر (أي "المُكرَّر") عند استدعاء `range[Symbol.iterator]()`، وتابِعه `next()` يُولّد قيم التكرار.

الخلاصة هي أنّ الكائن المُكرَّر منفصل عن الكائن الذي يُكرِّره هذا المُكرَّر (الجملة واضحة، صحيح؟ :-]).

نظريًا، يمكننا دمجهما معًا واستعمال كائن `range` نفسه مُكرِّرًا (iterator) لتبسيط الكود أكثر. هكذا تمامًا:

```
let range = {
  from: 1,
  to: 5,
  [Symbol.iterator]() {
    this.current = this.from;
    return this;
  },
  next() {
    if (this.current <= this.to) {
      return { done: false, value: this.current++ };
    } else {
      return { done: true };
    }
  }
};

for (let num of range) {
  alert(num); // 1، ثمّ 2 فَ 3 فَ 4 فَ 5
```

الآن صار يُعيد `range[Symbol.iterator]()` كائن `range` نفسه: وهذا الكائن فيه تابِع `next()` اللازم كما ويتذكّر حالة التعداد الحالية في `this.current`. الشيفرة أبسط، أجل. وأحيانًا لا بأس به هكذا.

المشكلة هنا هي استحالة وجود حلقتي `for..of` تعملان على الكائن في آن واحد، إذ سيتشاركان حالة التكرار؛ فليس هناك إلا مُكرِّرًا واحدًا: الكائن نفسه. لكن أصلًا وجود حلقتي for..of نادر، حتى في العمليات غير المتزامَنة.

مُكرَّرات لا تنتهي

يمكن أيضًا ألا تنتهي المُكرَّرات أبدًا. فمثلًا لا ينتهي المدى `range` لو صار `range.to = Infinity`. يمكن أيضًا أن نصنع كائن مُكرَّر (iterable object) يولّد أعدادًا شبه عشوائية (pseudorandom) لانهائية، ستفيد في حالات حرجة.

ما من حدود مفروضة على ناتج `next`. فيمكنه إعادة القيم مهما أراد وبالكم الذي أراد، لا مشكلة. طبعًا، المرور على هذا المُكرَّر بحلقة `for..of` لن ينتهي أبد الدهر، ولكن يمكننا إيقافها باستعمال `break`.

5.6.2 السلاسل النصية مُكرَّرة

تُعدّ المصفوفات والسلاسل النصية أكثر المُكرَّرات المقدَّمة من اللغة استعمالًا. بالنسبة إلى السلاسل النصية، فحلقة `for..of` تمرّ على محارفها:

```
for (let char of "test") {
  // تنفّذ أربع مرات: مرة لكلّ محرف
  alert( char ); // t ثمّ e ثمّ s ثمّ t
}
```

كما وتعمل -كما يجب!- مع الأزواج النائبة أو البديلة (Surrogate Pairs)!

```
let str = '𝒳😂';
for (let char of str) {
    alert( char ); // 𝒳 وثمّ 😂
}
```

5.6.3 نداء المُكرِّر جهارةً

لنعرف المُكرِّرات (iterator)معرفةً أعمق، لنرى كيف يمكن استعمالها جهارةً.

سنمرّ على سلسلة نصية بنفس الطريقة التي يمرّ بها `for..of`، ولكن هذه المرة ستكون النداءات مباشرة. تُنشِئ هذه الشيفرة مُكرِّرًا لسلسلة نصية وتأخذ القيم منه "يدويًا":

```
let str = "Hello";

// تنفّذ ما تنفّذه
// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();

while (true) {
  let result = iterator.next();
  if (result.done) break;
  alert(result.value); // تطبع المحارف واحدًا تلو الآخر
}
```

في الحياة الواقعية، نادرًا ما ستحتاج هذا. لكن المفيد أنّنا نتحكّم أكثر على عملية التكرار موازنةً بِـ `for..of`. فمثلًا يمكننا تقسيم عملية المرور على القيم: نمر على بعضها، نتوقّف ونفعل شيئًا آخر، ثمّ نواصل المرور على البقية مثلًا.

5.6.4 المُكرَّرات والشبيهات بالمصفوفات

هذان المصطلحان الرسميان يبدوان متشابهين إلى حدّ ما، ولكنّهما مختلفين تمام الاختلاف. حاولِ إدراكهما إدراكًا صحيحًا لتتجنّب هذا الاختلاط لاحقًا.

- المُكرَّرات (Iterables) - كائنات تُنفّذ التابع `Symbol.iterator`، كما شرحنا أعلاه.
- الشبيهات بالمصفوفات (Array-likes) - كائنات لها فهارس وصفة طول `length`، وبهذا "تشبه المصفوفات"... المصطلح يشرح نفسه.

حين نستعمل JavaScript للمهام الحقيقية في المتصفحات وغيرها من بيئات، نقابل مختلف الكائنات أكانت مُكرَّرات أو شبيهات بالمصفوفات، أو كليهما معًا. السلاسل النصية مثلًا مُكرَّرة (يمكن استعمال `for..of` عليها)، وشبيهة بالمصفوفات أيضًا (لها فهارس عددية وصفة `length`). ولكن ليس من الضروري أن يكون المُكرَّر شبيهًا بالمصفوفة، والعكس صحيح (لا يكون الشبيه بالمصفوفة مُكرَّرًا). فالمدى `range` في المثال أعلاه مُكرَّرًا، ولكنه ليس شبيه بالمصفوفة إذ ليس فيه صفات فهارس و`length`.

إليك كائنًا شبيهًا بالمصفوفات وليس مُكرَّرًا:

```
let arrayLike = { // فيه فهارس وطول => شبيه بالمصفوفات
"Hello",
"World",
  length: 2
};

// خطأ (ما من Symbol.iterator)
for (let item of arrayLike) {}
```

عادةً، لا تكون لا المُكرَّرات ولا الشبيهات بالمصفوفات مصفوفات حقًّا، فليس لها `push` أو `pop` وغيرها. لكن هذا غير منطقي. ماذا لو كان لدينا كائن من هذا النوع وأردنا التعامل معه بأنه مصفوفة؟ لنقل أنّا سنعمل على `range` باستعمال توابِع المصفوفات، كيف السبيل؟

5.6.5 التابع Array.from

التابع العام Array.from يأخذ مُكرَّرًا أو شبيهًا بالمصفوفات ويحوّله إلى مصفوفة "فعلية". بعدها ننادي توابِع المصفوفات التي نعرفها عليها. هكذا مثلًا:

```
let arrayLike = {
"Hello",
"World",
  length: 2
};

let arr = Array.from(arrayLike); // (*)
alert(arr.pop()); // تكتب World (أيّ أنّ التابع عمل)
```

يأخذ التابع `Array.from` في سطر `(*)` الكائن، ويفحصه أكان مُكرَّرًا أو شبيهًا بالمصفوفات، ويصنع مصفوفة جديدة ينسخ قيم ذلك الكائن فيها.

ذات الأمر للمُكرَّرات:

```
// نأخذ range من المثال أعلاه
let arr = Array.from(range);
alert(arr); // تكتب 1,2,3,4,5 (تحويل toString للمصفوفة يعمل)
```

والصياغة الكاملة للتابِع `Array.from` تتيح لنا تقديم دالة map اختيارية:

```
Array.from(obj[, mapFn, thisArg])
```

يمكن أن يكون الوسيط الاختياري الثاني `mapFn` دالةً تُطبّق على كلّ عنصر قبل إضافته للمصفوفة، ويتيح `thisArg` ضبط ماهيّة `this` للتابِع.

مثال:

```
// نأخذ range من المثال أعلاه

// نُربّع كلّ عدد
let arr = Array.from(range, num => num * num);

alert(arr); // 1,4,9,16,25
```

هنا نستعمل `Array.from` لتحويل سلسلة نصية إلى مصفوفة من المحارف:

```
let str = '𝒳 ';

// يقسم str إلى مصفوفة من المحارف
let chars = Array.from(str);
```

```
alert(chars[0]); // 𝒳
alert(chars[1]); //
alert(chars.length); // 2
```

على العكس من `str.split`، فهي هنا تعتمد على طبيعة تكراريّة السلسلة النصية، ولهذا تعمل كما ينبغي (كما تعمل `for..of`) مع الأزواج النائبة (surrogate pairs).

هنا أيضًا تقوم بذات الفعل، نظريًا:

```
let str = '𝒳 ';

let chars = []; // داخليًا، تُنفّذ Array.from ذات الحلقة
for (let char of str) {
  chars.push(char);
}

alert(chars);
```

...ولكن تلك أقصر.

يمكننا أيضًا صناعة تابِع `slice` مبني عليها يحترم الأزواج النائبة.

```
function slice(str, start, end) {
  return Array.from(str).slice(start, end).join('');
}

let str = '𝒳 𩷶';

alert( slice(str, 1, 3) ); //  𩷶

// التابع الأصيل/native في اللغة لا يدعم الأزواج النائبة
alert( str.slice(1, 3) ); // يُولّد نصّ "قمامة" (قطعتين من أزواج نائبة مختلفة)
```

5.6.6 الخلاصة

تُدعَى الكائنات التي يمكن استعمالها في حلقة `for..of` بالمُكرَّرات (iterable).

- على المُكرَّرات (تقنيًا) تنفيذ التابع ذي الاسم `Symbol.iterator`.
  - يُدعى ناتج `obj[System.iterator]` بالمُكرِّر (iterator) الذي يتكفَّل بعملية التكرار والمرور.

  - يجب أن يحتوي المُكرِّر التابع بالاسم `next()` حيث يُعيد كـائن `{done: Boolean, value: any}`... تُشير `done:true` هنا بأنَّ التكرار اكتمل، وإلّا فَـ `value` هي القيمة التالية.

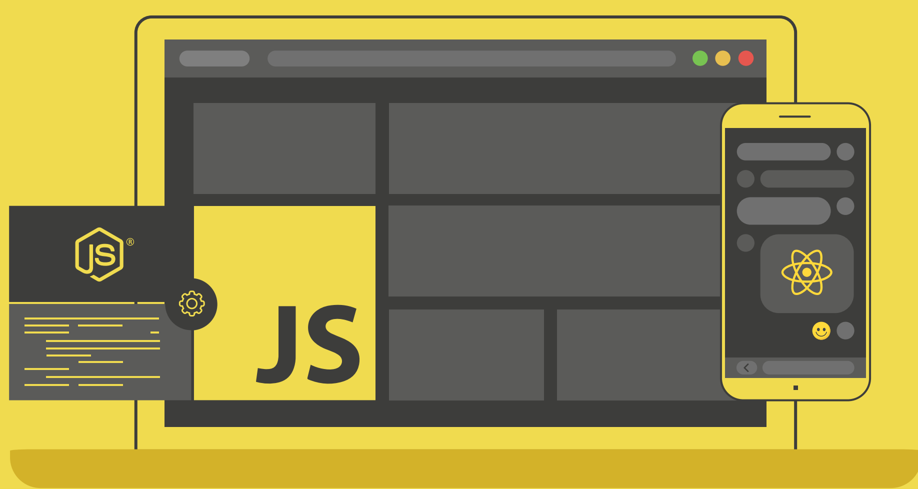
- تُنادي الحلقة `for..of` التابع `Symbol.iterator` تلقائيًا عند تنفيذها، ولكن يمكننا أيضًا فعل ذلـك يدويًا.
- تُنفِّذ المُكرِّرات المضمّنة في اللغة `Symbol.iterator` (مثل السلاسل النصية والمصفوفات).
- مُكرِّر السلاسل النصية يفهم الأزواج البديلة.

تُدعى الكائنات التي فيها صفات فهارس وصـفة طـول `length` بالشـبيهات بالمصـفوفات (array-like). يمكن أيضًا أن تكون لها صفات وتوابِع أخرى، إلّا أنّ ليس فيها توابِع المصفوفات المضمّنة في بنية اللغة.

لو نظرنا ورأينا مواصفات اللغة، فسنرى بأنّ أغلب التوابع المضمّنة فيها تتعامل مـع المصـفوفات على أنّهـا مُكرَّرات أو شبيهات بالمصفوفات بدل أن تكون مصفوفات "حقيقية"؛ هكذا تصير أكثر تجرّديّة (abstract).

تصنع `Array.from(obj[, mapFn, thisArg])` مصفوفةً `Array` حقيقيـة من المُكـرَّر أو الشـبيه بالمصفوفات `obj`، بهذا يمكن اسـتعمال توابِـع المصـفوفات عليهـا. يـتيح لنـا الوسـيطين `mapFn` و`thisArg` تقديم دالة لكلّ عنصر من عناصرها.

دورة تطوير التطبيقات باستخدام لغة JavaScript



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

5.7 النوع Map (الخرائط) والنوع Set (الأطقم)

تعلّمنا حتّى الآن بنى البيانات المعقّدة هذه:

- الكائنات Object: لتخزين التجميعات ذات المفاتيح (keyed collections).
- المصفوفات Array: لتخزين التجميعات المرتّبة (ordered collections).

ولكن في الحياة الواقعية، هذا لا يكفي. ولهذا تقدّم لك اللغة نوعيين آخرين: الخارطة Map والطقم Set.

5.7.1 الخارطة Map

تُعدّ الخارطة تجميعة ذات مفاتيح من عناصر البيانات، تمامًا مثل الكائنات Object، مع فرق بسيط، هو أنّ الخارطة Map تتيح استعمال المفاتيح مهمًا كان نوعها.

هذه توابِعها وخاصياتها:

- new Map(): يُنشِئ خارطة.
- map.set(key, value) - يضبط القيمة حسب مفتاحها.
- map.get(key) - يجلب القيمة حسب مفتاحها، وundefined لو لم يوجد key في الخارطة.
- map.has(key) - يُعيد true لو كان key موجودًا، وإلا فَـ false.
- map.delete(key) - يُزيل القيمة حسب مفتاحها.
- map.clear() - يُزيل كل شيء من الخارطة.
- map.size - يُعيد عدد العناصر الحالي.

إليك المثال الآتي:

```
let map = new Map();

map.set('1', 'str1');   // المفتاح سلسلة نصية
map.set(1, 'num1');     // المفتاح عدد
map.set(true, 'bool1'); // المفتاح قيمة منطقية

// أتذكر كيف أنّ الكائن العادي يُحوّل المفاتيح لأي سلاسل نصية؟
// الخارطة هنا تحترم النوع، وهذان السطران مختلفان:
alert( map.get(1)   ); // 'num1'
```

```
alert( map.get('1') ); // 'str1'

alert( map.size ); // 3
```

كما ترى، فالمفاتيح لا تُحوّل إلى سلاسل نصية (على العكس من الكائنات). يمكنك أن تضع أيّ نوع من المفاتيح تريد.

يمكن أن تستعمل الخارطة الكائناتَ نفسها مفاتيح، إليك المثال التالي:

```
let john = { name: "Ahmad" };

// لنخزّن عدد زيارات كل زائر لنا
let visitsCountMap = new Map();

// كائن john هو مفتاح الخارطة
visitsCountMap.set(john, 123);

alert( visitsCountMap.get(john) ); // 123
```

يُعدّ استعمال الكائنات على أنّها مفاتيح أحدُ أهمّ صفات Map. لو أردت المفاتيح سلاسل نصية، فالكائنات Object تكفيك وزيادة، لكن لو أردت المفاتيح كائنات، فسيخونك Object للأسف. لنرى:

```
let john = { name: "Ahmad" };

let visitsCountObj = {}; // نحاول استعمال كائن

visitsCountObj[john] = 123; // ونحاول استعمال كائن john مفتاحًا فيه

// وهذا ما وجدناه مكتوبًا!
alert( visitsCountObj["[object Object]"] ); // 123
```

المتغيّر visitsCountObj من نوع "كائن" (object)، ولهذا يحوّل كلّ المفاتيح (مثل john) إلى سلاسل نصية. وبهذا قدّم لنا المفتاح بالسلسلة النصية "[object Object]". ليس ما نريد قطعًا.

كيف تُوازن الخارطة Map المفاتيح

تستعمل Map الخوارزمية SameValueZero لتختبر تساوي المفتاح مع الآخر. تتشابه هذه الخوارزمية تقريبًا مع المساواة الصارمة === بفارق أنّ NaN تساوي NaN في نظرها. يعني ذلك بأنك تستطيع استعمال NaN كمفتاح هو الآخر. لا يمكن تغيير هذه الخوارزمية ولا تخصيصها.

سِلسلة الاستدعاءات

كلّما نادينا `map.set` أعاد لنا التابع الخارطة نفسها، وبهذا يمكن أن نستدعي التابع على ناتج الاستدعاء السابق:

```
map.set('1', 'str1')
  .set(1, 'num1')
  .set(true, 'bool1');
```

5.7.2 المرور على خارطة

هناك ثلاث طرائق للمرور على عناصر `Map` وتطبيق عملية عليها:

- `map.keys()`: يُعيد مُكرَّرًا (iterable) للمفاتيح،
- `map.values()`: يُعيد مُكرَّرًا للقيم،
- `map.entries()`: يُعيد مُكرَّرًا للمدخلات `[key, value]`، وهي التي تستعملها `for..of` مبدئيًا.

مثال:

```
let recipeMap = new Map([
  ['cucumber', 500],
  ['tomatoes', 350],
  ['onion',    50]
]);

// نمرّ على المفاتيح (الخضراوات)
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // cucumber, tomatoes, onion
}

// نمرّ على قيم المفاتيح (عدد الخضراوات)
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}

// نمرّ على مدخلات [key, value]
for (let entry of recipeMap) { // recipeMap.entries() مثل
  alert(entry); // cucumber,500 (وهكذا)
}
```

ترتيب الإدخال هو المستعمل

تسير عملية المرور على الكائنات بنفس الترتيب الذي أُدخلت به، فالخارطة تحفظ هذا الترتيب على العكس من الكائنات Object.

علاوةً على ذلك، فتملك الخارطة Map التابع المضمّن فيها forEach، كما المصفوفات Array:

```
// (key, value) تُنفّذ الدالة على كلّ زوج
recipeMap.forEach( (value, key, map) => {
  alert(`${key}: ${value}`); // cucumber: 500 إلخ إلخ
});
```

5.7.3 Object.entries: صنع خارطة من كائن

متى ما أُنشأت خارطة Map نستطيع تمرير مصفوفة (أو مُكرَّرًا آخرًا) لها أزواج "مفاتيح/قيم" لتهيئتها، هكذا تمامًا:

```
// [key, value] مصفوفة من أزواج
let map = new Map([
  ['1',  'str1'],
  [1,    'num1'],
  [true, 'bool1']
]);

alert( map.get('1') ); // str1
```

لو كان أمامنا كائنًا عاديًا ونريد صناعة Map منه، فيمكننا استعمال التابع المضمّن في اللغة Object.entries(obj) إذ يُعيد مصفوفة مكوّنة من أزواج "مفاتيح/قيم" للكائن، بنفس الصيغة التي يطلبها ذاك التابع. ولهذا يمكن أن نصنع خارطة من كائن بهذه الطريقة:

```
let obj = {
  name: "Ahmad",
  age: 30
};

let map = new Map(Object.entries(obj));

alert( map.get('name') ); // Ahmad
```

نرى هنا التابع `Object.entries` يُعيد مصفوفة بأزواج "مفاتيح/قيم": `[ ["name","Ahmad"] ,["age", 30] ]`، وهذا ما تحتاج إليه الخارطة.

5.7.4 Object.fromEntries: صنع كائن من خارطة

رأينا كيف نصنع خارطة `Map` من كائنٍ عاديّ باستعمال `Object.entries(obj)`. على العكس منه فالتابع `Object.fromEntries` يأخذ خارطة فيها أزواج `[key, value]` ويصنع كائنًا منها:

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

// now prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

يمكننا استعمال `Object.fromEntries` لنصنع كائنًا عاديًا من `Map`. يُفيدنا هذا مثلًا في تخزين البيانات في خارطة، بينما نريد تمريرها إلى شيفرة من طرف ثالثة تريد كائنًا عاديًا لا خارطة.

هذه الشيفرة المنشودة:

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // (*) نصنع كائنًا عاديًا

// وهكذا انتهينا!
// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2
```

متى ما استدعينا `map.entries()` أعادت مصفوفة مؤلّفة من أزواج "مفاتيح/قيم" بنفس التنسيق الذي يطلبه `Object.fromEntries` تمامًا، لحسن الحظ.

يمكننا تقصير السطر المعلّم `(*)` ذاك:

```
let obj = Object.fromEntries(map); // .entries() بدون
```

النتيجة نفسها إذ أنّ التابع `Object.fromEntries` يتوقّع كائنًا مُتعدَّدًا وسيطًا له، وليس مصفوفة بالضرورة. كما والتعداد القياسي للخارطة يتوقّع ذات أزواج "مفاتيح/قيم" التي يتوقّعها `map.entries()`، وهكذا نجد في يدنا كائنًا عاديًا له نفس "مفاتيح/قيم" الخارطة `map`.

5.7.5 الطقم Set

الأطقم (النوع `Set`) هي نوع خاصّ من التجميعات (collection) ليس له مفاتيح ولا يمكن أن يحوي أكثر من قيمة متطابقة. يمكن عدّها كأطقم المجوهرات والأسنان، حيث لا تتكرّر أي قطعة مرتين.

إليك توابِعه الرئيسة:

- `new Set(iterable)` - يصنع الطقم. في حال مرّرت كائن `iterable` (وهو عادةً مصفوفة)، فينسخ بياناته إلى الطقم.
- `set.add(value)` - يُضيف قيمة إلى الطقم ويُعيده ذاته.
- `set.delete(value)` - يُزيل القيمة ويُعيد `true` لو كانت القيمة `value` موجودة عند استدعاء التابع، وإلّا يُعيد `false`.
- `set.has(value)` - يُعيد `true` لو كانت القيمة موجودة في الطقم، وإلّا يُعيد `false`.
- `set.clear()` - يُزيل كلّ شيء من الطقم.
- `set.size` - خاصية عدد العناصر في الطقم.

الميزة الأهمّ للأطقم هي أنّك لو استدعيت `set.add(value)` أكثر من مرّة وبنفس القيمة، فكأنّك استدعيتهُ مرّة واحدة، لهذا تظهر كل قيمة في الطقم مرّة واحدة لا غير.

عُدّ مثلًا أنّ زوّارًا قادمين إلى وليمة ونريد تذكّر كلّ واحد لإعداد ما يكفي من طعام... ولكن يجب ألّا نسجّل الزوّار مرتين، فالزائر "واحد" ونعدّه مرّة واحدة فقط. الطقم هنا هو الخيار الأمثل:

```
let set = new Set();

let john = { name: "Ahmad" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// زارنا الناس، وهناك من زارنا أكثر من مرة
```

```
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// لا يحفظ الطقم إلا القيم الفريدة
alert( set.size ); // 3

for (let user of set) {
  alert(user.name); // (ثمّ Pete وMary) Ahmad
}
```

يمكن عوض الأطقم استعمال مصفوفة من المستخدمين، مع نصّ يتحقّق من البيانات عند إدخالها لألّا تحدث تكرارات (باستعمال arr.find). هذا ممكن نعم، لكن الأداء سيكون أشنع بكثير فتابِع البحث arr.find يمرّ على "كامل المصفوفة" فيفحص كلّ عنصر فيها. الطقم Set أفضل بمراحل فأداؤه في فحص تفرّد العناصر مُحسَّن داخل بنية اللغة.

5.7.6 المرور على طقم

يمكن لنا المرور على عناصر الطقم باستعمال حلقة for..of أو تابِع forEach:

```
let set = new Set(["oranges", "apples", "bananas"]);

for (let value of set) alert(value);

// نفس الأمر مع forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

ولكن لاحظ هنا طرافة التابع: لدالة ردّ النداء (callback function) المُمرّرة إلى forEach ثلاث وُسطاء: قيمة value، وذات القيمة الأولى valueAgain، والكائن الهدف. لاحظتَ كيف تكرّرت ذات القيمة في الوُسطاء مرّتين؟

يعزو هذا إلى توافق Set مع Map إذ لدالة ردّ النداء المُمرّرة إلى forEach الخارطة ثلاث وُسطاء أيضًا. معك حق، أمرها غريب، ولكنّها مفيدة فتُسهّل حياتنا لو أردنا استبدال الخارطة بالطقم في حالات حرجة، كما العكس أيضًا.

كما تدعم الأطقم نفس التوابع التي تدعمها الخارطة للتعامل مع المُكرِّرات (iterators):

- `set.keys()` - يُعيد كائنًا مُكرَّرًا (iterable object) من القيم،
- `set.values()` - تمامًا مثل `set.keys()` (موجود للتوافق مع `Map`)،
- `set.entries()` - يُعيد كائنًا مُكرَّرًا من المُدخلات `[value, value]` (موجود للتوافق مع `Map`).

5.7.7 ملخص

الخارطة `Map` هي تجميعة ذات مفاتيح. توابعها وخاصياتها:

- `new Map([iterable])` - يصنع خريطة ويضع فيها أزواج `[key,value]` داخل المُكرَّر `iteratable` الاختياري (يمكن أن يكون مثلًا مصفوفة).
- `map.set(key, value)` - يخزّن القيمة حسب مفتاحها.
- `map.get(key)` - يُعيد القيمة حسب مفتاحها، ويُعيد `undefined` لو لم يكن المفتاح `key` في الخارطة.
- `map.has(key)` - يُعيد `true` لو كان المفتاح `key` موجودًا، وإلا يُعيد `false`.
- `map.delete(key)` - يُزيل القيمة حسب مفتاحها.
- `map.clear()` - يُزيل كل ما في الخارطة.
- `map.size` - يُعيد عدد العناصر في الخارطة الآن.

اختلافاتها مع الكائنات العادية (`Object`):

- تدعم أنواع المفاتيح المختلفة، كما والكائنات نفسها أيضًا.
- فيها توابِع أخرى تفيدنا، كما وخاصية `size`.

الطقم `Set` هو تجميعة من القيم الفريدة.

توابعه وخاصياته:

- `new Set([iterable])` - يصنع طقمًا ويضع فيه أزواج `[key, value]` داخل المُكرَّر الاختياري (يمكن أن يكون مثلًا مصفوفة).
- `set.add(value)` - يُضيف القيمة `value` (ولو كانت موجودة لا يفعل شيء) ثمّ يُعيد الطقم نفسه.
- `set.delete(value)` - يُزيل القيمة ويُعيد `true` لو كانت موجودة عند استدعاء التابع، وإلا يُعيد `false`.

- `set.has(value)` - يُعيد `true` لو كانت القيمة في الطقم، وإلا يُعيد `false`.
- `set.clear()` - يُزيل كل ما في الطقم.
- `set.size` - عدد عناصر الطقم.

يسري ترتيب المرور على عناصر `Map` و`Set` بترتيب إدخالها فيهما دومًا، ولهذا لا يمكن أن نقول بأنّها تجميعات غير مرتّبة، بل أنّا لا نقدر على إعادة ترتيب عناصرها أو الحصول عليها بفهرسها فيها.

5.7.8 تمارين

ا. ترشيح العناصر الفريدة في مصفوفة

الأهمية: ★★★★★

عُدّ أنّ `arr` مصفوفة. أنشِئ دالة `unique(arr)` تُعيد مصفوفة مؤلّفة من العناصر الفريدة في `arr`.

مثال:

```
function unique(arr) {
  /* هنا تكتب شيفرتك*/
}

let values = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-O"
];

alert( unique(values) ); // Hare, Krishna, :-O
```

لاحظ أنّ السلاسل النصية استُعملت هنا، ولكن يمكن أن تكون القيم بأيّ نوع آخر.

غُشّ من هذه: استعمل `Set` لتخزين القيم الفريدة.

الحل:

```
function unique(arr) {
  return Array.from(new Set(arr));
}
```

ب. ترشيح الألفاظ المقلوبة

الأهمية: ★★★★☆

تُسمّى الكلمات التي لها ذات الأحرف ولكن بترتيب مختلف ألفاظًا مقلوبة، مثل هذه:

```
nap - pan
ear - are - era
cheaters - hectares - teachers
```

أو العربية:

```
ملّ - لمّ
مسكين - سيكمن
كاتب - اكتب - كتاب
```

اكتب دالة `aclean(arr)` تُعيد مصفوفةً دون هذه الألفاظ المقلوبة. هكذا:

```
let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era",
"hectares"];


alert( aclean(arr) ); // "nap,teachers,ear" أو "PAN,cheaters,era"
```

يجب أن يكون ناتج كلّ مجموعة ألفاظ كلمة واحدة فقط، ولا يهمّنا أيّ واحدة.

الحل:

لو أردنا البحث عن كل الألفاظ المقلوبة، سنقسم كلّ كلمة إلى حروفها ونرتّبها. متى ما رتّبناها حسب الأحرف، فستكون الألفاظ كلها متطابقة. هكذا:

```
nap, pan -> anp
ear, era, are -> aer
cheaters, hectares, teachers -> aceehrst
...
```

سنستعمل كلّ قيمة مختلفة (ولكن متطابقة بترتيب أحرفها) لتكون مفاتيح خريطة فنخزّن لفظًا واحدًا لكل مفتاح فقط:

```
function aclean(arr) {
  let map = new Map();
```

```
    for (let word of arr) {

      // نقسم الكلمة بأحرفها، ونرتّب الأحرف ونجمعها ثانيةً
      let sorted = word.toLowerCase().split('').sort().join(''); // (*)
      map.set(sorted, word);
    }

    return Array.from(map.values());
  }

let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era",
"hectares"];

alert( aclean(arr) );
```

نُنفّذ الترتيب حسب الأحرف بسلسلة استدعاءات كما في السطر `(*)`. سنقسمها على أكثر من سطر ليسهل فهمها:

```
let sorted = arr[i] // PAN
  .toLowerCase() // pan
  .split('') // ['p','a','n']
  .sort() // ['a','n','p']
  .join(''); // anp
```

هكذا يكون لدى الكلمتين المختلفتين `'PAN'` و`'nap'` ذات الشكل حين تُرتّب أحرفها: `'anp'`.

في السطر اللاحق نُضيف الكلمة إلى الخارطة.

```
map.set(sorted, word);
```

لو قابلنا بينما نمرّ على الكلمات كلمةً لها نفس الشكل حين تُرتّب أحرفها، فستعوّض القيمة السابقة التي لها نفس المفتاح في الخارطة. هكذا لن تزيد الكلمات لكلّ شكل على واحد، دومًا.

وفي النهاية يأخذ `Array.from(map.values())` متُعدَّدًا يمرّ على قيم الخارطة (لا نريد مفاتيحها في ناتج الدالة) فيُعيد المصفوفة نفسها.

يمكننا (في هذه المسألة) استعمال كائن عادي بدل الخارطة، إذ أنّ المفاتيح سلاسل نصية. هكذا سيبدو الحلّ لو اتبعنا هذا النهج:

```
function aclean(arr) {
  let obj = {};

  for (let i = 0; i < arr.length; i++) {
    let sorted = arr[i].toLowerCase().split("").sort().join("");
    obj[sorted] = arr[i];
  }

  return Object.values(obj);
}

let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era",
"hectares"];

alert( aclean(arr) );
```

ج. مفاتيح مُكرَّرة

الأهمية: ★★★★★

نريد تسجيل المصفوفة الناتجة من `map.keys()` في متغيّر وثمّ استدعاء توابِع تخصّ المصفوفات عليها مثل `push`.. ولكنّ ذلك لم ينفع:

```
let map = new Map();

map.set("name", "Ahmad");

let keys = map.keys();

// خطأ: keys.push ليست دالة
keys.push("more");
```

لماذا؟ وكيف يمكننا إصلاح الشيفرة ليعمل `keys.push`؟

الحل:

لأنّ التابع `map.keys()` يُعيد مُكرَّرًا لا مصفوفة. يمكننا تحويله إلى مصفوفة باستعمال `Array.from`:

```
let map = new Map();

map.set("name", "Ahmad");

let keys = Array.from(map.keys());

keys.push("more");

alert(keys); // name, more
```

5.8 النوع WeakMap والنوع WeakSet: الخرائط والأطقم ضعيفة الإشارة

كما عرفنا من فصل "كنس المهملات"، فمُحرّك JavaScript يُخزِّن القيمة في الذاكرة طالما يمكن أن يصل لها شيء (أي يمكن استعمالها لاحقًا). هكذا:

```
let john = { name: "Ahmad" };

// يمكننا الوصول إلى الكائن، فـ john هو الإشارة إليه

// عوّض تلك الإشارة
john = null;

// سيُزال الكائن من الذاكرة
```

عادةً ما تكون خاصيات الكائن أو عناصر المصفوفة أو أية بنية بيانات أخرى - عادةً ما تُعدّ "مُتاحة لباقي الشيفرة" ويُبقيها المحرّك في الذاكرة طالما بنية البيانات نفسها في الذاكرة.

لنفترض أنّا وضعنا كائنًا في مصفوفة، طالما المصفوفة موجودة ومُشار إليها، فسيكون الكائن موجودًا هو الآخر حتّى لو لم يكن هناك ما يُشير إليه مثلما في هذه الشيفرة:

```
let john = { name: "Ahmad" };
let array = [ john ];

john = null; // عوّض الإشارة
// الكائن john مخزّن داخل مصفوفة ولن يُكنس باعتباره مهملات
// إذ يمكننا أخذه بهذه: array[0]
```

وبنفس المفهوم، لو استعملنا كائنًا ليكون مفتاحًا في خارطة `Map` عادية، فسيبقى هذا الكائن موجودًا طالما الخارطة تلك موجودة، ويشغل الذاكرة مانعًا عملية كنس المهملات من تحريرها. إليك هذا المثال:

```
let john = { name: "Ahmad" };
let map = new Map();
map.set(john, "...");
john = null; // عوّض الإشارة

// الكائن john مخزّن داخل خارطة
// ويمكننا أخذه بهذه: map.keys()
```

على العكس فالخارطة ضعيفة الإشارة WeakMap مختلفة جذريًا عن هذا، فلا تمنع كنس مهملات أيٍّ من مفاتيحها الكائنات. لنأخذ بعض الأمثلة لتُدرك القصد هنا.

5.8.1 WeakMap

أولى اختلافات الخارطة ضعيفة الإشارة WeakMap عن تلك العادية Map هي أنّها تُلزم مفاتيحها بأن تكون كائنات لا أنواع أولية:

```
let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // لا مشاكل (المفتاح كائن)

// لا يمكن استعمال السلسلة النصية مفتاحًا
weakMap.set("test", "Whoops"); // خطأ، لأنّ "test" ليس كائنًا
```

بعد ذلك لو استعملنا أحد الكائنات ليكون مفتاحًا فيها، ولم يكن هناك ما يُشير إلى هذا الكائن، فسيُزال الكائن من الذاكرة (والخارطة) تلقائيا.

```
let john = { name: "Ahmad" };

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // عوّض الإشارة

// أُزيل الكائن john من الذاكرة!
```

وازِن هذه الشيفرة بشيفرة الخارطة Map أعلاه. الآن حتى لو لم يكن john موجودًا إلا مفتاحًا لِـ WeakMap، فسيُحذَف تلقائيًا من الخارطة (ومن الذاكرة).

لا تدعم الخارطة ضعيفة الإشارة WeakMap التكرار أو المرور على العناصر (iteration) ولا التوابع keys() أو values() أو entries()، ولهذا لا نقدر على أخذ كلّ المفاتيح أو القيم التي فيها. بل أنّ للخارطة WeakMap التوابع الآتية:

- weakMap.get(key)
- weakMap.set(key, value)

- `weakMap.delete(key)`
- `weakMap.has(key)`

تفكّر بسبب وجود هذا التقييد؟ الجواب هو: أسباب تقنية. عُدّ الكائن الآن قد فقد كلّ إشارة له (مثلما حصل مع الكائن `john` في الشيفرة أعلاه)، بهذا ستُكنَس مهملاته تلقائيًا ويحذف من الذاكرة، ولكن... وقت حدوث هذا الكنس غير موضّح تقنيًا. الواقع أنّ محرّك JavaScript يُحدِّد ذلك: فهو يُحدّد متى يمسح الذاكرة، الآن حالًا أو بعد قليل حتّى تحدث عمليات حذف أخرى. لذا فعدد العناصر الحالي داخل `WeakMap` غير معلوم تقنيًا، ربما يكون المحرّك حذفها كلها أو لم يحذفها، أو حذف بعضها، لا نعلم. لهذا السبب لا تدعم اللغة التوابع التي تحاول الوصول إلى كلّ القيم والعناصر.

الآن بعدما عرفناها، في أيّ حالات نستعمل هذه البنية من البيانات؟

5.8.2 استعمالاتها: بيانات إضافية

المجال الرئيسي لتطبيقات `WeakMap` هي "تخزين البيانات الإضافية".

لو كنّا نتعامل مع كائن "ينتمي" إلى شيفرة أخرى (وحتّى مكتبة من طرف ثالث) وأردنا تخزين بيانات معيّنة لترتبط بها، وهذه البيانات لا تكون موجودة إلا لو كان الكائن موجودًا، فَـ `WeakMap` هي ما نريد تمامًا: نضع البيانات في خارطة بإشارة ضعيفة `WeakMap` (مستعملين الكائن مفتاحًا لها). متى ما كُنس الكائن باعتباره مهملات، ستختفي تلك البيانات معه أيضًا.

```
weakMap.set(john, "secret documents");
// إن مات john فستُدمّر تلك المستندات فائقة السرية تلقائيًا
```

لنرى مثالًا يوضّح الصورة. عُدّ بأنّ لدينا شيفرة تسجّل عدد زيارات المستخدمين - تسجّلها في خارطة، حيث كائن المستخدم هو مفتاحها وعدد زياراته هي القيمة. لا نريد أن نُسجّل عدد زياراته فيما لو غادر المستخدم (أي أنّ عملية كنس المهملات كنست ذاك الكائن).

إليك مثالًا آخر عن دالة عَدّ باستعمال `Map`:

```
//    visitsCount.js
let visitsCountMap = new Map(); // خارطة: المستخدم => عدد زياراته
// تزيد عدد الزيارات
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

وهذا الجزء الثاني من الشيفرة (يمكن أن يستعمل هذا الملف ذاك):

```
//    main.js
let john = { name: "Ahmad" };

countUser(john); // عَدّ الزوّار
countUser(john);

// بعدها يغادر john الحفلة
john = null;
```

هكذا "يُفترض" أن يُكنس الكائن john باعتباره مهملات، لكنّه سيبقى في الذاكرة إذ تستعمله الخارطة visitsCountMap مفتاحًا فيها.

علينا مسح visitsCountMap حين نُزيل المستخدمين وإلا فسيزيد حجمها في الذاكرة إلى آباد الآبدين. لو كانت بنية البرمجية معقّدة، فستكون عملية المسح هذه مرهقة جدًا وغير عملية. لهذا يمكننا تجنّب التعب واستعمال WeakMap بدل العادية:

```
//    visitsCount.js
let visitsCountMap = new WeakMap(); // خارطة بإشارة ضعيفة: المستخدم => عدد زياراته

// تزيد عدد الزيارات
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

هكذا لا نمسح visitsCountMap يدويًا بل نترك للمحرّك القرار: لو لم يكن هناك ما يُشير إلى الكائن john عدا مفتاح WeakMap، سيحرّره من الذاكرة مع المعلومات التي في ذلك المفتاح داخل الخارطة ضعيفة الإشارة WeakMap.

5.8.3 استعمالاتها: الخبيئة

يكثُر أيضًا استعمال الخرائط للخبيئة، أي حين علينا تذكّر ناتج الدالة (تخبئته "cached") كي يستعمل أيّ استدعاء لاحِق على هذا العنصر تلك الخبيئة.

يمكن أن نستعمل الخارطة Map لتخزين النتائج:

```
//    cache.js
let cache = new Map();

// نحسب النتيجة ونتذكرها
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* حسابات الكائن هذا */ obj;

    cache.set(obj, result);
  }

  return cache.get(obj);
}

// الآن نستعمل process() في ملف آخر:

//    main.js
let obj = {/* فلنفترض وجود هذا الكائن*/};

let result1 = process(obj); // حسبنا القيمة

// ...بعدها، في مكان آخر من الشيفرة...
let result2 = process(obj); // تأخذ النتيجة تلك من الخبيئة

// ...بعدها، لو لم نرد الكائن بعد الآن:
obj = null;

alert(cache.size); // 1 (لا! ما زال الكائن في الخبيئة ويستهلك الذاكرة)
```

لو استدعينا `process(obj)` أكثر من مرّة بتمرير نفس الكائن، فستحسب الشيفرة النتيجة أوّل مرة فقط، وفي المرات القادمة تأخذها من الكائن `cache`. مشكلة هذه الطريقة هي ضرورة مسح `cache` متى ما انتفت حاجتنا من الكائن. لكن، لو استبدلنا `Map` وعوّضناها بِـ `WeakMap` فستختفي المشكلة تمامًا، وتُزال النتيجة المُخبّأة من الذاكرة تلقائيًا متى ما كُنس الكائن على أنّه مهملات.

```
//    cache.js
let cache = new WeakMap();
```

```
// نحسب النتيجة ونتذكرها
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* حسابات الكائن هذا */ obj;

    cache.set(obj, result);
  }

  return cache.get(obj);
}

//    main.js
let obj = {/* كائن من الكائنات */};

let result1 = process(obj);
let result2 = process(obj);

// ...بعدها، لو لم نرد الكائن بعد الآن:
obj = null;
```

هنا، لا يمكن أن نعرف cache.size إذ أنها خارطة بإشارة ضعيفة، ولكن الحجم صفر، أو سيكون صفر قريبًا؛ فما أن تبدأ عملية كنس المهملات على الكائن، ستُزال البيانات المُخبّأة هي الأخرى.

5.8.4 WeakSet

حتّى الأطقم ضعيفة الإشارة `WeakSet` تسلك ذات السلوك:

- تشبه الأطقم العادية `Set` ولكن لا يمكننا إلّا إضافة الكائنات إلى `WeakSet` (وليس الأنواع الأولية).
- يبقى الكائن موجودًا في الطقم طالما هناك ما يصل إليه.
- ويدعم -كما تدعم `Set`- التوابع `add` و`has` و`delete`، ولكن لا تدعم `size` أو `keys()` أو التعداد.

هي الأخرى تخدمنا نحن المطورون في تخزين البيانات الإضافية (إذ أنّ الإشارة إليها "ضعيفة")، ولكنها ليست لأيّ بيانات كانت، بل فقط التي تُعطي إجابة "نعم/لا". لو كان الكائن موجودًا داخل طقم بإشارة ضعيفة، فلا بدّ أنّه موجود لداعٍ.

يمكننا مثلًا إضافة المستخدمين إلى طقم بإشارة ضعيفة `WeakSet` لنسجّل من زار موقعنا:

```
let visitedSet = new WeakSet();

let john = { name: "Ahmad" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

visitedSet.add(john); // Ahmad زارنا
visitedSet.add(pete); // Pete وبعده
visitedSet.add(john); // Ahmad وعاد

// تحتوي visitedSet الآن على مستخدمين اثنين

// هل زارنا Ahmad؟
alert(visitedSet.has(john)); // true

// هل زارتنا Mary؟
alert(visitedSet.has(mary)); // false

john = null;

// ستنظِّف visitedSet تلقائيًا
```

التقييد الأهم في هذه الأنواع `WeakSet` و`WeakMap` هي عدم موجود المُكرَّرات واستحالة أخذ محتواها كلـه. لربّما ترى ذلك غباءً، إلّا أنّه لا يمنع هذه الأنواع من إجراء مهامها التي صُنعت لها: مخزن "إضـافي" من البيانـات للكائنات المخزّنة (أو المُدارة) في مكان آخر.

5.8.5 الخلاصة

الخارطة ضعيفة الإشارة (WeakMap) هي تجميعـة تشـبه الخرائـط العاديـة (Map-like collection)، ولا تتيح إلا استعمال الكائنات مفاتيحٍ فيها، كما وتُزيلها هي والقيمة المرتبطة بها ما إن تنعدم الإشارة إليها.

الطقم ضعيفة الإشارة (WeakSet) هي تجميعة تشبه الأطقم العادية (Set-like collection)، ولا تخزّن إلا الكائنات فيها، كما وتُزيلها ما إن تنعدم الإشارة إليها.

كِلا النوعان لا يدعمان التوابع والخاصيات التي تُشير إلى كل المفاتيح فيهما، أو حتى عددها. المسموح فقط هو العمليات على العناصر فيها عنصرًا بعنصر.

يُستعمل هذان النوعان WeakMap وWeakSet على أنّهما بنى بيانات "ثانوية" إلى جانب تلك "الأساسية" لتخزين العناصر. لو أُزيل الكائن من التخزين الأساسي، ولم يوجد له أي إشارة إلا مفتاحًا في WeakMap أو عنصرًا في WeakSet، مسحهُ المحرّك تلقائيًا.

5.8.6 تمارين

ا. تخزين رايات "غير مقروءة"

الأهمية: ★★★★★

لديك مصفوفة من الرسائل:

```
let messages = [
  {text: "Hello", from: "Ahmad"},
  {text: "How goes?", from: "Ahmad"},
  {text: "See you soon", from: "Alice"}
];
```

ويمكن للشيفرة عندك الوصول إليها، إلّا أنّ شيفرة أحدهم تُدير تلك الرسائل، فتُضيف رسائل جديدة وتُزيل قديمة، ولا تعرف متى يحدث هذا بالضبط.

السؤال هو: أيّ بنية من بنى البيانات تستعمل لتخزّن هذه المعلومة لكلّ رسالة: "هل قُرأت؟". يجب أن تكون البنية التي اخترتها مناسبة لتردّ على سؤال "هل قُرأت؟" لكلّ كائن رسالة.

> حين تُزال رسالة من مصفوفة messages، يجب أن تختفي من بنية البيانات لديك هي الأخرى.

> يجب ألّا نُعدّل كائنات الرسائل ولا نُضيف خاصيات من عندنا إليها؛ فيمكن أن يؤدّي هذا إلى عواقب وخيمة إذ لسنا من نديرها بل أحد آخر.

الحل:

لنجرّب تخزين الرسائل المقروءة في طقم بإشارة ضعيفة WeakSet:

```
let messages = [
  {text: "Hello", from: "Ahmad"},
  {text: "How goes?", from: "Ahmad"},
  {text: "See you soon", from: "Alice"}
];
```

```
let readMessages = new WeakSet();

// قرأ المستخدم رسالتين اثنتين
readMessages.add(messages[0]);
readMessages.add(messages[1]);
// في readMessages الآن عنصرين

// هيّا نُعيد قراءة أول رسالة!...
readMessages.add(messages[0]);
// ما زالت في readMessages عنصرين فريدين

// الجواب: هل قُرئت message [0]؟
alert("Read message 0: " + readMessages.has(messages[0])); // true نعم

messages.shift();
// الآن في readMessages عنصر واحد (تقنيًا فستُنظَف الذاكرة فيما بعد)
```

يتيح لنا الطقم ضعيفة الإشارة تخزينَ مجموعة من الرسائل والتأكّد من وجود كلّ منها بسهولة تامة. كما وأنّها تمسح نفسها بنفسها. للأسف بهذا نُضحّي بميزة التكرار، فلا يمكن أن نجلب "كلّ الرسائل المقروءة" منها مباشرةً، ولكن... يمكننا المرور على عناصر كل الرسائل في `messages` وترشيح تلك التي في الطقم لدينا.

يمكن أن يكون الحل الآخر هو إضافة خاصية مثل `message.isRead=true` إلى الرسالة بعد قراءتها. ولكنّنا لسنا من نُدير هذه الكائنات بل أحد آخر، ولهذا لا يُوصى بذلك بصفة عامة. ولكن، يمكننا استعمال خاصيّة رمزية فنتجنّب أي مشكلة أو تعارض.

هكذا:

```
// الخاصية الرمزية معروفة في الشيفرة لدينا، فقط
let isRead = Symbol("isRead");
messages[0][isRead] = true;
```

"لربما" الآن لن تعرف شيفرة الطرف الثالث بخاصيتنا الجديدة.

صحيح أن الرموز تتيح لنا تقليل احتمال حدوث المشاكل، إلّا أنّ استعمال `WeakSet` أفضل بعين بنية البرمجية.

ب. تخزين تواريخ القراءة

الأهمية: ★★★★★

لديك مصفوفة من الرسائل تشبه تلك في التمرين السابق، والفكرة هنا متشابهة قليلًا.

```
let messages = [
  {text: "Hello", from: "Ahmad"},
  {text: "How goes?", from: "Ahmad"},
  {text: "See you soon", from: "Alice"}
];
```

السؤال: أيّ بنية بيانات تستعمل لتخزين هذه المعلومة: "متى قُرئَت هذه الرسالة؟".

كان عليك (في التمرين السابق) تخزين معلومة "نعم/لا" فقط، أمّا الآن فعليك تخزين التاريخ، ويجب أن يبقى في الذاكرة إلى أن تُكنَس الرسالة على أنّها مهملات.

ملاحظة: تُخزّن التواريخ كائنات بصنف `Date` المضمّن في اللغة، وسنتكلم عنه لاحقًا.

الحل:

يمكن أن نستعمل الخارطة ضعيفة الإشارة لتخزين التاريخ:

```
let messages = [
  {text: "Hello", from: "Ahmad"},
  {text: "How goes?", from: "Ahmad"},
  {text: "See you soon", from: "Alice"}
];

let readMap = new WeakMap();

readMap.set(messages[0], new Date(2017, 1, 1));
// سنرى أمر كائن التاريخ لاحقًا
```

5.9 مفاتيح الكائنات وقيمها ومدخلاتها

لنأخذ راحة صغيرة بعيدًا عن بنى البيانات ولنتحدَّث عن طريقة المرور على عناصرها.

رأينا في الفصل السابق التوابع `map.keys()` و`map.values()` و`map.entries()`. هذه التوابع عامّة وقد اتّفق معشر المطوّرين على استعمالها عند التعامل مع بنى البيانات. ولو أنشأنا بنية بيانات من الصفر بيدنا، فعلينا توفير "تنفيذ" تلك التوابع أيضًا. هي أساسًا مدعومة لكلّ من:

- الخرائط `Map`
- الأطقم `Set`
- المصفوفات `Array`

كما وتدعم الكائنات العادية توابِع مثل تلك التوابع باختلاف بسيط في صياغتها.

5.9.1 التوابع keys وvalues وentries

هذه هي التوابع المتاحة للتعامل مع الكائنات العادية:

- Object.keys(obj) - يُعيد مصفوفة من مفاتيح الكائن obj المعطى.
- Object.values(obj) - يُعيد مصفوفة من قيم الكائن obj المعطى.
- Object.entries(obj) - يُعيد مصفوفة من أزواج مفاتيح/قيم `[key, value]` الكائن obj المعطى.

لاحظ رجاءً الفروق بينها وبين الخارطة مثلًا:

	الخارطة	الكائن
صياغة الاستدعاء	`map.keys()`	`Object.keys(obj)` لكن ليس `obj.keys()`
قيمة الإعادة	مُكرَّر (iterable)	مصفوفة "حقيقية"

أوّل فرق واضح جليّ: علينا استدعاء `Object.keys(obj)` لا `obj.keys()`. ولكن لماذا؟ السبب الأساس هو مرونة الاستعمال. لا تنسَ بأنّ الكائنات هي أساس كلّ بنية بيانات معقّدة في JavaScript، فيحدث بأنّ لدينا كائن طوّرناه ليحمل بيانات `data` مُحدَّدة، وفيه التابع `data.values()`، ولكنّا نريد أيضًا استدعاء `Object.values(data)` عليه.

الفرق الثاني هو أنّ التوابع `Object.*` تُعيد كائنات مصفوفات "فعلية" لا مُكرَّرات (iterable) فقط، ويعزو ذلك لأسباب تاريخية بحتة. خُذ هذا المثال:

```
let user = {
  name: "Ahmad",
  age: 30
};

Object.keys(user)    // ["name", "age"]
Object.values(user)  // ["Ahmad", 30]
Object.entries(user) // [ ["name","Ahmad"], ["age",30] ]
```

وهذا مثال آخر عن كيف نستعمل `Object.values` للمرور على قيم الخاصيات:

```
let user = {
  name: "Ahmad",
  age: 30
};

// نمرّ على القيم
for (let value of Object.values(user)) {
  alert(value); // 30 ثمّ Ahmad
}
```

تتجاهل هذه التوابع الخاصيات الرمزية

كما تتجاهل حلقة `for..in` الخاصيات التي تستعمل `Symbol(...)` مفاتيح لها، فهذه التوابع أعلاه تتجاهلها أيضًا

غالبًا يكون هذا ما نريد، ولكن لو أردت المفاتيح الرمزية أيضًا، فعليك استعمال التابع المنفصل Object.getOwnPropertySymbols إذ يُعيد مصفوفة بالمفاتيح الرمزية فقط. هناك أيضًا التابع Reflect.ownKeys(obj) إذ يُعيد المفاتيح كلها.

5.9.2 تعديل محتوى الكائنات

ليس للكائنات تلك التوابع المفيدة المُتاحة للعناصر (مثل `map` و`filter` وغيرها). لو أردنا تطبيق هذه التوابع على الكائنات فيجب أوّلًا استعمال `Object.entries` وبعدها `Object.fromEntries`:

1. استعمل `Object.entries(obj)` لتأخذ مصفوفة لها أزواج "مفتاح/قيمة" من الكائن `obj`.
2. استعمل توابِع المصفوفات على تلك المصفوفة (مثلًا `map`).

3. استعمل `Object.fromEntries(array)` على المصفوفة الناتج لتُحوّلها ثانيةً إلى كائن.

إليك مثالًا لدينا كائنًا فيه تسعير البضائع، ونريد مضاعفتها (إذ ارتفع الدولار):

```
let prices = {
  banana: 1,
  orange: 2,
  meat: 4,
};

let doublePrices = Object.fromEntries(
  // نحوّله إلى مصفوفة، ثمّ نستعمل map، ثمّ يُعيد إلينا fromEntries الكائن المطلوب
  Object.entries(prices).map(([key, value]) => [key, value * 2])
);
alert(doublePrices.meat); // 8
```

ربّما تراه صعبًا أوّل وهلة، ولكن لا تقلق فسيصير أسهل أكثر متى ما بدأت استعمالها مرّة واثنتان وثلاث، ويمكن بهذه الطريقة أن نصنع سلسلة فعّالة من التعديلات.

5.9.3 تمارين

ا. مجموع الخاصيات

الأهمية: ★★★★★

أمامك كائن `salaries` وفيه بعض الرواتب. اكتب دالة `sumSalaries(salaries)` تُعيد مجموع كلّ الرواتب، باستعمال `Object.values` وحلقة `for..of`. لو كان الكائن فارغًا فيجب أن يكون الناتج صفرًا 0.

مثال:

```
let salaries = {
  "Ahmad": 100,
  "Pete": 300,
  "Mary": 250
};
alert( sumSalaries(salaries) ); // 650
```

افتح المثال في بيئة تجربة حية.

الحل:

```
function sumSalaries(salaries) {

  let sum = 0;
  for (let salary of Object.values(salaries)) {
    sum += salary;
  }
  return sum; // 650
}

let salaries = {
  "Ahmad": 100,
  "Pete": 300,
  "Mary": 250
};
alert( sumSalaries(salaries) ); // 650
```

أو يمكننا (لو أردنا) معرفة المجموع باستعمال `Object.values` والتابع `reduce`:

```
// يمرّ reduce على مصفوفة من الرواتب،
// ويجمعها مع بعضها
// ويُعيد الناتج
function sumSalaries(salaries) {
  return Object.values(salaries).reduce((a, b) => a + b, 0) // 650
}
```

(اطلع على الحل في بيئة تجربة حية.)

ب. عدد الخاصيات

الأهمية: ★★★★★

اكتب دالة باسم `count(obj)` تُعيد عدد الخاصيات داخل الكائن:

```
let user = {
  name: 'Ahmad',
  age: 30
};
alert( count(user) ); // 2
```

حاوِل أن تكون الشيفرة بأصغر ما أمكن.

> أهمِل الخاصيات الرمزية وعُدّ فقط تلك "العادية".

(افتح المثال في بيئة تجربة حية.)

الحل:

```
function count(obj) {
  return Object.keys(obj).length;
}
```

(اطلع على الحل في بيئة تجربة حية.)

5.10 الإسناد بالتفكيك

الكائنات والمصفوفات في JavaScript هي أكثر بنى البيانات المستعملة. تُتيح لنا الكائنات إنشاء كيان واحد يُخزّن عناصر البيانات حسب مفاتيحها، وتُتيح لنا المصفوفات بجمع مختلف عناصر البيانات في تجميعة مرتّبة (ordered collection)، ولكن حين نُمرّرها هذه الكائنات والمصفوفات إلى دالة، غالبًا ما لا نريد كل الكائن/المصفوفة، بل بعضًا منها لا أكثر.

صياغة "الإسناد بالتفكيك" (Destructuring assignment) هي صياغة خاصّة تُتيح لنا "فكّ" المصفوفات أو الكائنات إلى مجموعة من المتغيرات إذ تكون أحيانًا أكثر منطقية. يفيدنا التفكيك أيضًا مع الدوال المعقّدة التي تحتوي على مُعاملات كثيرة وقيم مبدئية وغيرها وغيرها.

5.10.1 تفكيك المصفوفات

إليك مثال عن تفكيك مصفوفة إلى مجموعة من المتغيرات:

```
// معنا مصفوفة فيها اسم الشخص واسم عائلته
let arr = ["Ilya", "Kantor"]

// يضبط الإسناد بالتفكيك
// firstName = arr[0] هذه
// surname = arr[1] وهذه
let [firstName, surname] = arr;

alert(firstName); // Ilya
alert(surname);  // Kantor
```

يمكننا الآن العمل مع تلك المتغيرات عوض عناصر المصفوفة، وما إن تجمع تابِع `split` وغيرها من توابِع تُعيد مصفوفات، سترى بريق هذا التفكيك يتألق:

```
let [firstName, surname] = "Ilya Kantor".split(' ');
```

"التفكيك" (Destructuring) لا يعني "التكسير" (destructive)، فنُسمّيه "الإسناد بالتفكيك destructuring assignment" لأنّه "يفكّك" العناصر بنسخها إلى متغيرات، أمّا المصفوفة نفسها فتبقى دون تعديل. كتابة هذه الشيفرة أسهل من تلك الطويلة (ندعها لك تتخيّلها):

```
// let [firstName, surname] = arr;
let firstName = arr[0];
let surname = arr[1];
```

يمكنك "رمي" وتجاهل العناصر التي لا تريدها بإضافة فاصلة أخرى:

```
// لا نريد العنصر الثاني
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert( title ); // Consul
```

في الشيفرة هذه تخطّينا العنصر الثاني في المصفوفة وأسندنا الثالث إلى المتغير `title`، كما وتخطّينا أيضًا باقي عناصر المصفوفة (ما من متغيّرات لها).

تعمل الميزة مع المُكرَّرات (iterable) حين تكون على اليمين، إذ أنّنا نستطيع استعمالها مع أيّ مُكرَّر وليس المصفوفات فقط:

```
let [a, b, c] = "abc"; // ["a", "b", "c"]
let [one, two, three] = new Set([1, 2, 3]);
```

يمكن أن نستعمل أيّ متغيّر يمكن إسناده على الجانب الأيسر من سطر الإسناد. لاحظ مثلًا إسناد خاصية لكائن:

```
let user = {};
[user.name, user.surname] = "Ilya Kantor".split(' ');

alert(user.name); // Ilya
```

يمكن استعمال الإسناد بالتفكيك عند المرور على العناصر عبر `.entries()` رأينا في الفصل الماضي التابع Object.entries(obj). يمكننا استعماله مع التفكيك للمرور على مفاتيح الكائنات وقيمها:

```
let user = {
  name: "Ahmad",
  age: 30
};

// نمرّ على المفاتيح والقيم
for (let [key, value] of Object.entries(user)) {
  alert(`${key}:${value}`); // name:Ahmad, then age:30
}
```

...وذات الأمر للخارطة:

```
let user = new Map();
user.set("name", "Ahmad");
user.set("age", "30");

for (let [key, value] of user) {
  alert(`${key}:${value}`); // name:Ahmad, then age:30
}
```

ا. عامل البقية "..."

لو أردنا أخذ القيم الأولى إضافةً إلى كل ما يليها، فنُضيف عاملًا آخر يجلب "الباقي" باستعمال ثلاث نقاط "..." (لذلك ندعوه بعامل الباقي "..."):

```
let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius
alert(name2); // Caesar

// انتبه أنّ المتغير rest مصفوفة.
alert(rest[0]); // Consul
alert(rest[1]); // of the Roman Republic
alert(rest.length); // 2
```

ستكون قيمة المتغير rest مصفوفةً فيها عناصر الباقية في المصفوفة الأولى. يمكننا استعمال أيّ اسم آخر بدل rest، المهم أن يكون قبله ثلاث نقاط ويكون الأخير في جملة الإسناد بالتفكيك.

ب. القيم المبدئية

لو كانت القيم في المصفوفة أقلّ من تلك في جملة الإسناد فلن يحدث أيّ خطأ. يُعدّ المحرّك القيم "الغائبة" غير معرّفة:

```
let [firstName, surname] = [];

alert(firstName); // undefined
alert(surname); // undefined
```

لو أردنا قيمة مبدئية تعوّض تلك الناقصة فيمكننا تقديمها باستعمال =:

```
// القيم المبدئية
let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name);    // Julius (من المصفوفة)
alert(surname); // Anonymous (المبدئي)
```

يمكن أن تكون القيم المبدئية تعابيرَ معقّدة أو استدعاءات دوال حتّى. لن يقدّر ناتجها المحرّك إلّا لو لم تمرّر القيم تلك. فمثلًا يمكننا استعمال الدالة `promot` لأخذ قيمتين مبدئيتين. أمّا هنا فنستسأل عن القيمة الناقصة فقط:

```
// لا تطلب إلا اسم العائلة surname
let [name = prompt('name?'), surname = prompt('surname?')] =
["Julius"];

alert(name);    // Julius (نأخذه من المصفوفة)
alert(surname); // نحترم هنا ما يقول promot
```

5.10.2 تكفيك الكائنات

الإسناد بالتفكيك يدعم أيضًا الكائنات. هذه صياغته الأساس:

```
let {var1, var2} = {var1:…, var2:…}
```

على اليمين الكائن الموجود والذي نريد تقسيمه على عدّة متغيرات، وعلى اليسار نضع "نمط" الخاصيات المقابِلة له. لو كان الكائن بسيطًا، فهذا النمط هو قائمة باسم المتغيرات داخل `{...}`. مثال:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

let {title, width, height} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

تُسند الخاصيات `options.title` و `options.width` و `options.height` إلى المتغيرات المقابلِة لها. كما وأنّ الترتيب غير مهم: يمكنك فعل هذا أيضًا:

```
// let {...} غيّرنا الترتيب داخل
let {height, width, title} = { title: "Menu", height: 200, width: 100 }
```

يمكن أن يكون النمط على اليسار معقّدًا أكثر ومُحدّدًا فيحدّد طريقة ترابط الخاصيات بالمتغيرات عبر الخارطة (mapping). لو أردنا إسناد خاصية إلى متغير له اسم آخر فعلينا استعمال النقطتين الرأسيتين لذلك (مثلًا `options.width` يصير في المتغير `w`):

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

// { sourceProperty: targetVariable }
let {width: w, height: h, title} = options;

// width -> w
// height -> h
// title -> title

alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
```

تعني النقطتان الرأسيتان "هذا : يصير هذا". في المثال أعلاه، تصير الخاصية `width` بالاسم `w`، والخاصية `height` بالاسم `h` والخاصية `title` كما هي `title`.

يمكننا هنا أيضًا وضع قيمة مبدئية للخاصيات الناقصة باستعمال `"="` هكذا:

```
let options = {
  title: "Menu"
};

let {width = 100, height = 200, title} = options;
```

```
alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

كما يمكن أن تكون هذه القيم المبدئية أيّة تعابير أو استدعاءات دوال كما مقابلاتها في المصفوفات ومُعاملات الدوال، ولن يقيِّم المُحرِّك قيمتها إلّا لو لم تقدّم قيمة للدالة.

في الشيفرة أدناه، تطلب الدالة `promot` قيمة `width` ولا تطلب قيمة `title`:

```
let options = {
  title: "Menu"
};

let {width = prompt("width?"), title = prompt("title?")} = options;

alert(title);  // Menu
alert(width);  // (promot هنا نحترم أيضًا ما يقول)
```

يمكننا أيضًا جمع النقطتان الرأسيتان والقيم المبدئية:

```
let options = {
  title: "Menu"
};

let {width: w = 100, height: h = 200, title} = options;

alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
```

لو كان لدينا كائنًا معقّدًا فيه خاصيات كثيرة، فيمكننا استخراج ما نريد منه فقط:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

// استخرج العنوان title ليكون متغيرًا هو فقط
```

```
let { title } = options;

alert(title); // Menu
```

ا. نمط البقية "..."

ماذا لو كان للكائن خاصيات أكثر من المتغيرات التي لدينا، فهل يمكننا أخذها وإسنادها في متغيّر "rest" أيضًا؟ أجل يمكننا استعمال نمط الباقي تمامًا مثل المصفوفات. بعض المتصفحات القديمة لا تدعمه (مثل إنترنت إكسبلورر، استعمل Babel لترقيعه polyfill، أي لتعويض نقص الدعم)، إلّا أن الحديثة تدعمه.

هكذا نفعلها:

```
let options = {
  title: "Menu",
  height: 200,
  width: 100
};

// title = خاصية بالاسم title
// rest = كائن فيه باقي الخاصيات
let {title, ...rest} = options;

// صار الآن title="Menu", rest={height: 200, width: 100}
alert(rest.height);  // 200
alert(rest.width);   // 100
```

انتبه لو لم تضع `let`، ففي المثال أعلاه، صرّحنا عن المتغيرات على يمين جملة الإسناد: `let {…} = {…}`. يمكننا طبعًا استعمال متغيرات موجودة دون `let`، ولكن هناك أمر، فهذا لن يعمل:

```
let title, width, height;

// سترى خطأ في هذا السطر
{title, width, height} = {title: "Menu", width: 200, height: 100};
```

المشكلة هي أنّ JavaScript تتعامل مع `{...}` في سياق الشيفرة الأساس (أي ليس داخل تعبير آخر) على أنّها بنية كتلية Code Block، إذ يمكن استعمال البنى الكتلية هذه لجمع التعليمات البرمجية، هكذا:

```
{
  // بنية شيفرة
  let message = "Hello";
  // ...
  alert( message );
}
```

وهنا يظنّ محرّك JavaScript بأنّ هذه بنية كتلية فيعطينا الخطأ أعلاه، بينما ما نريد هو التفكيك. ولنقول للمحرّك بأنّ هذه ليست بنية شيفرة، نضع التعبير بين قوسين `(...)`:

```
let title, width, height;

// الآن جيد
({title, width, height} = {title: "Menu", width: 200, height: 100});

alert( title ); // Menu
```

5.10.3 تفكيك المتغيرات المتداخلة

لو كان في الكائن أو المصفوفة كائنات ومصفوفات أخرى داخله، فيمكننا استعمال أنماط معقّدة على يسار جملة الإسناد لنستخرج تلك المعلومات.

في الشيفرة أدناه، نجد داخل الكائن `options` كائنًا آخر في الخاصية `size`، ومصفوفة في الخاصية `items`. النمط على يسار جملة الإسناد لديه ذات البنية تلك لتستخرج هذه القيم من الكائن على يمينه:

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

// نقسم الإسناد بالتفكيك على أكثر من سطر لتوضيح العملية
let {
  size: { // هنا يكون المقاس
    width,
```

```
    height
  },
  items: [item1, item2], // وهنا نضع العناصر
  title = "Menu" // (ستُستعمل القيمة المبدئية) ليست موجودة في الكائن
} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
alert(item1);  // Cake
alert(item2);  // Donut
```

هكذا تُسند كلّ خاصيات `options` (عدا `extra` الناقصة يسار عبارة الإسناد) إلى المتغيرات المقابلة لها:

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
}
```

```
let {
  size: {
    width,
    height
  },
  items: [item1, item2],
  title = "Menu"
}
```

وفي النهاية يكون لدينا المتغيّرات `width` و`height` و`item1` و`item2` و`title` من تلك القيمة المبدئية.

لاحظ ألّا وجود لمتغيّرات تنسخ `size` و`items` إذ ما نريد هو محتواها لا هي.

5.10.4 مُعاملات الدوال الذكية

أحيانًا وأنت تعمل تجد نفسك تكتب دالة لها مُعاملات كثيرة وأغلبها اختيارية. يحدث هذا غالبًا مع دوال واجهات المستخدم. عُدّ أنّ لديك دالة تُنشئ قائمة، وللقائمة عَرض وارتفاع وعنوان وقائمة عناصر وغيرها.

هكذا تصنع تلك الدالة بالأسلوب الخطأ:

```
function showMenu(title = "Untitled", width = 200, height = 100, items = []) {
  // ...
}
```

تكمن المشكلة (في الحياة الواقعية) في تذكّر ترتيب تلك الوُسطاء. صحيح أنّ بيئات التطوير تفيدنا هنا عادةً -خصوصًا لو كان المشروع موثّق توثيقًا ممتازًا- ولكن مع ذلك فالمشكلة الأخرى هي طريقة استدعاء الدالة لو كانت كلّ مُعاملاتها المبدئية مناسبة لنا.

نستدعيها هكذا؟

```
// نضع undefined لو كانت القيم المبدئية تقوم بالغرض
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])
```

جريمة بحقّ الجمال. ورويدًا رويدًا تصير مستحيلة القراءة حين نُضيف مُعاملات أخرى.

التفكيك لنا بالمرصاد... أعني للعون! فيمكننا تمرير المُعاملات بصيغة كائن، وستُفكّكها الدالة حالًا في متغيرات:

```
// نمرّر كائنًا إلى الدالة
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

// ...ومباشرة تفكّها وتضعها في متغيرات
function showMenu({title = "Untitled", width = 200, height = 100,
items = []}) {
  // title, items - هذه من options
  // width, height - نستعمل القيم المبدئية
  alert( `${title} ${width} ${height}` ); // My Menu 200 100
  alert( items ); // Item1, Item2
}

showMenu(options);
```

يمكننا أيضًا استعمال التفكيك الأكثر تعقيدًا (مع الكائنات المتداخلة وتغيير الأسماء بالنقطتين الرأسيتين):

```
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};



function showMenu({
  title = "Untitled",
  width: w = 100,  // نضع width في w
  height: h = 200, // ونضع height في h
```

```
  items: [item1, item2] // أوّل عنصر في items يصير item1، وثاني عنصر يصير item2
}) {

  alert( `${title} ${w} ${h}` ); // My Menu 100 200
  alert( item1 ); // Item1
  alert( item2 ); // Item2
}

showMenu(options);
```

صياغة الدالة الكاملة تتطابق مع صياغة الإسناد بالتفكيك:

```
function({
  incomingProperty: varName = defaultValue
  ...
})
```

وحينها متى ما تمرّر كائن على أساس أنّه مُعامل، نضع الخاصية `incomingProperty` في المتغير `varName` وقيمته المبدئية هي `defaultValue`.

لاحظ بأنّ هذا النوع من التفكيك ينتظر مُعاملًا واحدًا على الأقل في الدالة `showMenu()`. لو أردنا أن تكون كلّ القيم كما هي مبدئيًا، فعلينا تقديم كائن فارغ:

```
showMenu({}); // هكذا، كل القيم كما هي مبدئيًا

showMenu(); // هذا سيصرخ علينا بخطأ
```

يمكننا إصلاح هذه المشكلة بتحديد `{}` قيمةً مبدئيةً لكامل الكائن الذي يحوي المُعاملات:

```
function showMenu({ title = "Menu", width = 100, height = 200 } = {})
{
  alert( `${title} ${width} ${height}` );
}

showMenu(); // Menu 100 200
```

5.10.5 الخلاصة

- يتيح الإسناد بالتفكيك ربط الكائن أو المصفوفة مع متغيرات عديدة أخرى، وآنيًا.
- صياغة الكائن الكاملة هي:

```
let {prop : varName = default, ...rest} = object
```

ويعني هذا بأنّ الخاصية `prop` تصير في المتغيّر `varName`، وفي حال لم توجد هذه الخاصية فستُستعمَل القيمة المبدئية `default`.

تُنسخ حاصيات الكائنات التي لا ترتبط إلى الكائن `rest`.

- صياغة المصفوفة الكاملة هي:

```
let [item1 = default, item2, ...rest] = array
```

يصير أوّل عنصر في `item1` وثاني عنصر في `item2` وباقي المصفوفة يصير باقيًا في `rest`.

- يمكن أيضًا استخراج البيانات من المصفوفات/الكائنات المتداخلة، ويلزم أن تتطابق بنية على يسار الإسناد تلك على يمينه.

5.10.6 تمارين

ا. الإسناد بالتفكيك

الأهمية: ★★★★★

لدينا هذا الكائن:

```
let user = {
  name: "Ahmad",
  years: 30
};
```

اكتب إسنادًا بالتفكيك يقرأ:

- خاصية `name` ويضعها في المتغير `name`.
- خاصية `years` ويضعها في المتغير `age`.
- خاصية `isAdmin` ويضعها في المتغير `isAdmin` (تكون false لو لم تكن موجودة)

إليك مثالًا بالقيم بعد إجراء الإسناد:

```
let user = { name: "Ahmad", years: 30 };

// ضع شيفرتك على الجانب الأيسر:
// ... = user

alert( name ); // Ahmad
alert( age ); // 30
alert( isAdmin ); // false
```

الحل:

```
let user = {
  name: "Ahmad",
  years: 30
};

let {name, years: age, isAdmin = false} = user;

alert( name ); // Ahmad
alert( age ); // 30
alert( isAdmin ); // false
```

ب. أكبر راتب

الأهمية: ★★★★★

إليك كائن الرواتب `salaries`:

```
let salaries = {
  "Ahmad": 100,
  "Pete": 300,
  "Mary": 250
};
```

اكتب دالة `topSalary(salaries)` تُعيد اسم الشخص الأكثر ثراءً وراتبًا.

- لو كان `salaries` فارغًا فيجب أن تُعيد `null`.
- لو كان هناك أكثر من شخص متساوي الراتب، فتُعيد أيًا منهم.

> استعمل `Object.entries` والإسناد بالتفكيك للمرور على أزواج "مفاتيح/قيم".

(افتح التمرين في بيئة تجريبية حية).

الحل:

```
function topSalary(salaries) {

  let max = 0;
  let maxName = null;

  for(const [name, salary] of Object.entries(salaries)) {
    if (max < salary) {
      max = salary;
      maxName = name;
    }
  }

  return maxName;
}
```

(اطلع على الحل في بيئة تجريبية حية).

5.11 النوع Date: التاريخ والوقت

حان وقت الحديث عن كائن آخر مضمّن في اللغة: التاريخ Date. يخزّن هذا الكـائن التـاريخ والـوقت ويقـدّم توابِع تُدير أختام التاريخ والوقت. يمكننا مثلًا استعماله لتخزين أوقات الإنشاء/التعديل أو حساب الوقت أو طباعة التاريخ الحالي في الطرفية.

5.11.1 الباني

استدعِ `new Date()` بتمرير واحدًا من الوُسطاء الآتية فتصنع كائن `Date` جديد:

أ. new Date()

استدعاء new Date() بلا وسطاء يُنشئ كائن `Date` بالتاريخ والوقت الحاليين:

```
let now = new Date();
alert( now ); // نعرض التاريخ والوقت الحاليين
```

ب. new Date(milliseconds)

يُنشئ كائن `Date` إذ تسـاوي قيمتـه عـدد المليثـوان milliseconds المُمـرَّرة (المليثانيـة هي 1/1000 من الثانية) حسابًا من بعد الأول من يناير عام 1970 بتوقيت UTC+0.

```
// UTC+0
// 01.01.1970 نعني بـ 0 التاريخ
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );

// 02.01.1970  نضيف الآن 24 ساعة لنحصل على
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
```

يُسمّى العدد الصحيح الذي يمثّل عدد المليثوان التي مـرّت من بدايـة عـام 1970 بـالختم الزمـني أو بصـمة وقت (timestamp)، وهو يمثّل التاريخ بنحوٍ عددي خفيف (lightweight). يمكننا إنشاء التـواريخ من الأختـام الزمنية باستعمال `new Date(timestamp)` وتحويل كائن التاريخ `Date` الموجـود إلى ختم زمـني باسـتعمال التابع `date.getTime()` (طالع أسفله).

والتواريخ قبل الأول من يناير 1970 أختامها سالبة:

```
// 31 ديسمبر 1969
    let Dec31_1969 = new Date(-24 * 3600 * 1000);
    alert( Dec31_1969 );
```

ج. new Date(datestring)

لو كان هناك وسيط واحد وكان سلسلة نصيّة، فسيُحلّله المحرّك تلقائيًا. الخوارزمية هنا هي ذات التي يستعملها `Date.parse`. لا تقلق، سنتكلم عنه لاحقًا.

```
let date = new Date("2017-01-26");
alert(date);
```

نجد في هذا المثال أن الوقت غير محدد لذا يكون بتوقيت GMT منتصف الليل، ويحدد وفقًا للمنطقة الزمنية التي تنفذ الشيفرة ضمنها، فالنتيجة يمكن أن تكون Thu Jan 26 2017 11:00:00 للبلدان ذات المنطقة الزمنية GMT+1100 أو يمكن أن تكون Wed Jan 25 2017 16:00:00 للبلدان الواقعة في المنطقة الزمنية GMT-0800.

د. new Date(year, month, date, hours, minutes, seconds, ms)

يُنشئ تاريخًا بالمكوّنات المُمرَّرة حسب المنطقة الزمنية المحلية. أوّل وسيطين إلزاميين أما البقية اختيارية.

- يجب أن يكون العام `year` بأربع خانات: `2013` صح، `98` خطأ.
- يبدأ الشهر `month` بالرقم `0` (يناير) وينتهي بالعدد `11` (ديسمبر).
- مُعامل التاريخ `date` هو رقم اليوم من الشهر. لو لم يكن موجودًا فسيعدّه الكائن `1`.
- لو لم تكن مُعاملات الساعة والدقيقة والثانية والمليثانية `hours/minutes/seconds/ms` موجودة، فسيعدّها الكائن `0`.

إليك المثال التالي:

```
new Date(2011, 0, 1, 0, 0, 0, 0); // 1 Jan 2011, 00:00:00
new Date(2011, 0, 1); // نفس تلك. الساعات والدقائق وغيرها 0 مبدئيًا
// أدنى دقّة للتاريخ هي مليثانية واحدة (واحد من ألف من الثانية)
let date = new Date(2011, 0, 1, 2, 3, 4, 567);
alert( date ); // 1.01.2011, 02:03:04.567
```

5.11.2 الوصول إلى مكوّنات التاريخ

إليك التوابع التي تتيح لك الوصول إلى العام والشهر وغيرها داخل كائن `Date`:

- getFullYear()‎: يجلب العام (4 خانات).
- getMonth()‎: يجلب الشهر، من 0 إلى 11.
- getDate()‎: يجلب رقم اليوم من الشهر، من 1 إلى 31. قد يبدو الاسم غريبًا قليلًا لك.
- getHours()‎ وgetMinutes()‎ وgetSeconds()‎ وgetMilliseconds()‎: تجلب هذه التوابع مكوّنات الوقت حسب كل تابِع: الساعة/الدقيقة/الثانية/المليثانية على التوالي.

إياك باستعمال getYear()‎ بدل getFullYear()‎

تقدّم الكثير من محرّكات JavaScript التابع غير القياسي `getYear()`. هذا التابع أصبح بائدًا، فهو يُعيد العام بخانتين أحيانًا. من فضلك لا تستعمله أبدًا، بل استعمل `getFullYear()` لتجلب العام.

كما يمكن أيضًا جلب رقم اليوم من الأسبوع عبر التابع التالي:

ا. التابع getDay()‎

يجلب getDay()‎ رقم اليوم من الأسبوع، بدءًا بِـ 0 (الأحد) وحتى 6 (السبت). أوّل يوم هو الأحد دومًا، إذ صحيح أنّ في بعض الدول هذا غير صحيح، لكن لا يمكن تغيير القيمة إطلاقًا.

تُعيد كلّ التوابع أعلاه المكوّنات حسب المنطقة الزمنية المحلية

توجد أيضًا مثيلاتها بنظام UTC حيث تُعيد اليوم والشهر والعام وغيرها في المنطقة الزمنية UTF+0: getUTCFullYear()‎ وgetUTCMonth()‎ وgetUTCDay()‎. ضع كلمة `"UTC"` بعد `"get"` وستجد المثيل المناسب. لو كانت منطقتك الزمنية المحلية بعيدة عن UTC، فستعرض الشيفرة أدناه الساعات مختلفة عن بعضها البعض:

```
// التاريخ الحالي
let date = new Date();

// الساعة حسب المنطقة الزمنية التي أنت فيها
alert( date.getHours() );

// الساعة حسب المنطقة الزمنية بتوقيت UTC+0 (أي توقيت لندن بدون التوقيت الصيفي)
alert( date.getUTCHours() );
```

هناك (إضافةً إلى هذه التوابع) تابِعان آخران مختلفان قليلًا ليس لهما نُسخ بتوقيت UTC:

ب. التابع ()getTime

يُعيد ()getTime ختم التاريخ الزمني، أي عدد المليثوان التي مرّت منذ الأول من يناير عام 1970 بتوقيت UTC+0.

ج. التابع ()GetTimezoneOffset

يُعيد ()GetTimezoneOffset الفرق بين المنطقة الزمنية الحالية وتوقيت UTC (بالدقيقة):

```
// لو كانت منطقتك الزمنية UTC-1، فالناتج 60
// لو كانت منطقتك الزمنية UTC+3، فالناتج -180
alert( new Date().getTimezoneOffset() );
```

5.11.3 ضبط مكونات التاريخ

تتيح لك التوابع الآتية ضبط مكوّنات التاريخ والوقت:

- العام: setFullYear(year, [month], [date])
- الشهر: setMonth(month, [date])
- التاريخ: setDate(date)
- الساعة: setHours(hour, [min], [sec], [ms])
- الدقيقة: setMinutes(min, [sec], [ms])
- الثانية: setSeconds(sec, [ms])
- المليثانية: setMilliseconds(ms)
- الوقت بالمليثانية: setTime(milliseconds) (تضبط التاريخ كلّه حسب عدد المليثوان منذ UTC 01.01.1970)

لدى كلّ تابع منها نسخة بتوقيت UTC (عدا `setTime()`). مثال: `setUTCHours()`. كما رأيت فيمكن لبعض التوابع ضبط عدّة مكوّنات في آن واحد مثل `setHours`. المكوّنات التي لا تُمرّر لا تُعدّل.

مثال:

```
let today = new Date();

today.setHours(0);
alert(today); // ما زال اليوم نفسه، ولكن الساعة تغيّرت إلى 0
```

```
today.setHours(0, 0, 0, 0);
alert(today); // ما زال اليوم نفسه، ولكنّا عند 00:00:00 تمامًا.
```

5.11.4 التصحيح التلقائي

ميزة "التصحيح التلقائي" في كائنات التواريخ Date مفيدة جدًا لنا، إذ يمكن أن نضع قيم تاريخ لامنطقية (مثل الخمسون من هذا الشهر) وسيُعدّلها الكائن بنفسه.

مثال:

```
let date = new Date(2013, 0, 32); // الثاني والثلاثين من يناير 2013؟!
alert(date); // آه، تقصد الأول من فبراير 2013!...
```

تترتّب المكوّنات اللامنطقية تلقائيًا. فمثلًا لو أضفت على التاريخ "28 فبراير 2016" يومين اثنين، فيمكن أن يكون "الثاني من مارس" أو "الأول من مارس" لو كانت السنة كبيسة. بدل أن نفكّر بهذا الحساب، نُضيف يومين ونترك الباقي على كائن Date:

```
let date = new Date(2016, 1, 28);
date.setDate(date.getDate() + 2);

alert( date ); // 1 مارس 2016
```

غالبًا ما تُستعمل هذه الميزة لنجلب التاريخ بعد فترة محدّدة من الزمن. فلنقل مثلًا نريد تاريخ "70 ثانية من الآن":

```
let date = new Date();
date.setSeconds(date.getSeconds() + 70);

alert( date ); // يعرض التاريخ الصحيح
```

يمكننا أيضًا ضبط القيمة لتكون صفرًا أو حتّى بالسالب. مثال:

```
let date = new Date(2016, 0, 2); // 2 يناير 2016

date.setDate(1); // نضبط التاريخ على أول يوم من الشهر
alert( date );

// أقل يوم ممكن هو 1، إذًا فيعدّ الكائن أنّ 0 هو آخر يوم من الشهر الماضي
```

```
date.setDate(0);
alert( date ); // 31 ديسمبر 2015
```

5.11.5 تحويل التاريخ إلى عدد، والفرق بين تاريخين

حين يتحوّل كائن Date إلى عدد يصير ختمًا زمنيًا مطابقًا لختم date.getTime():

```
let date = new Date();
alert(+date); // عدد المليثوان، نفس ناتج date.getTime()
```

تأثير هذا المهم والخطير هو أنّك تستطيع طرح التواريخ من بعض، والناتج سيكون الفرق بينهما بالمليثانيـة. يمكن استعمال الطرح لحساب الأوقات:

```
let start = new Date(); // نبدأ قياس الوقت

// إلى العمل
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

let end = new Date(); // ننتهي من قياس الوقت

alert( `The loop took ${end - start} ms` );
```

5.11.6 التاريخ الآن

لو أردنا قياس الوقت فقط فلا نحتاج كائن Date، بل هناك تابِعًا خاصًا باسم Date.now() يُعيد لنـا الختم الزمني الحالي.

يُكافئ هذا التابع الجملةَ new Date().getTime() إلّا أنّه لا يُنشئ كائن Date يتوسّط العمليـة، ولهـذا هو أسر ع ولا يزيد الضغط على عملية كنس المهملات. غالبًا ما يُستعمَل التابع لأنّه أسـهل أو لأنّ الأداء في تلـك الحالة مهم، مثلما في الألعاب بلغة JavaScript أو التطبيقات المتخصّصة الأخرى. ولهذا قد يكون الأفضل كتابة الشيفرة أدناه بدل تلك:

```
let start = Date.now(); // تبدأ المليثوان من تاريخ 1 يناير 1970

// إلى العمل
for (let i = 0; i < 100000; i++) {
```

```
    let doSomething = i * i * i;
  }

  let end = Date.now(); // انتهينا

  alert( `The loop took ${end - start} ms` ); // نطرح الأعداد لا التواريخ
```

5.11.7 قياس الأداء

لو أردنا قياس أداء دالة شرهة في استعمال المعالج، فعلينا أن نكون حذرين، هذا لو أردنا التعويل على القياس. فلنقيس مثلًا دالتين اثنتين تحسبان الفرق بين تاريخين: أيهما أسرع؟ نُطلق على قياسات الأداء هذه... "قياسات أداء" (Benchmark).

في المثال التالي، أمامنا date1 وdate2، أيّ دالة ستُعيد الفرق بينهما (بالمليثانية) أسرع من الأخرى؟

```
// هل هذه؟
function diffSubtract(date1, date2) {
  return date2 - date1;
}

// أم هذه...
function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}
```

وظيفة الدالتين متطابقة تمامًا، إلّا أن الثانية تستعمل التابع `date.getTime()` الصريح لتجلب التاريخ بالمليثانية، بينما الأخرى تعتمد على تحويل التاريخ إلى عدد. الناتج متطابق دومًا.

إذًا بهذه المعطيات، أيّ الدالتين أسرع؟

أوّل فكرة تخطر على البال هو تشغيل كلّ واحدة مرات عديدة متتابعة وقياس فرق الوقت. الدوال (في حالتنا هذه) بسيطة جدًا، ولهذا علينا تشغيل كلّ واحدة مئة ألف مرة على الأقل.

هيا نقيس الأداء:

```
function diffSubtract(date1, date2) {
  return date2 - date1;
}
```

```
function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}

function bench(f) {
  let date1 = new Date(0);
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

alert( 'Time of diffSubtract: ' + bench(diffSubtract) + 'ms' );
alert( 'Time of diffGetTime: ' + bench(diffGetTime) + 'ms' );
```

عجبًا! استعمال التابع `getTime()` أسرع بكثير! يعزو ذلك بسبب انعدام وجود تحويل للنوع (type conversion)، وهذا يُسهِّل على المحرّكات تحسين الأداء.

جميل! وصلنا إلى شيء، ولكنّ هذا القياس ليس قياسًا طيبًا بعد. تخيّل أنّ المعالج كان ينفّذ أمرًا ما بالتوازي مع تشغيل `bench(diffSubtract)` وكان يستهلك الموارد، وما إن شغّلنا `bench(diffGetTime)` كان ذلك الأمر قد اكتمل. هذا التخيّل هو تخيّل طبيعي لأمر واقعيّ جدًا حيث اليوم أنظمة التشغيل متعدّدة المهام. بهذا يكون لمرة القياس الأولى موارد معالجة أقل من المرة الثانية، ما قد يؤدّي إلى نتائج قياس خطأ.

إن أردنا التعويل على قياس الأداء، علينا إعادة تشغيل كل قياسات الأداء الموجودة أكثر من مرّة. هكذا مثلًا:

```
function diffSubtract(date1, date2) {
  return date2 - date1;
}

function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}

function bench(f) {
  let date1 = new Date(0);
```

```
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

let time1 = 0;
let time2 = 0;

// نشغّل bench(upperSlice) وbench(upperLoop) عشر مرات مرّة بمرّة
for (let i = 0; i < 10; i++) {
  time1 += bench(diffSubtract);
  time2 += bench(diffGetTime);
}

alert( 'Total time for diffSubtract: ' + time1 );
alert( 'Total time for diffGetTime: ' + time2 );
```

لا تبدأ محرّكات JavaScript الحديثة بتطبيق التحسينات المتقدّمة إلّا على "الشيفرات الحرجة" الـتي تُنفَّذ أكثر من مرّة (لا داعٍ بتحسين شيفرة نادرة التنفيذ). بهذا في المثال الأول، قد لا تكون مرات التنفيذ الأولى محسّنة كما يجب، وربما علينا إضافة تحمية سريعة:

```
// أضفناه لـ"تحمية" المحرّك قبل الحلقة الأساس
bench(diffSubtract);
bench(diffGetTime);

// الآن نقيس
for (let i = 0; i < 10; i++) {
  time1 += bench(diffSubtract);
  time2 += bench(diffGetTime);
}
```

الزم الحذر متى ما أجريت قياسات أداء على المستوى الذرّي

تُنفّذ محرّكات JavaScript الحديثة عددًا كبيرًا من التحسينات، وقد تُغيّر نتائج "الاختبارات الصناعية" موازنةً "بالاستعمال الطبيعي لها"، خصوصًا حين نقيس أداء ما هو صغير للغاية مثل طريقة عمل مُعامل رياضي، أو دالة مضمّنة في اللغة نفسها. لهذا، لو كنت تريد حقًا فهم الأداء كما يجب، فمن فضلك تعلّم طريقة عمل محرّك JavaScript حينها ربّما لن تحتاج هذه القياسات على المستوى الذرّي، أبدًا.

يمكنك أن تقرأ بعض المقالات الرائعة حول المحرّك V8 هنا http://mrale.ph.

5.11.8 تحليل سلسلة نصية باستعمال Date.parse

يمكن أن يقرأ التابع Date.parse(str) تاريخًا من سلسلة نصية. يجب أن يكون تنسيق تلك السلسلة هكذا: `YYYY-MM-DDTHH:mm:ss.sssZ`، إذ تعني:

- `YYYY-MM-DD`: التاريخ: اليوم-الشهر-العام.
- يُستعمل المحرف `"T"` فاصِلًا.
- `HH:mm:ss.sss`: الوقت: المليثانية والثانية والدقيقة والساعة.
- يمثّل الجزء الاختياري `'Z'` المنطقة الزمنية حسب التنسيق `+-hh:mm`. لو وضعت Z فقط فذلك يعني UTC+0.

يمكنك أيضًا استعمال تنسيقات أقصر مثل `YYYY-MM-DD` أو `YYYY-MM` أو حتّى YYYY.

باستدعاء `Date.parse(str)` فالسلسلة النصية تُحلّل حسب التنسيق فيها ويُعيد التابع الختم الزمني (رقم المليثوان منذ الأول من يناير 1970 بتوقيت UTC+0). لو كان التنسيق غير صحيح فيُعيد NaN.

إليك مثالًا:

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert(ms); // 1327611110417  (ختم زمني)
```

يمكننا إنشاء كائن `new Date` مباشرةً من الختم الزمني:

```
let date = new Date( Date.parse('2012-01-26T13:51:50.417-07:00') );

alert(date);
```

5.11.9 الخلاصة

- يُمثّل التاريخ والوقت في JavaScript بكائن Date. لا يمكننا إنشاء "تاريخ فقط" أو "وقتًا فقط"، فعلى كائنات التاريخ `Date` احتواء الاثنين معًا.
- تُعدّ الأشهر بدءًا بالصفر (يناير هو الشهر صفر، نعم).
- يُعدّ رقم اليوم من الأسبوع في `getDay()` من الصفر أيضًا (وهو يوم الأحد).
- يصحّح كائن التاريخ نفسه تلقائيًا حين تُضبط مكوّناته بقيم لا منطقية. يفيدنا لجمع/طرح الأيام والأشهر والأعوام.
- يمكن طرح التواريخ ومعرفة الفرق بينها بالمليثانية، ذلك لأنّ كائن التاريخ يتحوّل إلى ختم زمني حين يتحوّل إلى عدد.
- استعمل `Date.now()` لو أردت جلب الختم الزمني الحالي بسرعة.

لاحظ بأنّ الأختام الزمنية في JavaScript هي بالمليثانية، على العكس من أنظمة عديدة أخرى.

نجد نفسنا بين الحين والآخر قياسات وقت دقيقة. للأسف فلا توفّر JavaScript نفسها طريقة لحساب الوقت بالنانوثانية (1 على مليون من الثانية)، ولكن أغلب بيئاتها توفّر ذلك. فمثلًا تملك المتصفّحات التابع performance.now() إذ يُعيد عدد المليثوان منذ بدأ تحميل الصفحة بقدّة تصل إلى المايكروثانية (ثلاث خانات بعد الفاصلة):

```
alert(`Loading started ${performance.now()}ms ago`);
// "Loading started 34731.26000000001ms ago" :تظهر هكذا
```

تعني ".26" هنا المايكروثوان (260 مايكروثانية)، فلو زِدت على ثلاث خانات بعد الفاصلة فستجد أخطاءً في دقّة الحساب. أوّل ثلاثة هي الصحيحة فقط.

تملك لغة Node.js أيضًا وحدة `microtime` وأخرى غيرها. يمكن (تقنيًا) لأيّ جهاز أو بيئة أن تعطينا دقّة وقت أعلى، `Date` لا تقدّم ذلك لا أكثر.

5.11.10 تمارين

ا. إنشاء تاريخ

الأهمية: ★★★★★

أنشِئ كائن `Date` لهذا التاريخ: 20 فبراير 2012ـ 3:12 صباحًا. المنطقة الزمنية هي المحلية. اعرض التاريخ باستعمال `alert`.

الحل:

يستعمل مُنشِئ `new Date` المنطقة الزمنية الحالية. عليك ألا تنسى بأنّ الأشهر تبدأ من الصفر.

إذًا ففبراير هو الشهر رقم 1.

```
let d = new Date(2012, 1, 20, 3, 12);
alert( d );
```

ب. اعرض اسم اليوم من الأسبوع

الأهمية: ★★★★★

اكتب دالة `getWeekDay(date)` تعرض اسم اليوم من الأسبوع بالتنسيق الإنكليزي القصير: 'MO', 'TU', 'WE', 'TH', 'FR', 'SA', 'SU'.

مثال:

```
let date = new Date(2012, 0, 3);  // 3 يناير 2012
alert( getWeekDay(date) );        // "TU" يجب أن يطبع
```

(تجربة حية للتمرين)

الحل:

يُعيد التابع `date.getDay()` رقم اليوم من الأسبوع، بدءًا من يوم الأحد.

لنصنع مصفوفة فيها أيام الأسبوع لنعرف اليوم الصحيح من رقمه:

```
function getWeekDay(date) {
  let days = ['SU', 'MO', 'TU', 'WE', 'TH', 'FR', 'SA'];
  return days[date.getDay()];
}

let date = new Date(2014, 0, 3); // 3 يناير 2014
alert( getWeekDay(date) ); // FR
```

(تجربة حية للحل)

ج. اليوم من الأسبوع في أوروبا

الأهمية: ★★★★☆

في الدول الأوروبية، يبدأ الأسبوع بيوم الإثنين (رقم 1) وثمّ الثلاثاء (رقم 2) وحتّى الأحد (رقم 7). اكتب دالة `getLocalDay(date)` تُعيد يوم الأسبوع "الأوروبي" من التاريخ `date`.

```
let date = new Date(2012, 0, 3);  // 3 يناير 2012
alert( getLocalDay(date) );       // يكون يوم ثلاثاء، يجب أن تعرض 2
```

(تجربة حية للتمرين)

الحل:

```
function getLocalDay(date) {

  let day = date.getDay();

  if (day == 0) {
    // يوم الأحد 0 في أوروبا هو الأخير (7)
    day = 7;
  }

  return day;
}
```

(تجربة حية للحل)

د. ما هو التاريخ الذي كان قبل كذا يوم؟

الأهمية: ★★★★☆

أنشِئ دالة `getDateAgo(date, days)` تُعيد بتمرير التاريخ `date` اسم اليوم من الشهر قبل فترة `days` يوم.

مثال: لو كان اليوم العشرون من الشهر، فتُعيد `getDateAgo(new Date(), 1)` التاسع عشر و `getDateAgo(new Date(), 2)` الثامن عشر.

يجب أن نعوّل بأن تعمل الدالة في حال `days=356` وأكثر حتّى:

```
let date = new Date(2015, 0, 2);
```

```
alert( getDateAgo(date, 1) ); // 1، (1 يناير 2015)
alert( getDateAgo(date, 2) ); // 31، (31 ديسمبر 2014)
alert( getDateAgo(date, 365) ); // 2، (2 يناير 2014)
```

ملاحظة: يجب ألّا تُعدّل الدالة التاريخ `date` المُمرّر.

(تجربة حية للتمرين)

الحل:

الفكرة بسيطة، أن نطرح عدد الأيام من التاريخ `date`:

```
function getDateAgo(date, days) {
  date.setDate(date.getDate() - days);
  return date.getDate();
}
```

ولكن... يجب ألّا تُعدّل الدالة على `date`. هذا مهم إذ أنّ الشيفرة خارج الدالة التي تُعطينا التاريخ لا تريد منّا تغييره. لننفّذ ذلك، علينا نسخ التاريخ هكذا أولًا:

```
function getDateAgo(date, days) {
  let dateCopy = new Date(date);

  dateCopy.setDate(date.getDate() - days);
  return dateCopy.getDate();
}

let date = new Date(2015, 0, 2);

alert( getDateAgo(date, 1) ); // 1، (1 يناير 2015)
alert( getDateAgo(date, 2) ); // 31، (31 ديسمبر 2014)
alert( getDateAgo(date, 365) ); // 2، (2 يناير 2014)
```

(تجربة حية للحل)

ه. آخر يوم من الشهر كذا؟

الأهمية: ★★★★★

اكتب دالة `getLastDayOfMonth(year, month)` تُعيد آخر يوم من الشهر. أحيانًا يكون الثلاثين، أو الحادي والثلاثين أو الثامن/التاسع عشر من فبراير.

المُعاملات:

- `year`: العام بأربع خانات، مثلًا 2012.
- `month`: الشهر من 0 إلى 11.

مثال: `getLastDayOfMonth(2012, 1) = 29` (سنة كبيسة، فبراير).

(تجربة حية للتمرين)

الحل:

فلنصنع تاريخًا باستعمال الشهر التالي، ولكنّ نمرّر الصفر ليكون رقم اليوم:

```
function getLastDayOfMonth(year, month) {
  let date = new Date(year, month + 1, 0);
  return date.getDate();
}

alert( getLastDayOfMonth(2012, 0) ); // 31
alert( getLastDayOfMonth(2012, 1) ); // 29
alert( getLastDayOfMonth(2013, 1) ); // 28
```

عادةً ما تبدأ التواريخ بالواحد، لكن يمكننا (تقنيًا) تمرير أيّ عدد وسيُعدّل التاريخ نفسه. لذا حين نمرّر 0 نعني بذلك "يومًا واحد قبل الأول من الشهر"، أي "اليوم الأخير من الشهر الماضي".

(تجربة حية للحل)

و. كم من ثانية مضت اليوم؟

الأهمية: ★★★★★

اكتب دالة `getSecondsToday()` تُعيد عدد الثواني منذ بداية هذا اليوم. فمثلًا لو كانت الساعة الآن `10:00 am`، وبدون التوقيت الصيفي، فستعطينا الدالة:

```
getSecondsToday() == 36000 // (3600 * 10)
```

يجب أن تعمل الدالة مهما كان اليوم. أيّ ألا تحتوي على قيمة داخلها بتاريخ "اليوم"... اليوم.

الحل:

لنعرف عدد الثواني يمكننا توليد تاريخًا باستعمال اليوم الحالي والساعة 00:00:00، وثمّ نطرح منها "الوقت والتاريخ الآن". سيكون الفرق حينها بعدد المليثوان منذ بداية هذا اليوم، فنقسمه على 1000 لنعرف الثواني فقط:

```
function getSecondsToday() {
  let now = new Date();

  // أنشِئ كائنًا باستعمال اليوم والشهر والسنة حاليًا
  let today = new Date(now.getFullYear(), now.getMonth(),
now.getDate());

  let diff = now - today; // الفرق بالمليثانية
  return Math.round(diff / 1000); // نحوّله إلى ثوان
}

alert( getSecondsToday() );
```

الحل الآخر هو جلب الساعة والدقيقة والثانية وتحويلها إلى عدد الثواني:

```
function getSecondsToday() {
  let d = new Date();
  return d.getHours() * 3600 + d.getMinutes() * 60 + d.getSeconds();
}
```

5.11.11 كم من ثانية بقت حتّى الغد؟

الأهمية: ★★★★★

أنشِئ دالة `getSecondsToTomorrow()` تُعيد عدد الثواني حتّى يحلّ الغد. فمثلًا لو كان الوقت الآن `23:00`، تُعيد لنا:

```
getSecondsToTomorrow() == 3600
```

> يجب أن تعمل الدالة مهما كان اليوم، وألا تعتبر "اليوم" هذا اليوم.

الحل:

لنعرف عدد المليثوان حتّى قدوم الغد، يمكننا أن نطرح من "الغد 00:00:00" التـاريخ اليـوم. أوّلًا، نولّـد هـذا "الغد" وثمّ ننفّذ الطرح:

```
function getSecondsToTomorrow() {
  let now = new Date();

  // تاريخ الغد
  let tomorrow = new Date(now.getFullYear(), now.getMonth(),
now.getDate()+1);

  let diff = tomorrow - now; // الفرق بالمليثانية
  return Math.round(diff / 1000); // نحوّله إلى ثوان
}
```

حل بديل:

```
function getSecondsToTomorrow() {
  let now = new Date();
  let hour = now.getHours();
  let minutes = now.getMinutes();
  let seconds = now.getSeconds();
  let totalSecondsToday = (hour * 60 + minutes) * 60 + seconds;
  let totalSecondsInADay = 86400;

  return totalSecondsInADay - totalSecondsToday;
}
```

لاحظ أنّ هناك دولًا كثيرة تستعمل التوقيت الصيفي، لذا ستجد هناك أيام فيهـا 23 أو 25 سـاعة. يمكن أن نتعامل مع هذه الأيام بنحوٍ منفصل.

ا. تنسيق التاريخ نسبيًا

الأهمية: ★★★★☆

اكتب دالة `formatDate(date)` تُنسّق التاريخ `date` حسب الآتي:

- لو مرّت أقلّ من ثانية من `date`، فتُعيد `"right now"`.
- وإلّا، لو مرّت أقلّ من دقيقة من `date`، فتُعيد `"n sec. ago"`.

- وإلّا، لو أقل من ساعة، فتُعيد `"m min. ago"`.
- وإلّا، فتُعيد التاريخ كاملًا بالتنسيق `"DD.MM.YY HH:mm"`، أي (شَكلًا): الدقيقة:الساعة العام:الشهر:اليوم (كلها بخانتين). مثل: `31.12.16 10:00`.

أمثلة:

```
alert( formatDate(new Date(new Date - 1)) ); // "right now"

alert( formatDate(new Date(new Date - 30 * 1000)) ); // "30 sec. ago"

alert( formatDate(new Date(new Date - 5 * 60 * 1000)) ); // "5 min. ago"

// 31.12.16, 20:00 تاريخ الأمس، مثلًا
alert( formatDate(new Date(new Date - 86400 * 1000)) );
```

(تجربة حية للتمرين)

الحل:

لنجلب الوقت المنقضي منذ `date` وحتّى الآن، سنطرح التاريخين.

```
function formatDate(date) {
  let diff = new Date() - date; // الفرق بالمليثانية

  if (diff < 1000) { // أقل من ثانية واحدة
    return 'right now';
  }

  let sec = Math.floor(diff / 1000); // نحوّل الفرق إلى ثوان
  if (sec < 60) {
    return sec + ' sec. ago';
  }

  let min = Math.floor(diff / 60000); // نحوّل الفرق إلى دقائق
  if (min < 60) {
    return min + ' min. ago';
  }
```

```
  // نسّق التاريخ
  // ونُضيف أصفارًا لو كان اليوم/الشهر/الساعة/الدقيقة بخانة واحدة
  let d = date;
  d = [
    '0' + d.getDate(),
    '0' + (d.getMonth() + 1),
    '' + d.getFullYear(),
    '0' + d.getHours(),
    '0' + d.getMinutes()
  ].map(component => component.slice(-2)); //نأخذ الخانتين الأخيرتين من كلّ مكوّن

  // ندمج المكوّنات في تاريخ
  return d.slice(0, 3).join('.') + ' ' + d.slice(3).join(':');
}

alert( formatDate(new Date(new Date - 1)) ); // "right now"
alert( formatDate(new Date(new Date - 30 * 1000)) ); // "30 sec. ago"
alert( formatDate(new Date(new Date - 5 * 60 * 1000)) ); // "5 min.
ago"

// 31.12.16, 20:00 تاريخ الأمس، مثلًا
alert( formatDate(new Date(new Date - 86400 * 1000)) );
```

حل بديل:

```
function formatDate(date) {
  let dayOfMonth = date.getDate();
  let month = date.getMonth() + 1;
  let year = date.getFullYear();
  let hour = date.getHours();
  let minutes = date.getMinutes();
  let diffMs = new Date() - date;
  let diffSec = Math.round(diffMs / 1000);
  let diffMin = diffSec / 60;
  let diffHour = diffMin / 60;

  // التنسيق
```

```
    year = year.toString().slice(-2);
    month = month < 10 ? '0' + month : month;
    dayOfMonth = dayOfMonth < 10 ? '0' + dayOfMonth : dayOfMonth;

    if (diffSec < 1) {
      return 'right now';
    } else if (diffMin < 1) {
      return `${diffSec} sec. ago`
    } else if (diffHour < 1) {
      return `${diffMin} min. ago`
    } else {
      return `${dayOfMonth}.${month}.${year} ${hour}:${minutes}`
    }
  }
```

لاحظ بأنّ هذه الطريقة سيّئة لو أردت دعم اللغات دعمًا صحيحًا (في العربية هناك ثانية واحدة وثانيتين وثلاث ثوان وخمسون ثانية وهكذا).

(تجربة حية للحل)

5.12 صيغة JSON وتوابعها

لنقل بأنّ لدينا كائن معقّد البنية ونريد تحويله إلى سلسلة نصية كي نُرسله عبر الشبكة أو نطبعه في الطرفية لتسجيل المخرجات. الطبيعي هو أن تكون في هذه السلسلة النصية كلّ الخاصيات المهمة. يمكننا إجراء هذا التحويل بهذه الطريقة:

```
let user = {
  name: "Ahmad",
  age: 30,

  toString() {
    return `{name: "${this.name}", age: ${this.age}}`;
  }
};

alert(user); // {name: "Ahmad", age: 30}
```

ولكن... أثناء التطوير، نُضيف خاصيات جديدة ونُغيّر أسماء القديمة أو نحذفها حتّى. تحديث ذلك، مثل التابع `toString`، كلّ مرّة سيكون جحيمًا حقيقيًا. يمكن أن نمرّ على الخاصيات في الكائن، ولكن ماذا لو كان معقّدًا وفيه كائنات وخاصيات متداخلة؟ حينها سنحتاج إجراء تحويل لتلك أيضًا.

لحسن حظّنا فكتابة تلك الشيفرة لهذه المعضلة ليس له داعٍ، فهناك من حلّها بالفعل.

5.12.1 JSON.stringify

نسق JSON (اختصار إلى JavaScript Object Notation، أي صيغة كائنات جافاسكربت) هو نسق عام لتمثيل القيم والكائنات، ويوثّقه المعيار RFC 4627. في بادئ الأمر كان غرض كتابته هو لاستعماله في JavaScript، ولكن رويدًا رويدًا بدأت اللغات الأخرى صناعة مكتبات تتعامل معه أيضًا. لهذا يسهل لنا استعمال JSON لتبادل البيانات حين يستعمل جهاز العميل JavaScript بينما الخادم مكتوب بلغة روبي/PHP/جافا/أي لغة خنفشارية أخرى.

تقدّم JavaScript التوابع الآتية:

- `JSON.stringify` لتحويل الكائنات إلى صياغة JSON.
- `JSON.parse` لإرجاع بيانات مصاغة بصياغة JSON إلى كائن كما كان.

فمثلًا هنا نستعمل `JSON.stringify` على طالب `student`:

```
let student = {
  name: 'Ahmad',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js'],
  wife: null
};

let json = JSON.stringify(student);

alert(typeof json); // حصلنا على سلسلة نصية

alert(json);

/* كائن مرمّز بِـJSON:
{
  "name": "Ahmad",
  "age": 30,
  "isAdmin": false,
  "courses": ["html", "css", "js"],
  "wife": null
}
*/
```

يأخذ التابع `JSON.stringify(student)` الكائن ويحوّله إلى سلسلة نصية. تُسمّى سلسلة `json` النصية الناتج بكائن مرمّز بِـ(JSON-encoded) JSON أو مُسلسل (serialized) أو stringified أو marshalled. صرنا مستعدّين لإرسال الكائن عبر الوِب أو تخزينه في مخزن بيانات خام.

لاحظ من فضلك الاختلافات المهمة بين الكائن المرمّز بِـJSON من الكائن العادي الحرفي:

- تستعمل السلاسل النصية علامات اقتباس مزدوجة. لا مكان لعلامات الاقتباس المفردة أو الفواصل ` في JSON. بهذا يصير `'Ahmad'` هكذا `"Ahmad"`.
- حتّى خاصيات الكائنات تُحاط بعلامات اقتباس مزدوجة، ولا مناص من ذلك. بهذا يصير `age:30` هكذا `"age":30`.

يمكن استعمال `JSON.stringify` على الأنواع الأولية أيضًا.

تدعم JSON أنواع البيانات الآتية:

- الكائنات `{ ... }`
- المصفوفات `[ ... ]`
- الأنواع الأولية:
  - السلاسل النصية,
  - الأعداد,
  - القيم المنطقية `true/false`,
  - قيمة اللاشيء `null`.

مثال:

```
// العدد في JSON ليس إلّا عددًا
alert( JSON.stringify(1) ) // 1

// السلسلة النصية في JSON ليست إلّا سلسلة نصيّة، بين علامات اقتباس مزودجة
alert( JSON.stringify('test') ) // "test"

alert( JSON.stringify(true) ); // true

alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]
```

مواصفة JSON هي مواصفة مستقلّة لغويًا وتحمل البيانات فقط. لذا يُهمِل `JSON.stringify` خاصيات الكائنات الخاصّة JavaScript.

نذكر منها:

- خاصيات الدوال (التوابع).
- الخاصيات الرمزية.
- الخاصيات التي تُخزّن `undefined`.

```
let user = {
  sayHi() { // ignored
    alert("Hello");
  },
  [Symbol("id")]: 123, // يتجاهلها
```

```
    something: undefined // يتجاهلها
};
alert( JSON.stringify(user) ); // (كائن فارغ) {}
```

غالبًا، لا مانع من ذلك. لو كان هناك مانع فسنرى قريبًا طريقة تخصيص عملية السَلسلة هذه.

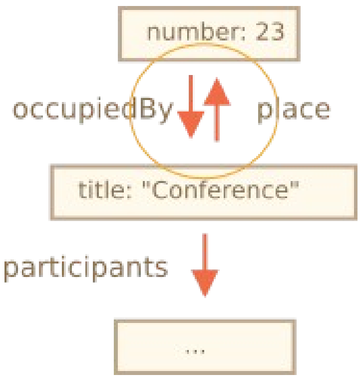
بفضل دهاء المبرمجين، فالكائنات المتداخلة مدعومة وستُحوَّل تلقائيًا. مثال:

```
let meetup = {
  title: "Conference",
  room: {
    number: 23,
    participants: ["john", "ann"]
  }
};
alert( JSON.stringify(meetup) );
/* البنية كاملة تتحوّل إلى سلسلة نصية:
{
  "title":"Conference",
  "room":{"number":23,"participants":["john","ann"]},
}
*/
```

إليك التقييد: وجود الإشارات التعاودية (circular references) ممنوع. مثال:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: ["john", "ann"]
};

meetup.place = room;       // (meetup -> room) يُشير الاجتماع إلى الغرفة
room.occupiedBy = meetup; // (room -> meetup) تُشير الغرفة إلى الاجتماع
JSON.stringify(meetup); // JSON خطأ تحاول تحويل بنية تعاوية إلى
```

هنــا فشــل التحويــل بســبب الإشــارات التعاوديــة: فتُشــير `room.occupiedBy` إلى `meetup` و `meetup.place` إلى `room`:



5.12.2 الاستثناءات وتعديل الكائنات: آلة الاستبدال

إليك الصياغة الكاملة للتابِع `JSON.stringify`:

```
let json = JSON.stringify(value[, replacer, space])
```

المعاملات:

- `value`: القيمة التي ستُرمَّز.
- `replacer`: مصفوفة من الخاصيات لترميزها، أو دالة ربط (mapping) بالشكل `function(key, value)`.
- `space`: عدد المسافات لاستعمالها لتنسيق السلسلة النصية.

في أغلب الوقت نستعمل `JSON.stringify` بتمرير المُعامل الأول فقط. ولكن لو أردنا تعديل عملية الاستبدال مثل تعديل الإشارات التعاودية، فيمكننا استعمال المُعامل الثاني للتابِع.

لو مرّرنا مصفوفة فيها خاصيات، فستُرمَّز تلك الخاصيات فقط. مثال:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "Ahmad"}, {name: "Alice"}],
  place: room // يُشير الاجتماع إلى الغرفة
};
```

```
room.occupiedBy = meetup; // room references meetup
alert( JSON.stringify(meetup, ['title', 'participants']) );
// {"title":"Conference","participants":[{},{}]}
```

ربّما نكون هنا صارمين كثيرًا، فقائمة الخاصيات تُطبّق على كامل بنية الكائن، بهذا الكائنات في participants فارغة إذ أنّ name ليست في القائمة.

لنضمّن في تلك القائمة كلّ خاصية عدا room.occupiedBy إذ ستتسبّب بإشارة تعاودية:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "Ahmad"}, {name: "Alice"}],
  place: room // يُشير الاجتماع إلى الغرفة
};

room.occupiedBy = meetup; // تُشير الغرفة إلى الاجتماع

alert( JSON.stringify(meetup, ['title', 'participants', 'place',
'name', 'number']) );
/*
{
  "title":"Conference",
  "participants":[{"name":"Ahmad"},{"name":"Alice"}],
  "place":{"number":23}
}
*/
```

الآن سَلسلنا كلّ ما في occupiedBy، ولكن قائمة الخاصيات صارت طويلة. لحسن حظّنا يمكننا استعمال دالة بدل المصفوفة لتكون آلة الاستبدال replacer. ستُستدعى الدالة لكلّ زوج (key, value) ويجب أن تُعيد القيمة "المُستبدَلة" التي ستحلّ مكان الأصلية، أو undefined لو أردنا إهمال الخاصية.

في حالتنا هذه ستُعيد القيمة value "كما هي" لكلّ الخاصيات باستثناء occupiedBy. لنُهمل occupiedBy لذا ستُعيد الشيفرة undefined:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "Ahmad"}, {name: "Alice"}],
  place: room // يُشير الاجتماع إلى الغرفة
};

room.occupiedBy = meetup; // تُشير الغرفة إلى الاجتماع

alert( JSON.stringify(meetup, function replacer(key, value) {
  alert(`${key}: ${value}`);
  return (key == 'occupiedBy') ? undefined : value;
}));

/* أزواج key:value التي تدخل آلة الاستبدال:
:             [object Object]
title:        Conference
participants: [object Object],[object Object]
[object Object]
name:         Ahmad
[object Object]
name:         Alice
place:        [object Object]
number:       23
*/
```

لاحِظ بأنّ الدالة `replacer` تأخذ كلّ زوج "مفتاح/قيمة" بما في ذلك الكائنات المتداخلة وعناصر المصفوفات، فهي تتطبّق تكراريًا. وقيمة `this` داخل `replacer` هي الكائن الذي يحتوي على الخاصية الحالية.

الاستدعاء الأوّل خاصّ قليلًا، فهو يستلم "كائن تغليف": `{"": meetup}`. بعبارة أخرى فأوّل زوج `(key, value)` يكون مفتاحه فارغًا وقيمته هي الكائن الهدف كلّه. لهذا نرى السطر الأول في المثال أعلاه `":[object Object]"`. الغرض هو تقديم كلّ ما أمكن من "تسلّط" لأداة الاستبدال، بهذا يمكنها تحليل الكائنات كاملةً واستبدالها أو إهمالها لو تطلّب الأمر.

5.12.3 التنسيق: المسافات

المُعامل الثالث للتابِع `JSON.stringify(value, replacer, space)` هو عدد المسافات الـتي ستُستعمل لتنسيقها تنسيقًا جميلًا (Pretty format).

في المثال السابق، لم يكن للكائنات المُسلسلة (stringified objects) أيّة مسافات أو مسافات بادئـة. لا بأس لو كنّا سنرسل الكائن عبر الشبكة، فالمُعامل `space` يُستعمل فقط لتجميل الناتج.

هنا بعبارة `space = 2` نوجه JavaScript بأن تعـرض الكائنـات المتداخلـة على عـدّة أسـطر، بمسـافتين بادئتين داخل كل كائن:

```
let user = {
  name: "Ahmad",
  age: 25,
  roles: {
    isAdmin: false,
    isEditor: true
  }
};

alert(JSON.stringify(user, null, 2));
/* إزاحة بمسافتين: 
{
  "name": "Ahmad",
  "age": 25,
  "roles": {
    "isAdmin": false,
    "isEditor": true
  }
}
*/

/* بينما JSON.stringify(user, null, 4) يعطينا إزاحة أكبر: 
{
    "name": "Ahmad",
    "age": 25,
    "roles": {
```

```
            "isAdmin": false,
            "isEditor": true
        }
    }
*/
```

نستعمل المُعامل space فقط لغرض الناتج الجميل وعمليات تسجيل المخرجات.

5.12.4 تابِع "toJSON" مخصّص

كما يوجد toString للتحويل إلى سلاسل نصية، يمكن للكائنات أيضًا تقديم تـابِع toJSON للتحويـل إلى JSON. تستدعي JSON.stringify ذاك التابع تلقائيًا لو وجدته. مثال:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  date: new Date(Date.UTC(2017, 0, 1)),
  room
};

alert( JSON.stringify(meetup) );
/*
  {
    "title":"Conference",
    "date":"2017-01-01T00:00:00.000Z",  // (1)
    "room": {"number":23}               // (2)
  }
*/
```

نرى هنا بأنّ date (في (1)) صار سلسلة نصية. هذا لأنّ التواريخ كلّها توفّر تنفيذًا للتابِع toJSON مضـمّنًا فيها، وهو يُعيد سلاسل نصية بهذا التنسيق.

لنُضيف الآن تابِع toJSON مخصّص للكائن room (في (2)):

```
let room = {
  number: 23,
  toJSON() {
    return this.number;
  }
};

let meetup = {
  title: "Conference",
  room
};

alert( JSON.stringify(room) ); // 23

alert( JSON.stringify(meetup) );
/*
  {
    "title":"Conference",
    "room": 23
  }
*/
```

كما نرى، استُعمِل التابع `toJSON` مرتين، مرة حين اسـتدعاه `JSON.stringify(room)` مباشـرةً، ومـرة حين كانت الخاصية `room` داخل كائن مرمّز آخر.

5.12.5 التابع JSON.parse

لنفكّ ترميز سلسلة JSON نصية، سنحتاج تابِعًا آخر بالاسم JSON.parse. صياغته هي:

```
let value = JSON.parse(str, [reviver]);
```

المعاملات:

- `str`: سلسلة JSON النصية التي سيُحلّلها.
- `reviver`: دالة اختيارية `function(key,value)` تُستدعَى لكلّ زوج `(key, value)` ويمكن لها تعديل القيمة.

مثال:

```
// (stringified array) مصفوفة مُسلسلة
let numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1
```

أو حين استعمالها للكائنات المتداخلة:

```
let userData = '{ "name": "Ahmad", "age": 35, "isAdmin": false,
"friends": [0,1,2,3] }';

let user = JSON.parse(userData);

alert( user.friends[1] ); // 1
```

يمكن أن يكون كائن JSON بالتعقيد اللازم مهمًا كان. يمكن أن تحتوي الكائنات والمصفوفات كائناتَ ومصفوفات أخرى، ولكنّ لزامٌ عليها أن تلتزم بنفس نسق JSON.

إليك بعض المشاكل الشائعة حين كتابة JSON يدويًا (أحيانًا نفعل ذلك لأغراض تنقيح الشيفرات):

```
let json = `{
  name: "Ahmad",                     // خطأ: اسم خاصية بدون علامات اقتباس
  "surname": 'Smith',               // خطأ: علامات اقتباس مُفردة في القيمة (يجب
أن تكون مزدوجة)
  'isAdmin': false                  // خطأ: علامات اقتباس مُفردة في المفتاح (يجب
أن تكون مزدوجة)
  "birthday": new Date(2000, 2, 3), // خطأ: استعمال "new" ممنوع، فقط وفقط قيم
  "friends": [0,1,2,3]              // هنا لا بأس
}`;
```

وأجل، لا تدعم JSON التعليقات، فلو أضفتها سيتحوّل الكائن إلى كائن غير صالح.

هناك نسق آخر بالاسم JSON5 ويُتيح لنا عدم إحاطة المفاتيح بعلامات اقتباس، وكتابة التعليقات وغيرها. إلّا أنّها مكتبة مستقلة وليست في مواصفة لغة JavaScript. لم يصنع المطوّرون كائنات JSON العادية لتكون بهذه الصرامة لأنّهم كسالى، بل لتُعوّل على شيفرات خوارزميات التحليل، إضافة إلى عملها بسرعة فائقة.

5.12.6 استعمال آلة الإحياء

تخيّل أنّنا استلمنا كائن `meetup` مُسَلسَل من الخادم، وهذا شكله:

```
// title: (meetup title), date: (meetup date)
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';
```

...نريد الآن "فكّ ترميزه"، أي إعادته إلى كائن JavaScript عادي. يكون ذلك باستدعاء `JSON.parse`:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert( meetup.date.getDate() ); // خطأ!
```

لحظة... خطأ؟!

قيمة الخاصية `meetup.date` هي سلسلة نصية وليست كائن تاريخ `Date`. كيف سيعرف `JSON.parse` بأنّ عليه تعديل تلك السلسلة النصية لتصير `Date`؟

لنمرّر الآن إلى `JSON.parse` دالة "آلة الإحياء" في المُعامل الثاني، وستُحيي كلّ القيم "كما هي"، عدا `date` ستعدّلها لتكون تاريخًا:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( meetup.date.getDate() ); // الآن صار يعمل!
```

وأجل، تعمل الشيفرة للكائنات المتداخلة أيضًا:

```
let schedule = `{
  "meetups": [
    {"title":"Conference","date":"2017-11-30T12:00:00.000Z"},
    {"title":"Birthday","date":"2017-04-18T12:00:00.000Z"}
  ]
}`;
```

```
schedule = JSON.parse(schedule, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( schedule.meetups[1].date.getDate() ); // يعمل!
```

5.12.7 الخلاصة

- تنسيق JSON هو تنسيق بيانات يستقلّ بمعياره ومكتباته في غالبية لغات البرمجة.
- يدعم JSON الكائنات العادية والمصفوفات والسلاسل النصية والأعداد والقيم المنطقية و`null`.
- تقدّم JavaScript التابعان JSON.stringify لسَلسلة الكائنات إلى JSON، وJSON.parse للقراءة من JSON.
- يدعم كلا التابعين دوال تعديل لتكون القراءة والكتابة "ذكيّة".
- لو كان في الكائن تابِع `toJSON`، فسيستدعيه `JSON.stringify`.

5.12.8 تمارين

1. تحويل كائن إلى JSON وإعادته كما كان

الأهمية: ★★★★★

حوّل الكائن `user` إلى JSON واقرأه ثانيةً ليكون متغيرًا آخرًا.

```
let user = {
  name: "Ahmad Smith",
  age: 35
};
```

الحل:

```
let user = {
  name: "Ahmad Smith",
  age: 35
};

let user2 = JSON.parse(JSON.stringify(user));
```

ب. استثناء الإشارات السابقة

الأهمية: ★★★★★

يمكننا في الحالات العادية من الإشارات التعاودية استثناء خاصية محدّدة لألا تُسلسل، حسب اسمها. ولكن أحيانًا لا نستطيع استعمال الاسم إذ يُستعمل في الإشارات التعاودية زائدًا الخاصيات العادية. يمكننا هنا فحص الخاصية حسب قيمتها.

اكتب دالة `replacer` تُسلسل كل شيء ولكن تُزيل الخاصيات التي تُشير إلى `meetup`:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  occupiedBy: [{name: "Ahmad"}, {name: "Alice"}],
  place: room
};

// إشارات تعاودية
room.occupiedBy = meetup;
meetup.self = meetup;

alert( JSON.stringify(meetup, function replacer(key, value) {
  /* شيفرتك هنا */
}));

/* هكذا النتيجة المطلوبة:
{
```

```
    "title":"Conference",
    "occupiedBy":[{"name":"Ahmad"},{"name":"Alice"}],
    "place":{"number":23}
  }
*/
```

الحل:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  occupiedBy: [{name: "Ahmad"}, {name: "Alice"}],
  place: room
};

room.occupiedBy = meetup;
meetup.self = meetup;

alert( JSON.stringify(meetup, function replacer(key, value) {
  return (key != "" && value == meetup) ? undefined : value;
}));

/*
{
  "title":"Conference",
  "occupiedBy":[{"name":"Ahmad"},{"name":"Alice"}],
  "place":{"number":23}
}
*/
```

علينا هنا (أيضًا) فحص `key==""` لنستثني أوّل نداء إذ لا مشكلة بأن تكون القيمة `value` هي `meetup`.

دورة تطوير التطبيقات
باستخدام لغة بايثون

احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن

6. التعامل المتقدم مع الدوال

يتضمن هذا الفصل الأقسام التالية:

1. التعاود Rescursion والمكدس Stack
2. المعاملات "البقية" ومعامل التوزيع
3. المنغلقات Closure ومجال المتغيرات
4. إفادة "var" القديمة
5. الكائن العمومي Global object
6. كائنات الدوال وتعابير الدوال المسمّاة NFE
7. صياغة "الدالة الجديدة" new Function
8. المُهلة setTimeout والفترة setInterval
9. المزخرفات والتمرير: التابعان call وapply
10. ربط الدوال Function binding
11. الحديث عن الدوال السهمية مرة أخرى

6.1 التعاود Rescursion والمكدس Stack

فلنعد الآن إلى الدوال functions ونرى أمرها بتمعّن وتعمّق أكثر. سنتكلم أولًا عن التعاود (Rescursion).

لو كنت ذا علم بالبرمجة فالأغلب أنّك تعرف ما هذا التعاود ويمكنك تخطّي هذا الفصل. يُعدّ التعاود (Rescursion) نمطًا برمجيًا نستعمله حين يمكن تقسيم المهمة الكبيرة جدًا إلى مهام أبسط منها متشابهة، أو حين يمكن تبسيط المهمة الواحدة إلى عملية بعضها بسيط وآخر يتشابه بين بعضه، أو نستعمله (كما سنرى قريبًا) للتعامل مع أنواعٍ محدّدة من بنى البيانات.

يمكن للدالة حين تحاول إجراء مهمّة ما نداءَ دوال أخرى. أحيانًا يمكن أن تنادي تلك الدالة "نفسها" ثانيةً. وهذا ما ندعوه بالتعاود أي أن تعاود الدالة استدعاء نفسها.

6.1.1 نهجان في التطوير

لنبدأ بما هو أبسط. لنكتب دالة `pow(x, n)` ترفع `x` إلى الأسّ الطبيعي `n`. بعبارة أخرى، تضرب `x` بنفسه `n` مرّة.

```
pow(2, 2) = 4
pow(2, 3) = 8
pow(2, 4) = 16
```

يمكننا تنفيذ هذه الدالة بطريقتين اثنتين.

- التفكير بالتكرار: حلقة `for`:

```
function pow(x, n) {
  let result = 1;

  // نضرب الناتج في x - n مرّة داخل الحلقة
  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

alert( pow(2, 3) ); // 8
```

- التفكير بالتعاود: تبسيط المهمة ونداء "الذات":

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}

alert( pow(2, 3) ); // 8
```

لاحظ كيف أنّ تلك الشيفرة التعاودية مختلفة جذريًا عن سابقتها.

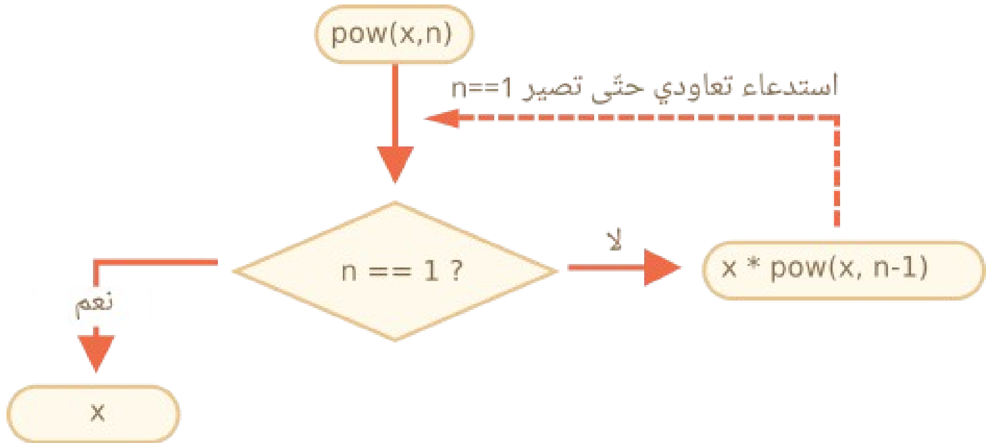
حين تُستدعَى `pow(x, n)` تنقسم عملية التنفيذ إلى فرعين:

```
              if n==1  = x
             /
pow(x, n) =
             \
              else     = x * pow(x, n - 1)
```

1. حين `n == 1`، نرى كل شيء كالعادة. نسمّي تلك الحالة "بأساس التعاود" (base of recursion)، لأنها تُعطينا الناتج البديهي مباشرة: `pow(x, 1)` تساوي `x`.
2. عدى تلك فيمكننا تمثيل `pow(x, n)` على أنها `x * pow(x, n - 1)`.

يمكنك رياضيًا كتابة xn = x * xn-1. نسمّي هذه "خطوة تعاودية" (recursive step)، فنعدّل مهمة الأس الكبيرة لتصير عملية أبسط (الضرب في `x`) ونستعمل استدعاءً أبسط لمهمة الأس (`pow` ولكن `n` أقل). في الخطوات اللاحقة تصير أبسط وأبسط إلى أن تصل `n` إلى `1`.

يمكن أيضًا أن نقول بأن `pow` تستدعي نفسها تعاوديًا حتى تكون `n == 1`.



فمثلًا كي نحسب قيمة `pow(2, 4)`، على التعاود إجراء هذه المهام:

1. `pow(2, 4) = 2 * pow(2, 3)`
2. `pow(2, 3) = 2 * pow(2, 2)`
3. `pow(2, 2) = 2 * pow(2, 1)`
4. `pow(2, 1) = 2`

للتلخيص، يبسّط التعاود استدعاء الدالة إلى استدعاءً آخر أبسط، وبعدها أبسط، وأبسط، وأبسط، حتّى يظهر الناتج ويصير معلومًا.

غالبًا ما تكون شيفرة التعاود أقصر

عادةً ما يكون الحل باستعمال التعاود أقصر من التكرار بالحلقات.

يمكننا هنا مثلًا إعادة كتابة نفس الشيفرة ولكن باستعمال المُعامل الشرطي `?` بدل `if` لتصير `pow(x, n)` أقصر أكثر وتبقى مقروءةً لنا:

```
function pow(x, n) {
  return (n == 1) ? x : (x * pow(x, n - 1));
}
```

يُسمّى أقصى عدد من الاستدعاءات المتداخلة (بما في ذلك أول استدعاء) "بعمق التعاود" (Rescursion Depth). في حالتنا هنا سيكون هذا العمق `n`.

يَحدُّ محرّك JavaScript من أقصى عمق تعاودي ممكن. يمكن أن نقول بأنّ 10000 هو الحدّ الذي يمكننا الاعتماد عليه (ولو أنّ بعض المحرّكات ترفع هذا الحدّ أكثر). أجل، هناك تحسينات تلقائية تحاول رفع هذا الحدّ ("تحسينات نهاية الاستدعاء") ولكنّها ليست مدعومة في كلّ مكان ولا تعمل إلّا على الحالات البسيطة.

يقصّر هذا من تطبيقات استعمال التعاود، ولكنّه مع ذلك مستعمل بشدة، إذ هناك مهام كثيرة لو استعملت عليها التعاود لأعطتك شيفرة أقصر وأسهل للصيانة.

6.1.2 سياق التنفيذ والمكدس

لنرى الآن كيف يعمل التعاود أصلًا، ولذلك لا بدّ من أن نرى ما خلف كواليس الدوال هذه.

تُخزَّن المعلومات حول عملية تنفيذ الدالة (حين تعمل) في "سياقها التنفيذي" (execution context).

يُعدّ سياق التنفيذ بنيةَ بيانات داخلية تحوي التفاصيل التي تخصّ عملية تنفيذ الدالة: إلى أين وصلت الآن؟ ما المتغيرات الحالية؟ ما قيمة `this` (لا نستعملها هنا) وتفاصيل أخرى داخلية. لكلّ استدعاء دالة سياق تنفيذي واحد مرتبط بها.

حين تستدعي الدالة دوال أخرى متداخلة، يحدث:

- تتوقف الدالة الحالية مؤقتًا.
- يُحفَظ سياق التنفيذ المرتبط بها في بنية بيانات خاصّة تسمى "مكدس سياق التنفيذ" (execution context stack).
- يُنفَّذ الاستدعاء المتداخل.
- ما إن ينتهي، يجلب المحرّك التنفيذ القديم ذاك من المكدس، وتواصل الدالة الخارجية عملها حيث توقفت.

لنرى ما يحدث أثناء استدعاء `pow(2, 3)`.

ا. pow(2, 3)

يخزّن سياق التنفيذ (في بداية استدعاء `pow(2, 3)`) المتغيرات هذه: `x = 2, n = 3` وأنّ سير التنفيذ هو في السطر رقم `1` من الدالة.

يمكن أن تصوره هكذا:

```
Context: { x: 2, n: 3, at line 1 } | call: pow(2, 3)
```

هذا ما يجري حين يبدأ تنفيذ الدالة. بعد أن يصير الشرط `n == 1` خطأً، ينتقل سير التنفيذ إلى الفرع الثاني من الإفادة `if`:

```
function pow(x, n) {            // 1
  if (n == 1) {                 // 2
    return x;                   // 3
  } else {                      // 4
    return x * pow(x, n - 1);   // 5
  }                             // 6
```

```
}                          // 7

alert( pow(2, 3) );        // 9
```

ما زالت المتغيرات كما هي، ولكن السطر تغيّر. بذلك يصير السياق الآن:

```
Context: { x: 2, n: 3, at line 5 } | call: pow(2, 3)
```

علينا لحساب `x * pow(x, n - 1)` استدعاء `pow` فرعيًا بالوُسطاء الجديدة `pow(2, 2)`.

ب. pow(2, 2)

كي يحدث الاستدعاء المتداخل، يتذكّر محرّك JavaScript سياق التنفيذ الحالي داخل "مكدس سياق التنفيذ".

هنا نستدعي ذات الدالة `pow`، ولكن ذلك لا يهم إذ أنّ العملية هي ذاتها لكلّ الدوال:

1. "يتذكّر المحرّك" السياقَ الحالي أعلى المكدس.
2. يَصنع سياقًا جديدًا للاستدعاء الفرعي.
3. متى انتهى الاستدعاء الفرعي يُطرح (pop) السياق السابق من المكدس ويتواصل التنفيذ.

هذا مكدس السياق حين ندخل الاستدعاء الفرعي `pow(2, 2)`:

```
Context: { x: 2, n: 2, at line 1 } | call: pow(2, 2)
Context: { x: 2, n: 3, at line 5 } | call: pow(2, 3)
```

سياق التنفيذ الحالي والجديد هو الأعلى (بالخط الثخين) وأسفله السياقات التي تذكّرها المحرّك سابقًا.

حين ننتهي من الاستدعاء الفرعي يمكننا بسهولة بالغة مواصلة السياق السابق، إذ أنّ متغيراته ومكان توقف الشيفرة محفوظان بالضبط في السياق.

> نرى أنّا استعملنا كلمة "سطر" (line) إذ ليس في المثال إلّا استدعاءً فرعيًا واحدًا في السطر، ولكن يمكن أن تحتوي الشيفرات ذات السطر الواحد (بصفة عامة) على أكثر من استدعاء فرعي، هكذا: `pow(…) + pow(…) + somethingElse(…)`.
>
> لذا سنكون أدقّ لو قلنا بأن عملية التنفيذ تتواصل "بعد الاستدعاء الفرعي مباشرةً".

ج. pow(2, 1)

تتكرّر العملية: يُصنع استدعاء فرعي جديد في السطر `5` بالوسطاء الجديدة `x=2` و `n=1`. صنعنا سياقًا جديدًا، إذًا ندفع (push) الأخير أعلى المكدس:

```
Context: { x: 2, n: 1, at line 1 } | call: pow(2, 1)
Context: { x: 2, n: 2, at line 1 } | call: pow(2, 2)
Context: { x: 2, n: 3, at line 5 } | call: pow(2, 3)
```

الآن هناك سياقين اثنين قديمين، وواحد يعمل للاستدعاء `pow(2, 1)`.

د. المخرج

نرى الشرط `n == 1` صحيحًا أثناء تنفيذ الاستدعاء `pow(2, 1)` (عكس ما سبقه من مرّات)، إذًا فالفرع الأول من إفادة `if` سيعمل هنا:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}
```

لا استدعاءات متداخلة من هنا، بذلك تنتهي الدالة وتُعيد `2`. وحين تنتهي الدالة لا يكون هناك حاجة لسياق التنفيذ فيُزَال من الذاكرة، وبعدها يرجع السياق السابق أعلى المكدس:

```
Context: { x: 2, n: 2, at line 5 } | call: pow(2, 2)
Context: { x: 2, n: 3, at line 5 } | call: pow(2, 3)
```

يتواصل تنفيذ الاستدعاء `pow(2, 2)`، وفيه ناتج الاستدعاء الفرعي `pow(2, 1)` لذا يُنهي أيضًا تنفيذ `x * pow(x, n - 1)` فيُعيد 4.

بعدها يستعيد المحرّك السياقَ السابق:

```
Context: { x: 2, n: 3, at line 5 } | call: pow(2, 3)
```

وحين ينتهي، يكون عندنا ناتج `pow(2, 3) = 8`.

عمق التعاود في هذه الحالة هو: 3.

كما نرى في الصور أعلاه فعمق التعاود يساوي أقصى عدد من السياقات في المكدس.

لكن اعلم بأنّ السياقات تطلب مساحة من الذاكرة. في حالتنا هنا نرفع العدد للأسّ `n`، وبذلك نحتاج ما يكفي من ذاكرة تسع لتخزين `n` سياق لكل القيم الأصغر من `n`.

خوارزميات الحلقات والتكرار أفضل من حيث حجز الذاكرة:

```
function pow(x, n) {
  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

تستعمل دالة pow المتكرّرة سياقًا واحدًا تغيّر فيه المتغيران i و result أثناء عملها، كما وأنّ احتياجاتها للذاكرة قليلة وثابتة ولا تعتمد على قيمة n.

يمكن كتابة التعاودات أيًا كانت بصيغة الحلقات، وغالبًا ما تكون تلك الحلقات أفضل أداءً.

...ولكن أحيانًا ما تكون إعادة الكتابة تلك تافهة وبلا قيمة حقيقية خصوصًا حين تستعمل الدالة استدعاءاتٍ دوال تعاودية تختلف حسب شروط معيّنة، أو أن تدمج الدالة نتائج الاستدعاءات أو حين يصير تفريع الدالة أصعب أكثر فأكثر. حينها سيكون ذلك التحسين من المحرّك بلا داعٍ ولا حاجة لبذل كل ذلك المجهود له.

هكذا يعطينا التعاود شيفرة أقصر سهلة الفهم ويمكننا دعمها بلا عناء. لا نحتاج إلى هذا "التحسين" في كلّ مكان؛ ما نريد هو فقط شيفرة جيدة، ولهذا نستعمل التعاود.

6.1.3 مسح الأشجار تعاوديًا

مسح الأشجار تعاوديًا (Recursive Traversal) هو تطبيق آخر عن روعة التعاود. لنقل بأنّ لدينا شركة ويمكن أن نمثّل بنية موظّفيها في هذا الكائن:

```
let company = {
  sales: [{
    name: 'Ahmad',
    salary: 1000
  }, {
    name: 'Alice',
    salary: 600
  }],

  development: {
```

```
    sites: [{
      name: 'Peter',
      salary: 2000
    }, {
      name: 'Alex',
      salary: 1800
    }],

    internals: [{
      name: 'Jack',
      salary: 1300
    }]
  }
};
```

أي أنّ في الشركة أقسام عدّة.

- يمكن أن يحتوي كل قسم على مصفوفة من العاملين. فمثلًا لقسم المبيعات `sales` عاملين اثنين: Ahmad و Alice.
- أو أن ينقسم القسم إلى أقسام فرعية، مثل قسم التطوير `development` له فرعان: تطوير المواقع `sites` والبرمجيات الداخلية `internals`. ولكلّ من الفرعين موظفين منفصلين.
- يمكن أيضًا أن يكبُر القسم الفرعي ويصير فروع من القسم الفرعي (أي "فِرَق").

مثلًا قسم المبيعات `sites` سيتطوّر ويتحسّن وينقسم مستقبلًا إلى فرعين `siteA` و `siteB`. وبعدها ربما (لو عمل فريق التسويق بجدّ) ينقسم أكثر أيضًا. طبعًا هذا تخيّل فقط وليس في الصورة تلك.

الآن، ماذا لو أردنا دالة تعطينا مجموع كل الرواتب؟ كيف السبيل؟

لو جرّبنا بالتكرار فسيكون أمرًا عسيرًا إذ أنّ البنية ليست ببسيطة. أول فكرة على البال هي حلقة `for` تمرّ على الشركة `company` وداخلها حلقات فرعية على الأقسام بالمستوى الأول. ولكن هكذا سنحتاج حلقات فرعية متداخلة أيضًا لتمرّ على الموظفين في الأقسام بالمستوى الثاني مثل قسم `sites`... وبعدها حلقات أخرى داخل تلك فوقها للأقسام بالمستوى الثالث إن عمل فريق التسويق كما يجب... لو وضعنا 3-4 من هذه الحلقات الفرعية المتداخلة في شيفرة لتعمل جولة مسح على كائن واحد، فستنتج لنا شيفرة قبيحة حقًا. إذًا، لنجرّب التعاود الآن.

كما رأينا، حين تُلاقي الدالة قسمًا عليها جمع رواتبه، تواجه حالتين اثنتين:

1. إمّا يكون قسمًا بسيطًا فيه "مصفوفة" من الناس، وهكذا تجمع رواتبهم في حلقة بسيطة.
2. أو تجد "كائنًا" فيه N من الأقسام الفرعية، حينها تصنع N من الاستدعاءات المتعاودة لتحصي مجموع كلّ قسم فرعي وتدمج النتائج كلها.

الحالة الأولى هي أساس التعاود، أي عملنا العادي حين نستلم مصفوفة.

الحالة الثانية (حين نرى كائنًا) هي خطوة في التعاود. يمكن تقسيم تلك المهمة المعقّدة إلى مهام فرعية لكلّ قسم. ربّما تنقسم تلك المهام الفرعية ثانيةً، ولكنّها عاجلًا أم آجلًا ستنتهي بالحالة (1) لا محالة.

ربّما... يكون أسهل لو قرأت الخوارزمية من الشيفرة ذاتها:

```
// الكائن كما هو، ضغطناه لألا نُطيل القصة فقط
let company = {
  sales: [{name: 'Ahmad', salary: 1000}, {name: 'Alice', salary: 600
}],
  development: {
    sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800
}],
    internals: [{name: 'Jack', salary: 1300}]
  }
};

// الدالة التي ستنفّذ هذا العمل
function sumSalaries(department) {
  if (Array.isArray(department)) { // (1) حالة
    return department.reduce((prev, current) => prev + current.salary,
0); // نجمع عناصر المصفوفة
  } else { // (2) حالة
    let sum = 0;
    for (let subdep of Object.values(department)) {
      // نستدعي الأقسام الفرعية تعاوديًا، ونجمع النتائج
      sum += sumSalaries(subdep);
    }
    return sum;
  }
}

alert(sumSalaries(company)); // 6700
```

الشيفرة قصيرة وسهل فهمها (كما هو أملي). هنا تظهر قوّة التعاود، فسيعمل على أيّ مستوى من الأقسام الفرعية المتداخلة.

إليك رسمة توضّح الاستدعاءات:

```
{
  sales: [{
    name: 'John',
    salary: 1000
  }, {
    name: 'Alice',
    salary: 600
  }],

  development: {
    sites: [{
      name: 'Peter',
      salary: 2000
    }, {
      name: 'Alex',
      salary: 1800
    }],

    internals: [{
      name: 'Jack',
      salary: 1300
    }]
  }
};
```

الفكرة بسيطة للغاية: لو كان كائنًا `{...}`، نستعمل الاستدعاءات الفرعية، ولو كانت مصفوفات `[...]` هي آخر "أوراق" شجرة التعاود، فتعطينا الناتج مباشرةً.

لاحظ كيف أنّ الشيفرة تستعمل مزايا ذكيّة ناقشناها سابقًا:

- التابع `arr.reduce` في الفصل "توابِع المصفوفات" لنجمع الرواتب.
- الحلقة `for(val of Object.values(obj))` للمرور على قيم الكائن إذ يُعيد التابع `Object.values` مصفوفة بالقيم.

6.1.4 بنى التعاود

بنية البيانات التعاودية (أيّ التي يحدّد أساسها التعاود) هي بنية تكرّر نفسها على أجزاء منفصلة.

رأينا لتوّنا مثالًا عن هذه البنية: بنية الشركة أعلاه. "القسم" في الشركة هو إمّا:

- مصفوفة من الناس.

- أو كائنًا فيه "أقسام أخرى".

لو كنت مطوّر وِيب فالأمثلة التي تعرفها وتُدركها هي مستندات HTML و XML.

ففي مستندات HTML، يمكن أن يحتوي وسم HTML على قائمة من:

- أجزاء من نصوص.
- تعليقات HTML.
- وسوم HTML أخرى (أي ما يمكن أن يحتوي على أجزاء من نصوص أو تعليقات أو وسوم أخرى وهكذا).

وهذا ما نسمّيه بالبنى التعاوديّة.

لنفهم التعاود أكثر سنشرح بنية تعاود أخرى تسمّى "القوائم المترابطة" (Linked List). يمكن أن تكون هذه القوائم أحيانًا بديلًا أفضل موازنةً بالمصفوفات.

ا. القوائم المترابطة

لنقل بأنّا نريد تخزين قائمة كائنات مرتّبة.

ستصنع مصفوفة كالعادة:

```
let arr = [obj1, obj2, obj3];
```

ولكن... هناك مشكلة تخصّ المصفوفات. عمليات "حذف العنصر" و"إدراج العنصر" مُكلفة. فمثلًا على عملية `arr.unshift(obj)` إعادة ترقيم كلّ العناصر للكائن الجديد `obj`، ولو كانت المصفوفة كبيرة فستأخذ العملية وقتًا طويلًا. الأمر نفسه ينطبق لعملية `arr.shift()`.

التعديلات على بنية البيانات (التي لا تحتاج إلى إعادة الترقيم بالجملة) هي تلك التي تؤثّر على نهاية المصفوفة: `arr.push/pop`. لذا يمكن أن تكون المصفوفة بطيئة حقًا لو كانت الطوابير طويلة حين نعمل مع المصفوفات من عناصرها الأولى.

يمكننا عوض ذلك استعمال بنية بيانات أخرى لو أردنا إدخال البيانات وحذفها سريعًا. تُدعى هذه البنية بالقائمة المترابطة.

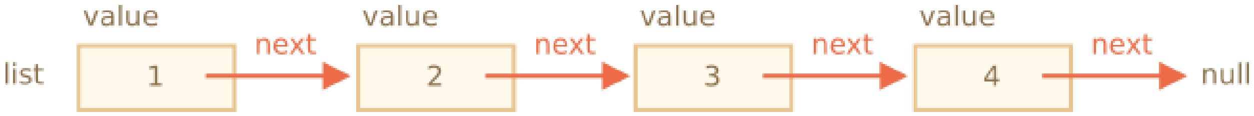
يُعرّف "عنصر القائمة المترابطة" تعاوديًا على أنّه كائن فيه:

- قيمة `value`.
- خاصية "التالي" `next` تُشير إلى "عنصر القائمة المترابطة" التالي أو إلى `null` لو كانت هذه نهاية القائمة.

مثال:

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};
```

إليك التمثيل البصري لهذه القائمة:



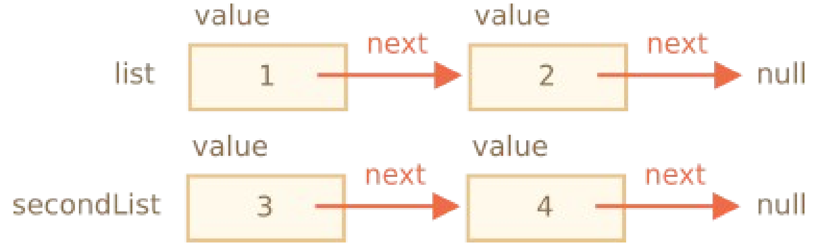
هذه شيفرة أخرى لنصنع القائمة:

```
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };
list.next.next.next.next = null;
```

هنا نرى بوضوح أكثر كيـف أنّ هنـاك كائنـات متعـدّدة لكـلّ منهـا خاصـية value وأخـرى next تُشـير إلى العنصر بقرب "هذا". متغيّر list هو أول الكائنات في السلسلة وبهذا لو اتّبعنا إشارات next بدءًا منها سنصل إلى أيّ عنصر آخر نريد.

يمكننا قسمة القائمة إلى أجزاء عدّة ودمجها لاحقًا لو أردنا:

```
let secondList = list.next.next;
list.next.next = null;
```



للدمج:

```
list.next.next = secondList;
```

وطبعًا يمكننا إدخال العناصر إلى أي مكان وإزالتها من أي مكان.

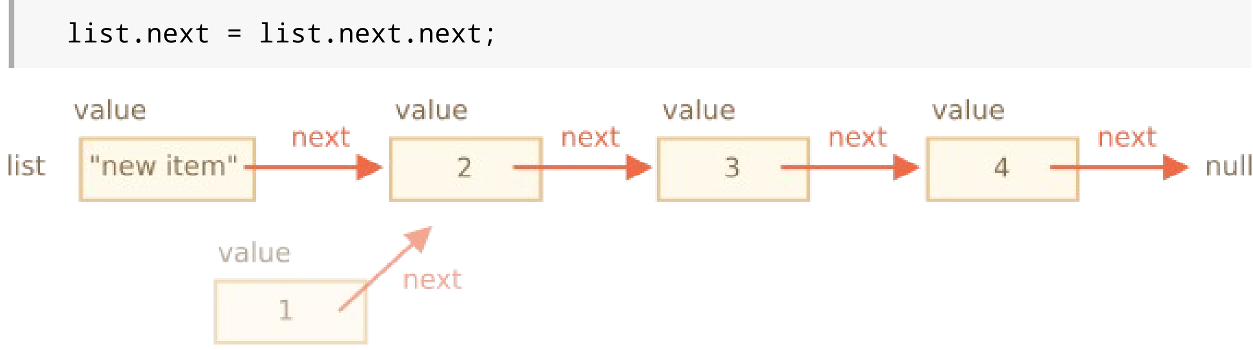
فمثلًا لو أردنا إضافة قيمة جديدة للبداية فعلينا تحديث رأس القائمة:

```
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };

// نُضيف قيمة جديدة إلى بداية القائمة
list = { value: "new item", next: list };
```

list "new item" → next → 1 → next → 2 → next → 3 → next → 4 → next → null

ولنُزيل قيمة من المنتصف نعدّل خاصية `next` للكائن الذي يسبق الجديد:

```
list.next = list.next.next;
```



هكذا أجبرنا `list.next` بأن "تقفز" فوق 1 لتصل 2، بهذا استثنينا القيمة 1 من السلسلة. وطالما أنّها ليست مخزّنة في أيّ مكان آخر فستُزال من الذاكرة تلقائيًا.

وعلى عكس المصفوفات فلسنا هنا نُعيد الترقيم بالجملة، أي أنّ إعادة ترتيب العناصر أسهل.

بالطبع فالقوائم ليست أفضل من المصفوفات دومًا وإلا فاستعملناها هي دومًـا ومـا احتجنـا المصـفوفات أبدًا. السلبية الأساس هي أنّ الوصول إلى العنصر حسب رقمه ليس سهلًا كما في المصـفوفات حيث نسـتعمل الإشارة المباشرة `arr[n]`. ولكن في القوائم علينا البدء من العنصر الأول والانتقال `N` مرّة عـبر `next` لنصـل إلى العنصر بالرقم N.

ولكننا لا نحتاج دومًا إلى هذه العمليـات فمثلًا حين نريـد طـابورًا (queue) أو حتّى طـابورًا متعـدّد الطـرفين (deque) فيجب أن نستعمل بنية مرتّبة تتيح بإضافة/إزالة العناصر من الجهتين بسرعة فائقة، وليس بالضروري أن نعرف ما في وسطها.

يمكننا تحسين القوائم هكذا:

- إضافة الخاصية `prev` مع الخاصية `next` للإشارة إلى العنصر السابق، لننتقل وراءً بسهولة أكبر.
- إضافة متغيّر بالاسم `tail` يُشيـر إلى آخـر عنصـر من القائمـة (وتحديثـه مـتى أضـفنا/أزلنـا عناصـر من النهاية).
- ...يمكن أن تتغيّر بنية البيانات حسب متطلباتنا واحتياجاتنا.

6.1.5 الخلاصة

المصطلحات:

- التعاود - هو مصطلح برمجي يعني اسـتدعاء دالـة من داخلهـا. يمكن اسـتعمال الـدوال التعاوديـة لحـلّ المهام المختلفة بطرق ذكية نظيفة.

حين تستدعي الدالة نفسها نسمّي ذلك "خطـوة تعـاود". تُعـدّ وُسـطاء الدالـة الـتي تبسّـط المهمّـة إلى أقصى درجة بحيث لا تستدعي الدالة أيّ شيء بعدها – تُعدّ "أساس التعاود".

- بنية البيانات التعاودية - هي أيّة بنية بيانات تُحدّد نفسها بنفسها.

فمثلًا يمكن تعريف القائمة المترابطة على أنّها بنية بيانات تحتوي على كـائن يُشيـر إلى قائمـة (أو يُشيـر إلى null).

```
list = { value, next -> list }
```

تُعدّ الأشجار مثل شجرة عناصر HTML أو شجرة الأقسام في هذا الفصل كائنات تعاوديـة بطبعهـا، فهي تتفرّ ع ولكلّ فر ع فروع أخرى.

يمكن استعمال الدوال التعاودية للمرور فيها كما رأينا في مثال `sumSalary`.

يمكن إعادة كتابة أيّ دالة تعاودية لتصير دالة تستعمل التكرار، وغالبًا مـا نفعـل هـذا لتحسـين أداء الـدوال. ولكن هناك مهام عديدة يكون الحلّ التعاودي سريعًا كفايةً وأسهل كتابةً ودعمًا.

6.1.6 تمارين

ا. اجمع كل الأعداد إلى أن تصل للممرّر

الأهمية: ★★★★★

اكتب الدالة `sumTo(n)` التي تجمع الأعداد `1 + 2 + ... + n`.

مثال:

```
sumTo(1) = 1
sumTo(2) = 2 + 1 = 3
sumTo(3) = 3 + 2 + 1 = 6
sumTo(4) = 4 + 3 + 2 + 1 = 10
...
sumTo(100) = 100 + 99 + ... + 2 + 1 = 5050
```

اكتب 3 شيفرات:

1. واحدة باستعمال حلقة for.
2. واحدة باستعمال التعاود، إذ أنّ `sumTo(n) = n + sumTo(n-1)` طالما `n > 1`.
3. واحدة باستعمال المتتاليات الحسابية.

مثال عن الناتج:

```
function sumTo(n) { /*... شيفرتك هنا ... */ }

alert( sumTo(100) ); // 5050
```

ملاحظة: أيّ الشيفرات أسرع من الأخرى؟ وأيها أبطأ؟ ولماذا؟

ملاحظة أخرى: هل يمكن أن نستعمل التعاود لحساب `sumTo(100000)`؟

الحل:

الحلّ باستعمال الحلقة:

```
function sumTo(n) {
  let sum = 0;
  for (let i = 1; i <= n; i++) {
    sum += i;
  }
  return sum;
}

alert( sumTo(100) );
```

الحلّ باستعمال التعاود:

```
function sumTo(n) {
  if (n == 1) return 1;
  return n + sumTo(n - 1);
}

alert( sumTo(100) );
```

الحلّ باستعمال المعادلة `sumTo(n) = n*(n+1)/2`:

```
function sumTo(n) {
  return n * (n + 1) / 2;
}

alert( sumTo(100) );
```

عن الملاحظة: بالطبع فالمعادلة هي أسرع الحلول فلا تستعمل إلا ثلاث عمليات لكلّ عدد `n`. الرياضيات إلى جانبنا هنا!

الشيفرة باستعمال الحلقة تأتي في المرتبة الثانية من ناحية السرعة. لو تلاحظ فنحن هنا نجمع الأعداد نفسها في الشيفرتين التعاودية والحلقية، إلّا أنّ التعاودية فيها استدعاءات متداخلة أكثر وإدارةً لمكدس التنفيذ، هذا ما يستهلك موارد أكثر فتصير الشيفرة أبطأ.

عن الملاحظة الأخرى: تدعم بعض المحرّكات "تحسين نهاية الاستدعاء" (tail call optimization)، ويعني أنّه لو كان الاستدعاء التعاودي هو آخر ما في الدالة (مثلما في الدالة `sumTo` أعلاه)، فلن تواصل الدالة الخارجية عملية التنفيذ كي لا يتذكّر المحرّك سياقها التنفيذي. يتيح هذا للمحرّك إزالة قضية الذاكرة بذلك يكون ممكنًا عدّ

`sumTo(100000)`. ولكن، لو لم يدعم محرّك JavaScript هذا النوع من التحسين (وأغلبها لا تدعم) فستواجه الخطأ: تخطّيت أكبر حجم في المكدس، إذ يُفرض -عادةً- حدّ على إجمالي حجم المكدس.

ب. احسب المضروب

الأهمية: ★★★★☆

المضروب هو عدد طبيعي مضروب بِ "العدد ناقصًا واحد" وثمّ بِ "العدد ناقصًا اثنين" وهكذا إلى أن نصل إلى 1. نكتب مضروب n بهذا الشكل: `n!`

يمكننا تعريف المضروب هكذا:

```
n! = n * (n - 1) * (n - 2) * ...*1
```

قيم المضاريب لأكثر من n:

```
= 1
= 2 * 1 = 2
= 3 * 2 * 1 = 6
= 4 * 3 * 2 * 1 = 24
= 5 * 4 * 3 * 2 * 1 = 120
```

مهمّتك هي كتابة الدالة `factorial(n)` لتحسب `n!` باستعمال الاستدعاءات التعاودية.

```
alert( factorial(5) ); // 120
```

ملاحظة وفائدة: يمكنك كتابة `n!` هكذا `n * (n-1)!` مثلًا: `3! = 3*2! = 3*2*1! = 6`

الحل:

حسب التعريف فيمكن كتابة المضروب `n!` هكذا `n * (n-1)!`.

أي أنّه يمكننا حساب ناتج `factorial(n)` على أنّه n مضروبًا بناتج `factorial(n-1)`. ويمكن أن ينخفض استدعاء `n-1` أنزل وأنزل إلى أن يصل 1.

```
function factorial(n) {
  return (n != 1) ? n * factorial(n - 1) : 1;
}

alert( factorial(5) ); // 120
```

القيمـة الأسـاس للتعـاود هي 1. يمكننـا أن نجعـل 0 هي الأسـاس ولكنّ ذلـك لا يهم، ليسـت إلا خطـوة تعاود أخرى:

```
function factorial(n) {
  return n ? n * factorial(n - 1) : 1;
}

alert( factorial(5) ); // 120
```

ج. أعداد فيبوناتشي

الأهمية: ★★★★★

لمتتالية فيبوناتشي الصيغة Fn = Fn-1 + Fn-2. أي أنّ العدد التالي هو مجموع العددين الذين سبقاه. أوّل عددين هما 1، وبعدها 2(1+1) ثمّ 3(1+2) ثمّ 5(2+3) وهكذا: 1, 1, 2, 3, 5, 8, 13, 21....

ترتبط أعداد فيبوناتشي بالنسبة الذهبية وبظواهر طبيعية أخرى عديدة حولنا من كلّ مكان.

اكتب الدالة fib(n) لتُعيد عدد فيبوناتش n-th. مثال لطريقة عملها:

```
function fib(n) { /* شيفرتك هنا */ }

alert(fib(3)); // 2
alert(fib(7)); // 13
alert(fib(77)); // 5527939700884757
```

يجب أن تعمل الدالة بسرعة. يجب ألا يأخذ استدعاء fib(77) أكثر من جزء من الثانية.

الحل:

أوّل حلّ نفكّر به هو الحلّ بالتعاود. أعداد فيبوناتشي تعاودية حسب تعريفها:

```
function fib(n) {
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

alert( fib(3) ); // 2
alert( fib(7) ); // 13
// fib(77); // سيكون استدعاءً أبطأ من السلحفاة
```

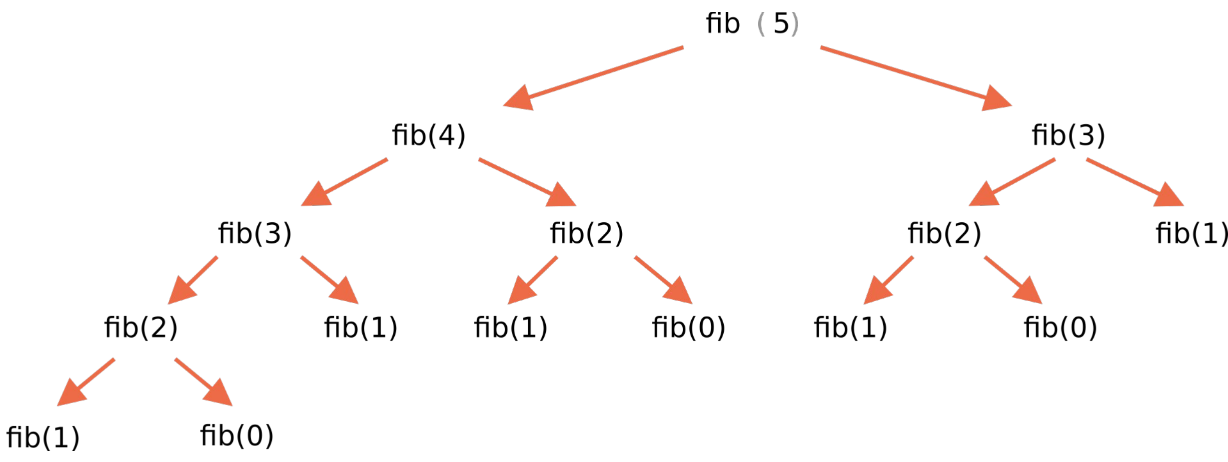
ولكن لو كانت قيمة `n` كبيرة فسيكون بطيئًا جدًا. يمكن أن يعلّق الاستدعاء `fib(77)` محرّك JavaScript لفترة من الوقت بينما يستهلك موارد المعالج كاملةً. يعزو ذلك إلى أنّ الدالة تؤدّي استدعاءات فرعية كثيرة، وتُعيد تقدير (evaluate) القيم ذاتها مرارًا وتكرارًا.

لنرى مثلًا جزءًا من حسابات `fib(5)`:

```
...
fib(5) = fib(4) + fib(3)
fib(4) = fib(3) + fib(2)
...
```

يمكننا أن نرى هنا بأنّ قيمة `fib(3)` مفيدة للاستدعاءين `fib(5)` و`fib(4)`. لذا فستُستدعى `fib(3)` وتُقدّر قيمتها مرتين كاملتين منفصلتين عن بعضهما البعض.

إليك شجرة التعاود كاملةً:



نرى بوضوح كيف أنّ `fib(3)` تُقدّر مرتين اثنتين و`fib(2)` تُقدّر ثلاث مرات. إجمالي الحسابات يزداد أسرع مما تزداد قيمة `n`، ما يجعل الحسابات مهولة حين نصل `n=77`.

يمكننا تحسين أداء الشيفرة بتذكّر القيم التي قدّرنا ناتجها قبل الآن: لو حسبنا قيمة `fib(3)` مثلًا، فيمكننا إعادة استعمالها في أيّ حسابات مستقبلية أو، يمكن أن نترك التعاود كله ونحاول استعمال خوارزمية مختلفة جذريًا تعتمد على الحلقات.

فبدلًا من أن نبدأ بِـ `n` وننطلق نحو أسفل، يمكن أن نصنع حلقة تبدأ من `1` و`2` ثمّ تسجّل ناتج ذلك على أنّه `fib(3)`، وناتج القيمتين السابقتين على أنّه `fib(4)` وهكذا دواليك إلى أن تصل إلى القيمة المطلوبة. هكذا لا نتذكّر في كلّ خطوة إلى قيمتين سابقتين فقط.

إليك خطوات الخوارزمية الجديدة هذه بالتفصيل الممل.

البداية:

```
// a = fib(1)، b = fib(2)، هذه القيم حسب التعريف رقم 1
let a = 1, b = 1;

// نأخذ c = fib(3) ليكون مجموعها
let c = a + b;

/* لدينا الآن fib(1) و fib(2) و fib(3)
a  b  c
2
*/
```

الآن نريد معرفة `fib(4) = fib(2) + fib(3)`.

لنحرّك ما في المتغيّرات إلى الجانب: `a,b` سيكونان `fib(2),fib(3)` و `c` سيكون مجموعهما:

```
a = b; // now a = fib(2)
b = c; // now b = fib(3)
c = a + b; // c = fib(4)

/* الآن: لدينا المتتابعة
   a  b  c
2, 3
*/
```

الخطوة التالية تعطينا عددًا آخر في السلسلة:

```
a = b; // الآن صار a = fib(3)
b = c; // الآن صار b = fib(4)
c = a + b; // c = fib(5)

/* الآن لدينا المتتابعة (أضفنا عددًا آخر):
      a  b  c
2, 3, 5
*/
```

…وهكذا إلى أن نصل إلى القيمة المطلوبة. وهذا أسر ع بكثير من التعاود وليس فيه أيّة حسابات متكرّرة.

الشيفرة كاملةً:

```
function fib(n) {
  let a = 1;
  let b = 1;
  for (let i = 3; i <= n; i++) {
    let c = a + b;
    a = b;
    b = c;
  }
  return b;
}

alert( fib(3) ); // 2
alert( fib(7) ); // 13
alert( fib(77) ); // 5527939700884757
```

تبدأ الحلقة بالقيمة `i=3` إذ أنّ قيمتا المتتابعة الأولى والثانية مكتوبتان داخل المتغيّران `a=1` و `b=1`.

يُدعى هذا الأسلوب بالبرمجة الديناميكية من أسفل إلى أعلى.

د. طباعة قائمة مترابطة

لنقل بأنّ أمامنا القائمة المترابطة هذه:

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};
```

اكتب الدالة `printList(list)` لتطبع لنا عناصر القائمة واحدةً واحدة.

اصنع نسختين من الحل: واحدة باستعمال الحلقات وواحدة باستعمال التعاود.

أيّ الحلّين أفضل؟ بالتعاود أو بدون؟

الحل:

- نسخة الحلقات: هذا الحلّ باستعمال الحلقات:

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};

function printList(list) {
  let tmp = list;

  while (tmp) {
    alert(tmp.value);
    tmp = tmp.next;
  }

}

printList(list);
```

لاحظ كيف استعملنا المتغير المؤقت `tmp` للمرور على عناصر القائمة. يمكننا نظريًا استعمال مُعامل الدالة `list` بدل ذلك:

```
function printList(list) {
  while(list) { // (*)
```

```
      alert(list.value);
      list = list.next;
    }
  }
```

ولكن... سنندم على ذلك لاحقًا إذ قد نحتاج إلى توسيع عمل الدالة وإجراء عملية أخرى غير هذه على القائمة، ولو بدّلنا `list` فلن نقدر على ذلك حتمًا.

وعلى سيرة الحديث عن تسمية المتغيرات "كما ينبغي"، فهنا تُعدّ القائمةُ `list` ذاتَ القائمة، أي العنصر الأوّل من تلك القائمة، ويجب أن يبقى الاسم كما هو هكذا، مقروءًا وواضحًا.

بينما لا يعدو دور `tmp` إلّا أداةً لمسح القائمة، تمامًا مثل `i` في حلقات `for`.

- نسخة التعاود: مفهوم النسخة التعاودية من الدالة `printList(list)` بسيط: علينا -كي نطبع قائمةً- طباعةَ العنصر الحالي `list` وتكرار ذلك على كلّ `list.next`:

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};
function printList(list) {
  alert(list.value); // نطبع العنصر الحالي
  if (list.next) {
    printList(list.next); // ذات الحركة لكلّ عنصر باقٍ في القائمة
  }
}
printList(list);
```

أمّا الآن، فأيّ النسختين أفضل؟

نظريًا تُعدّ نسخة الحلقات أفضل أداءً. صحيح أنّ الاثنتين عملهما واحد إلّا أن الحلقات لا تستهلك الموارد بتداخل استدعاءات الدوال.

ولكن لو نظرنا للجهة الأخرى من الكأس فالنسخة التعاودية أقصر وأسهل فهمًا أحيانًا.

ه. طباعة قائمة مترابطة بترتيب معكوس

الأهمية: ★★★★★

اطبع القائمة المترابطة من التمرين السابق، ولكن بعكس ترتيب العناصر.

اصنع نسختين من الحل: واحدة باستعمال الحلقات وواحدة باستعمال التعاود.

الحل:

- نسخة التعاود: هنا توجد خدعة في فكرة التعاود، إذ علينا أوّلًا طباعة الباقي من القائمة وبعدها طباعة القائمة الحالية:

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};

function printReverseList(list) {
  if (list.next) {
    printReverseList(list.next);
  }
  alert(list.value);
}
printReverseList(list);
```

- نسخة الحلقات: نسخة الحلقات هنا أكثر تعقيدًا (بقليل) عن سابقتها. فما من طريقة لنأخذ آخر قيمة في قائمتنا `list`، ولا يمكننا أن "نعود" فيها. لذا يمكننا أوّلًا المرور على العناصر بالترتيب المباشر وحِفظها في مصفوفة، بعدها طباعة ما حفظناه بالعكس:

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};

function printReverseList(list) {
  let arr = [];
  let tmp = list;

  while (tmp) {
    arr.push(tmp.value);
    tmp = tmp.next;
  }

  for (let i = arr.length - 1; i >= 0; i--) {
    alert( arr[i] );
  }
}

printReverseList(list);
```

لاحظ كيف أنّ الحل باستعمال التعاود هو بالضبط كما باستعمال الحلقات، إذ يتبع القائمة ويحفظ العناصر في سلسلة من الاستدعاءات المتداخلة (في مكدس سياق التنفيذ)، وبعدها يطبع القيم.

6.2 المعاملات "البقية" ومعامل التوزيع

تتوقّع العديد من دوال JavaScript المضمّنة في اللغة عددًا من الوُسطاء لا ينتهي مثل:

- `Math.max(arg1, arg2, ..., argN)` - يُعيد أكبر وسيط من الوُسطاء.
- `Object.assign(dest, src1, ..., srcN)` - ينسخ الخصائص من `src1..N` إلى `dest`.
- ...وهكذا.

ستتعلّم في هذا الفصل كيف نفعل ذلك أيضًا، كما وكيف نمرّر المصفوفات إلى هذه الدوال على أنّها مُعاملات.

6.2.1 المُعاملات "البقية" ...

يمكن أن ننادي الدالة بأيّ عدد من الوُسطاء كيفما كانت الدالة معرَّفة، مثل:

```
function sum(a, b) {
  return a + b;
}

alert( sum(1, 2, 3, 4, 5) );
```

لن ترى أيّ خطأ بسبب تلك الوُسطاء "الزائدة". ولكن طبعًا فالنتيجة لن تأخذ بالحسبان إلا أوّل اثنين.

يمكن تضمين بقية المُعاملات في تعريف الدالة باستعمال الثلاث نقاط `...` ثمّ اسم المصفوفة التي ستحتويهم. تعني تلك النقط حرفيًا "اجمع المُعاملات الباقية في مصفوفة".

فمثلًا لجمع كلّ الوُسطاء في المصفوفة `args`:

```
function sumAll(...args) { // args هو المصفوفة اسم
  let sum = 0;

  for (let arg of args) sum += arg;
  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

يمكن لو أردنا أن نأخذ المُعاملات الأولى في متغيّرات ونجمع البقية فقط.

هنا نأخذ الوسيطين الأولين في متغيرات والباقي نرميه في المصفوفة `titles`:

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Julius Caesar
  // titles الباقي نضعه في مصفوفة الأسماء
  // titles = ["Consul", "Imperator"] مثلًا
  alert( titles[0] ); // Consul
  alert( titles[1] ); // Imperator
  alert( titles.length ); // 2
}

showName("Julius", "Caesar", "Consul", "Imperator");
```

يجب أن تُترك المُعاملات البقية إلى النهاية

تجمع المُعاملات البقية كلّ الوُسطاء التي بقيت، وبهذا فالآتي ليس منطقيًا وسيتسبّب بخطأ:

```
function f(arg1, ...rest, arg2) { // !الوسيط arg2 بعد ...البقية؟
  // خطأ
}
```

يجب أن يكون `...rest` الأخير دومًا.

6.2.2 متغير الوسطاء arguments

هناك كائن آخر شبيه بالمصفوفات يُدعى `arguments` ويحتوي على كلّ الوُسطاء حسب ترتيب فهارسها مثل هذا المثال:

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );
  // المصفوفة مُتعدِّدة
  // for(let arg of arguments) alert(arg);
}
// Julius, Caesar ,2 :تعرض
showName("Julius", "Caesar");
// (ما من مُعطى ثانٍ) Ilya, undefined ,1 :تعرض
showName("Ilya");
```

قديمًا لم تكن المُعاملات البقية موجودة في اللغة ولم يكن لدينا سوى استعمال `arguments` لنجلب كلّ مُعاملات الدالة. وما زالت تعمل الطريقة إلى يومنا هذا ويمكن أن تراها في الشيفرات القديمة.

ولكن السلبية هنا هي أنّ `arguments` ليست مصفوفة (على الرغم من أنّها شبيهة بالمصفوفات ومُتعدّدة). بهذا لا تدعم توابع المصفوفات فلا ينفع أن نستدعي عليها `arguments.map(...)` مثلًا. كما وأنّها تحتوي على كل الوُسطاء دومًا. لا يمكن أن نأخذ منها ما نريد كما نفعل مع المُعاملات البقية. لهذا متى ما احتجنا إلى ميزة كهذه، فالأفضل استعمال المُعاملات البقية بدلًا من `arguments`.

ليس للدوال السهمية "arguments"

لو حاولت الوصول إلى كائن الوُسطاء `arguments` من داخل الدالة السهمية، فستستلم الناتج من الدالة "الطبيعية" الخارجية. إليك مثالًا:

```
function f() {
  let showArg = () => alert(arguments[0]);
  showArg();
}

f(1); // 1
```

كما نذكر فليس للدوال السهمية قيمة `this` تخصّها، أمّا الآن صرنا نعلم بأنّ ليس لها كائن `arguments` أيضًا.

6.2.3 معامل التوزيع

رأينا كيف نأخذ مصفوفة من قائمة من المُعطيات. ولكن ماذا لو أردنا العكس من ذلك؟

فمثلًا لنقل أردنا استعمال الدالة المبنية في اللغة `Math.max` والتي تُعيد أكبر عدد من القائمة:

```
alert( Math.max(3, 5, 1) ); // 5
```

لنقل أنّ لدينا المصفوفة `[3, 5, 1]`. كيف نستدعي `Math.max` عليها؟

لا ينفع تمريرها "كما هي" لأنّ `Math.max` يتوقّع قائمةً بالوُسطاء العددية لا مصفوفة واحدة:

```
let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN
```

وطبعًا لا يمكن أن نفكّ عناصر القائمة يدويًا في الشيفرة `Math.max(arr[0], arr[1], arr[2])` لأنّنا في حالات لا نعرف كم من عنصر هناك أصلًا. وما إن يُنفَّذ السكربت يمكن أن يكون فيه أكبر مما كتبناه أو حتّى لا شيء أصلًا، وسنحصد لاحقًا ما جنته هذه الشيفرة.

عاش مُنقذنا مُعامل التوزيع! عاش عاش عاش! من بعيد نراه مشابهًا تمامًا للمُعاملات البقية، كما ويستعمل `...`، إلّا أنّ وظيفته هي العكس تمامًا.

فحين نستعمل `...arr` في استدعاء الدالة، "يتوسّع" الكائن المُتعدَّد `...arr` إلى قائمة من الوُسطاء.

فمثلًا نعود إلى `Math.max`:

```
let arr = [3, 5, 1];

// 5 (يحوّل التوزيع المصفوفة إلى قائمة من الوُسطاء)
alert( Math.max(...arr) );
```

يمكن أيضًا أن نمرّر أكثر من مُتعدَّد واحد بهذه الطريقة:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(...arr1, ...arr2) ); // 8
```

أو حتّى ندمج مُعامل التوزيع مع القيم العادية:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25
```

كما يمكن أن نستعمل مُعامل التوزيعة لدمج المصفوفات:

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];
alert(merged); // 0,3,5,1,2,8,9,15 (0 ثمّ arr ثمّ 2 ثمّ arr2)
```

استعملنا في الأمثلة أعلاه مصفوفة لنشرح مُعامل التوزيع، إلّا أنّ المُكرَّرات (iterable) أيًا كانت تنفع أيضًا.

فمثلًا نستعمل هنا مُعامل التوزيع لنحوّل السلسلة النصية إلى مصفوفة محارف:

```
let str = "Hello";
alert( [...str] ); // H,e,l,l,o
```

يستعمل مُعامل التوزيع هذا داخليًا المُعدِّدات لجمع العناصر، كما تفعل حلقة `for..of`. لذا لو استلمت `for..of` سلسلةً نصيّة فتُعيد لنا المحارف وتصير `...str` بالقيمة `"H","e","l","l","o"`. وهكذا تُمرّر قائمة المحارف إلى مُهيّئ المصفوفة `[...str]`.

يمكننا أيضًا لهذه المهمة استعمال `Array.from` إذ أنّه يحوّل المُكرَّر (مثل السلاسل النصية) إلى مصفوفة:

```
let str = "Hello";
// يُحوّل Array.from المُكرَّر إلى مصفوفة
alert( Array.from(str) ); // H,e,l,l,o
```

ناتجه هو ذات ناتج `[...str]`.

ولكن… هناك فرق ضئيل بين `Array.from(obj)` و`[...obj]`:

- يعمل `Array.from` على الشبيهات بالمصفوفات والمُكرَّرات.
- ويعمل مُعامل التوزيع على المُكرَّرات فقط لا غير.

لذا لو أردت تحويل شيء إلى مصفوفة فالتابع `Array.from` أكثر استعمالًا وشيوعًا.

6.2.4 الخلاصة

متى رأينا `"..."` في الشيفرة نعرف أنّه إمّا المُعاملات البقية وأمّا مُعامل التوزيع.

إليك طريقة بسيطة للتفريق بينهما:

- حين ترى `...` موجودة في نهاية مُعاملات الدالة (أي في تعريفها) فهي "المُعاملات البقية" وستجمع بقية قائمة الوُسطاء في مصفوفة.
- وحين ترى `...` في استدعاء دالة أو ما شابهه فهو "مُعامل توزيع" يوسّع المصفوفة إلى قائمة.

طرائق الاستعمال:

- تُستعمل المُعاملات البقية لإنشاء دوال تقبل أيّ عدد كان من الوُسطاء.
- يُستعمل مُعامل التوزيع لتمرير مصفوفة إلى دوال تطلب (عادةً) قائمة طويلة من الوُسطاء.

كلا الميزتين تساعدك في التنقل بين القائمة ومصفوفة المُعاملات بسهولة ويُسر.

يمكنك أيضًا أن ترى كل وُسطاء استدعاء الدالة "بالطريقة القديمة" `arguments` وهو كائن مُتعدَّد شبيه بالمصفوفات.

6.3 المنغلقات ومجال المتغيرات

لغة JavaScript هي لغة وظيفية التوجه function-oriented language (قد تترجم حرفيًا إلى "لغة دالية التوجه") إلى أقصى حدّ، فتعطينا أقصى ما يمكن من حريّة. يمكننا إنشاء الدوال ديناميكيًا ونسخها إلى متغيرات أخرى أو تمريرها كوسيط إلى دالة أخرى واستدعائها من مكان آخر تمامًا لاحقًا حين نريد.

كما نعلم بأنّ الدوال تستطيع الوصول إلى المتغيرات خارجها. نستعمل هذه الميزة كثيرًا. ولكن، ماذا يحدث حين يتغيّر المتغيّر الخارجي؟ هل تستلم الدالة أحدث قيمة له أو تلك التي كانت موجودة لحظة إنشاء الدالة؟

كما وماذا يحدث حين تنتقل الدالة إلى مكان آخر في الشيفرة واستُدعَت من ذلك المكان: هل يمكنها الوصول إلى المتغيرات الخارجية في المكان الجديد؟

يختلف سلوك اللغات عن بعضها من هذه الناحية. في هذا الفصل سنتحدّث عن سلوك JavaScript.

6.3.1 أسئلة تحتاج أجوبة

لنبدأ أولًا بحالتين اثنتين ندرس بهما الآلية الداخلية للغة خطوةً بخطوة، بهذا ستملك ما يكفي لتُجيب على الأسئلة الآتية وأخرى غيرها أكثر تعقيدًا في المستقبل.

- تستعمل الدالة `sayHi` المتغير الخارجي `name`. ما القيمة التي ستستعملها الدالة حين تعمل؟

```
let name = "Ahmad";

function sayHi() {
  alert("Hi, " + name);
}

name = "Pete";

sayHi(); // ماذا ستعرض؟ "Ahmad" أم "Pete"؟
```

يشيع وجود هذه الحالات في المتصفّحات كما وفي الخواديم عند التطوير. يمكن أن تعمل الدالة بعدما تُنشأ بفترة لو أراد المطوّر (مثلًا بعد أن يتفاعل المستخدم أو يستلم المتصفّح طلبًا من الشبكة).

إذًا فالسؤال هو: هل تستعمل آخر التعديلات؟

- تصنع الدالة `makeWorker` دالةً أخرى وتُعيدها، ويمكن أن نستعدي تلك الدالة الجديدة من أيّ مكان آخر نريد. السؤال هو: هل يمكنها الوصول إلى المتغيرات الخارجية تلك التي من مكان إنشائها الأصلي، أم تلك التي في المكان الجديد، أم من المكانين معًا؟

```
function makeWorker() {
  let name = "Pete";

  return function() {
    alert(name);
  };
}

let name = "Ahmad";

// نصنع الدالة
let work = makeWorker();

// نستدعيها
work();
```

ماذا ستعرض الدالة `work()`؟ Pete الاسم الذي تراه عند الإنشاء أم Ahmad الاسم الذي تراه عند الاستدعاء؟

6.3.2 البيئات المعجمية

علينا أولًا أن نعرف ما هو "المتغير" هذا أصلًا لتُدرك ما يجري بالضبط.

في لغة JavaScript، كلّ دالة عاملة أو كتلة شفرات `{...}` أو حتّى السكربت كلّه - تملك كائنًا داخليًا مرتبطًا بها (ولكنّه مخفي) يُدعى "بالبيئة المُعجمية" (Lexical Environment).

تتألّف كائنات البيئات المُعجمية هذه من قسمين:

1. "سجلّ مُعجمي" (Environment Record): وهو كائن يخزّن كافة المتغيرات المحلية على أنّها خاصيات له (كما وغيرها من معلومات مثل قيمة `this`).
2. إشارة إلى "البيئة المُعجمية الخارجية" - أي المرتبطة مع الشيفرة الخارجية للكائن المُعجمي.

ليس "المتغير" إلا خاصية لإحدى الكائنات الداخلية الخاصة: السجل المُعجمي (`Environment Record`). وحين نعني "بأخذ المتغير أو تغيير قيمته" فنعني "بأخذ خاصية ذلك الكائن أو تغيير قيمتها".

إليك هذه الشيفرة البسيطة مثالًا (فيها بيئة مُعجمية واحدة فقط):

```js
let phrase = "Hello";
alert(phrase);
```

البيئة المُعجمية

phrase: "Hello"

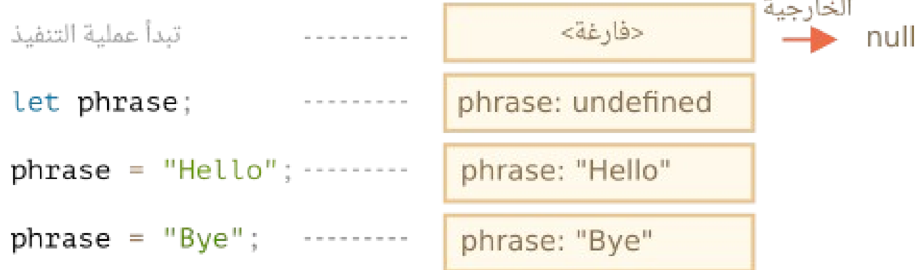
الخارجية

null

هذا ما نسمّيه البيئة المُعجمية العمومية (global) وهي مرتبطة بالسكربت كاملًا.

نعني بالمستطيل (في الصورة أعلاه) السجل المُعجمي (أي مخزن المتغيرات)، ونعني بالسهم الإشارة الخارجية له. وطالما أنّ البيئة المُعجمية العمومية ليس لها إشارة خارجية، فذاك السهم يُشير إلى `null`.

وهكذا تتغيّر البيئة حين تعرّف عن متغيّر وتُسند له قيمة:



نرى في المستطيلات على اليمين كيف تتغيّر البيئة المُعجمية العمومية أثناء تنفيذ الشيفرة:

1. حين تبدأ الشيفرة، تكون البيئة المُعجمية فارغة.
2. بعدها يظهر التصريح `let phrase`، لكن لم تُسند للمتغيّر أيّ قيمة، لذا تخزّن البيئة `undefined`.
3. تُسند للمتغير `phrase` قيمة.
4. وهنا تتغيّر قيمة `phrase`.

بسيط حتّى الآن، أم لا؟

نلخّص الموضوع:

- المتغير هو فعليًا خاصية لإحدى الكائنات الداخلية الخاصة، وهذا الكائن مرتبط بالكتلة أو الدالة أو السكربت الذي يجري تنفيذه حاليًا.
- حين نعمل مع المتغيرات نكون في الواقع نعمل مع خصائص ذلك الكائن.

ا. تصريحات الدوال Function Declarations

لم نرى حتّى اللحظة إلا المتغيرات. حان وقت تصريحات الدوال.

الدوال على عكس متغيرات `let`، فليست تُهيّأ تمامًا حين تصلها عملية التنفيذ، لا، بل قبل ذلك حين تُنشأ البيئة المُعجمية.

وحين نتكلم عن أعلى الدوال مستوًى، فنعني ذلـك لحظـة بـدء السـكربت. ولهـذا السـبب يمكننـا اسـتدعاء الدوال التي صرّحناها حتّى قبل أن نرى ذاك التعريف.

نرى في الشيفرة أدناه كيف أنّ البيئة المُعجمية تحتوي شيئًا منذ بداية التنفيذ (وليست فارغة)، وما تحتويـه هي `say` إذ أنّها تصريح عن دالة. وبعدها تسجّل `phrase` المُصرّح باستعمال `let`:

تبدأ عملية التنفيذ --------- say: function ← الخارجية null

```
let phrase = "Hello";        ---------  say: function
                                        phrase: "Hello"
function say(name) {
  alert( `${phrase}, ${name}` );
}
```

ب. البيئات المُعجمية الداخلية والخارجية

الآن لنتعمّق ونرى ما يحدث حين تحاول الدالة الوصول إلى متغير خارجي.

تستعمل `say()` أثناء الاستدعاء المتغير الخارجي `phrase`. لنرى تفاصيـل مـا يجـري بالضـبط. تُنشأ بيئـة مُعجمية تلقائيًا ما إن تعمل الدالة وتخزّن المتغيرات المحلية ومُعاملات ذلك الاستدعاء.

فمثلًا هكذا تبدو بيئة استدعاء `say("Ahmad")` (وصل التنفيذ السطر الذي عليه سهم):

```
let phrase = "Hello";

function say(name) {
 alert( `${phrase}, ${name}` );
}

say("Ahmad"); // Hello, Ahmad
```

البيئة المُعجمية الخاصة بالاستدعاء

```
let phrase = "Hello";

function say(name) {
▶ alert( `${phrase}, ${name}` );     name: "John" ← الخارجية   say: function       ← الخارجية null
}                                                              phrase: "Hello"

say("John"); // Hello, John
```

إذًا... حين نكون داخل استدعاءً لأحد الدوال نرى لدينا بيئتين مُعجميتين: الداخلية (الخاصة باستدعاء الدالة) والخارجية (العمومية):

- ترتبط البيئة المُعجمية الداخلية مع عملية التنفيذ الحالية للدالة `say`. تملك خاصية واحـدة فقـط: `name` (وسيط الدالة). ونحن استدعينا `say("Ahmad")` بهذا تكون قيمة `name` هي `"Ahmad"`.

- البيئة المُعجمية الخارجية وهي هنا البيئة المُعجمية العمومية. تملك متغير `phrase` والدالة ذاتها.

للبيئة المُعجمية الداخلية إشارة إلى تلك "الخارجية".

حين تريد الشيفرة الوصول إلى متغير من المتغيرات، يجري البحث أولًا في البيئة المُعجمية الداخلية، وبعدها الخارجية، والخارجية أكثر وأكثر حتى نصل العمومية.

لو لم يوجد المتغير في عملية البحث تلك فسترى خطأً (لو استعملت النمط الصارم Strict Mode). لو لم تستعمل `use strict` فسيُنشئ الإسناد إلى متغير غير موجود (مثل `user = "Ahmad"`) متغيرًا عموميًا جديدًا باسم `user`. سبب ذلك هو التوافق مع الإصدارات السابقة.

لنرى عملية البحث تلك في مثالنا:

- حين تحاول `alert` في دالة `say` الوصول إلى المتغير `name` تجده مباشرةً في البيئة المُعجمية للدالة.
- وحين تحاول الوصول إلى متغير `phrase` ولا تجده محليًا، تتبع الإشارة في البيئة المحلية وتصل البيئة المُعجمية خارجها، وتجد المتغير فيها.

```
let phrase = "Hello";

function say(name) {
  alert( `${phrase}, ${name}` );
}

say("John"); // Hello, John
```

name: "John" ← الخارجية → say: function phrase: "Hello" ← الخارجية → null

يمكننا أخيرًا تقديم إجابة على السؤال في أول الفصل.

تأخذ الدالة المتغيرات الخارجية من مكانها الآن، أي أنها تستعمل أحدث القيم. لا القيم القديمة في أي مكان مهما بحثت. فحين تريد إحدى الدوال متغيرًا ما تأخذ قيمته الحالية من بيئتها المُعجمية هي أو الخارجية بالنسبة لها. إذًا، إجابة السؤال الأول هي `Pete`:

```
let name = "Ahmad";
function sayHi() {
  alert("Hi, " + name);
}
name = "Pete"; // (*)
sayHi(); // (*) Pete
```

سير تنفيذ الشيفرة أعلاه:

1. للبيئة المُعجمية العمومية `name: "Ahmad"`.

2. في السطر `(*)` يتغيّر المتغير العمومي ويصير الآن `name: "Pete"`.

3. تأخذ الدالة `sayHi()` حين تُنفَّذ قيمة `name` من الخارج (أي البيئة المُعجمية العمومية) حيث صارت الآن `"Pete"`.

لكلّ استدعاء منك، بيئة مُعجمية من اللغة

لاحظ بأنّ محرّك اللغة يُنشئ بيئة مُعجمية جديدة للدالة في كلّ مرة تعمل فيها الدالة، ولو استدعيت الدالة أكثر من مرة فلكلّ استدعاء منها بيئة مُعجمية خاصة بها مستقلة المتغيرات المحلية والمُعاملات، ومخصّصة فقط لذلك الاستدعاء.

البيئات المُعجمية كائن في توصيف اللغة

كائن "البيئة المُعجمية" (Lexical Environment) هو كائن في توصيف اللغة، أي أنّه موجود "نظريًا" فقط في توصيف اللغة لشرح طريقة عمل الأمور، ولا يمكننا أخذ هذا الكائن في الشيفرة ولا التعديل عليه مباشرةً. كما يمكن أن تُحسّن محرّكات JavaScript هذا الكائن أو تُهمل المتغيرات غير المستخدمة فتحفظ الذاكرة أو غيرها من خُدع داخلية، كلّ هذا بمنأى عن السلوك الظاهر لنا فيظلّ كما هو.

6.3.3 الدوال المتداخلة

تكون الدالة "متداخلة" متى صنعتها داخل دالة أخرى. ويمكنك بسهولة بالغة فعل ذلك داخل JavaScript.

يمكننا استعمال هذه الميزة لتنظيم الشيفرة الإسباغيتية، هكذا:

```
function sayHiBye(firstName, lastName) {

  // دالة مساعدة متداخلة نستعملها أسفله
  function getFullName() {
    return firstName + " " + lastName;
  }

  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );

}
```

صنعنا هنا الدالة المتداخلة `getFullName()` لتسهّل حياتنا علينا، فيمكنها هي الوصول إلى المتغيرات الخارجية وإعادة اسم الشخص الكامل. كثيرًا ما نستعمل الدوال المتداخلة في JavaScript.

والممتع أكثر هو أنّه يمكننا إعادة الدوال المتداخلة، إمّا باعتبارها خاصية لكائن جديد (لو كانت الدالة الخارجية تصنع كائنًا له توابع) أو أن تكون نتيجةً للدالة مستقلة بذاتها. ويمكننا لاحقًا استعمالها أينما أردنا، وأيّما كان مكانها الجديد فلديها الحقّ بالوصول إلى ذات المتغيرات الخارجية تلك.

مثال على ذلك: أسندنا الدالة المتداخلة إلى كائن جديد باستعمال دالة مُنشئة:

```
// تُعيد الدالة المُنشئة كائنًا جديدًا
function User(name) {

  // نصنع تابع الكائن على أنّه دالة متداخلة
  this.sayHi = function() {
    alert(name);
  };
}

let user = new User("Ahmad");
// يمكن أن تصل شيفرة تابع الكائن "sayHi" إلى "name" الخارجي
user.sayHi();
```

وهنا أنشأنا دالة "عدّ" وأعدناها، لا أكثر:

```
function makeCounter() {
  let count = 0;

  return function() {
    // يمكنها الوصول إلى  متغير "count" الخارجي
    return count++;
  };
}
let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

لنتفحّص مثال `makeCounter`. تصنع الشيفرة دالة "العدّ" وتُعيد العدد التالي كلّما استدعيناها. صحيح أنّ الدالة بسيطة لكن بتعديلها قليلًا يمكن استعمالها لأمور عديدة مفيدة مثل مولّدات الأعداد شبه العشوائية وغيرها.

ولكن كيف يعمل هذا العدّاد داخليًا؟

متى ما عملت الدالة الداخلية، تبدأ بالبحث عن المتغير في `count++` بدءًا منها وانطلاقًا إلى خارجها. فهكذا سيكون الترتيب في المثال أعلاه:

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}
```

1. المتغيرات المحلية للدالة المتداخلة...
2. المتغيرات المحلية للدالة الخارجية...
3. وهكذا حتى نصل المتغيرات العمومية.

في هذا المثال وجدنا المتغير `count` في الخطوة الثانية. فلو عُدِّلت قيمة المتغير الخارجي فيحدث هذا التعديل في المكان الذي وجدنا المتغير فيه. لهذا تجد `count++` المتغير الخارجي وتزيد قيمته في البيئة المُعجمية التي ينتمي المتغير إليها، تمامًا كما لو استعملنا `let count = 1`.

إليك سؤالين تفكّر بهما (أيضًا):

1. هل يمكننا بطريقة أو بأخرى تصفير العدّاد `count` من الشيفرة التي لا تنتمي إلى `makeCounter`؟ مثلًا بعد استدعاءات `alert` في المثال أعلاه.
2. حين نستدعي `makeCounter()` أكثر من مرة تُعيد لنا دوال `counter` كثيرة. فهل هي مستقلة بذاتها أم تتشارك ذات متغير `count`؟

حاول حلّ السؤالين قبل مواصلة القراءة ... انتهيت؟

إذًا حان وقت الإجابات.

1. ما من طريقة أبدًا: متغير `count` هو متغير محلي داخل إحدى الدوال ولا يمكننا الوصول إليه من الخارج.
2. كلّ استدعاء من `makeCounter()` يصنع بيئة مُعجمية جديدة للدالة لها متغير `count` خاص بها. لذا فدوال `counter` الناتج مستقلة عن بعضها البعض.

إليك شيئًا تجربّه بنفسك:

```
function makeCounter() {
  let count = 0;
  return function() {
    return count++;
  };
}
let counter1 = makeCounter();
let counter2 = makeCounter();

alert( counter1() ); // 0
alert( counter1() ); // 1
alert( counter2() ); // (مستقل) 0
```

آمل بأنّ الصورة الآن صارت أوضح أكثر. صحيح أنّا تكلمنا فقط عن المتغيرات الخارجية ولكن إدراك هذا القدر فقط يكفي أغلب الأحيان. هناك طبعًا تفاصيل أخرى في مواصفات اللغة لم نتحدّث عنها للإيجاز. في القسم التالي سنتكلم عن هذه التفاصيل أكثر.

6.3.4 البيئات بالتفصيل الممل

إليك ما يجري في مثال `makeCounter` خطوةً بخطوة. احرص على اتباعه لتحرص على فهم آلية عمل البيئات بالتفصيل. لاحظ أنّا شرحنا الخاصية الإضافية `[[Environment]]` هنا، ولم نشرحها سابقًا للتبسيط.

أولًا، حين يبدأ السكربت لا يكون هناك إلا بيئة مُعجمية عمومية:

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}
...
```

في تلك اللحظة ليس هناك إلا دالة `makeCounter` إذ أنها تصريح عن دالة، ولم يبدأ تشغيلها بعد.

تستلم كافة الدوال "لحظة إفاقتها للحياة" خاصية مخفية باسم `[[Environment]]` فيها إشارة إلى البيئة المُعجمية حيث أُنشئت. لم نتحدّث عن هذه قبلًا، وهي الطريقة التي تعرف الدالة فيها مكان صناعتها الأولي.

هنا أُنشأت `makeCounter` في البيئة المُعجمية العمومية، ولهذا فتُبقي `[[Environment]]` إشارة إليها. أي وبعبارة أخرى "نطبع" على الدالة إشارةً للبيئة المُعجمية التي نشأت بها وخاصية `[[Environment]]` هي الخاصية الدالية المخفية التي تسجّل تلك الإشارة.

ثانيًا، تبدأ أخيرًا الشيفرة بالعمل، ويرى المحرّك متغيرا عموميًا جديدًا بالاسم `counter` صرّحنا عنه وقيمته هي ناتج استدعاء `makeCounter()`. إليك نظرة على اللحظة التي تكون فيها عملية التنفيذ على أول سطر داخل `makeCounter()`:

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
```

البيئة المُعجمية لاستدعاء makeCounter()

count: 0 → الخارجية

البيئة المُعجمية العمومية

makeCounter: function

counter: undefined

الخارجية → null

تُنشأ بيئة مُعجمية لحظة استدعاء `makeCounter()` لتحمل متغيراتها ومُعاملاتها.

وكما الحال مع البيئات هذه فهي تخزّن أمرين:

- سجلّ بيئي فيه المتغيرات المحلية. في حالتنا هنا متغير `count` هو الوحيد المحلي (يظهر حين يُنفّذ سطر `let count`).
- الإشارة إلى البيئة المُعجمية الخارجية (وتُضبط قيمة لخاصية `[[Environment]]` للدالة). تُشير هنا `[[Environment]]` للدالة `makeCounter` إلى البيئة المُعجمية العمومية.

إذًا لدينا بيئتين مُعجميتين اثنتين: الأولى عمومية والثانية مخصّصة لاستدعاء `makeCounter` الحالي، بينما الإشارة الخارجية لها هي البيئة العمومية.

ثالثًا، تُصنع -أثناء تنفيذ `makeCounter()`- دالة صغيرة متداخلة.

لا يهمّنا إن كان التصريح عن الدالة أم تعبير الدالة هو من أنشأ... الدالة، فالخاصية `[[Environment]]` تُضاف لكل الدوال، وتُشير إلى البيئة المُعجمية التي صُنعت فيها تلك الدوال. وبطبيعة الحال فهذه الدالة الصغيرة المتداخلة لديها نصيب من الكعكة.

قيمة الخاصية `[[Environment]]` للدالة المتداخلة هذه هي البيئة المُعجمية الحالية للدالة `makeCounter()` (مكان صناعة الدالة المتداخلة):

```
function makeCounter() {
  let count = 0;

  return function() { [[Environment]]
    return count++;
  };
}

let counter = makeCounter();
```

متغير البيئة → count: 0 → الخارجية

makeCounter: function

counter: undefined

الخارجية → null

لاحـظ أنّ الدالـة الداخليـة (في هـذه الخطـوة) أُنشـئت، صـحيح، ولكن لم نسـتعدها بعـد. الشـيفرة في `return count++;` لا تعمل.

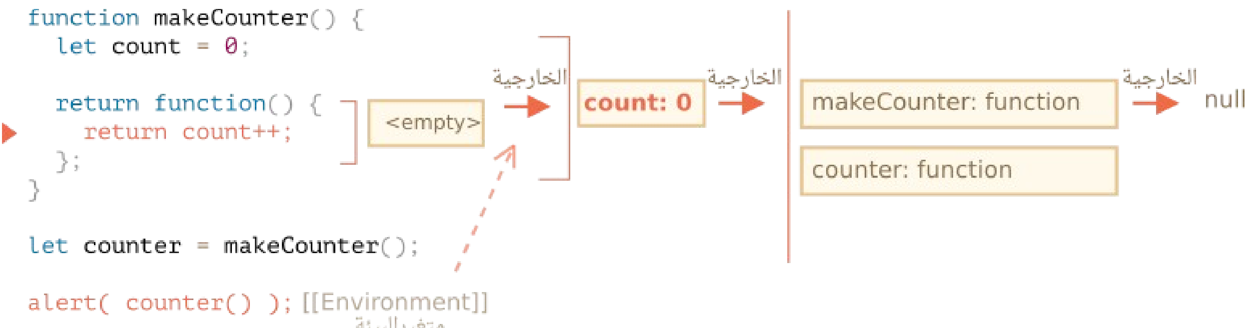
رابعًا، تُواصل عملية التنفيذ العمل وينتهي استدعاء `makeCounter()` ويُسنَد ناتجها (وهو الدالة المتداخلة الصغيرة) إلى المتغير العمومي `counter`:

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() );
```

[[Environment]] متغيرالبيئة → count: 0 الخارجية → makeCounter: function / counter: function الخارجية → null

ليس لتلك الدالة إلا سطرًا واحدًا: `return count++` وسيُنفّذ ما إن نشغّل الدالة.

خامسًا، وحين استدعاء `counter()` تُنشأ بيئة مُعجمية جديدة، لكنّهـا فارغـة إذ أن ليس للدالـة `counter` متغيرات محلية فيها، إلّا أنّ لخاصية الدالة `counter` أي `[[Environment]]` فائدة ففيها الإشارة "الخارجية" للدالة وهي التي تتيح لنا الوصول إلى متغيرات استدعاء `makeCounter()` السابق متى ما أنشأناه:



أما الآن فحين يبحث الاستدعاء عن متغير `count` فهـو يبحث أولًا في بيئتـه المُعجميـة (الفارغـة)، فلـو لم يجدها بحث في البيئة المُعجمية لاستدعاء `makeCounter()` الخارجي، ويجد المتغير فيه.

لاحظ آلية إدارة الذاكرة هنا. صحيح أنّ استدعاء `makeCounter()` انتهى قبل فـترة إلا أن بيئتـه المُعجميـة بقيت في الذاكرة لأنّ الدالة المتداخلة تحمل الخاصية `[[Environment]]` التي تُشير إلى تلك البيئة.

يمكن القول بصفة عامة بأنّ البيئة المُعجمية لا تمـوت طالمـا يمكن لدالـة من الـدوال اسـتعمالها. وحين لا توجد هكذا دالة - تُمسح البيئة آنذاك.

سادسًا، لا يُعيد استدعاء `counter()` قيمة الخاصية `count` فحسـب، بل أيضًا يزيدها واحدًا. لاحـظ كيـف أنّ التعديل حدث "في ذات مكانه" (In place)، فتعدّلت قيمة `count` في البيئة ذاتها التي وجدناه فيها.

```
function makeCounter() {
  let count = 0;

  return function() { [[Environment]]
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() );
```

متغير البيئة

تغيّرت هنا

count: 1 ← الخارجية → makeCounter: function / counter: function ← الخارجية → null

سابقًا، تعمل استدعاءات `counter()` التالية بنفس الطريقة.

أفترض الآن بأنّ إجابة السؤال الثاني في أول الفصل ستكون جليّة. دالة `work()` في الشيفرة أدناه تأخذ الاسم `name` من مكانه الأصل عبر إشارة البيئة المُعجمية الخارجية إليه:

```
function makeWorker() {
  let name = "Pete";

  return function() {
    alert(name);
  };
}

let name = "John";
let work = makeWorker();

work(); // Pete
```

<فارغة> ← الخارجية → name: "Pete" ← الخارجية → makeWorker: function name: "John" ← الخارجية → null

إذًا، فالناتج هنا هو `"Pete"`. ولكن لو لم نكتب `let name` في `makeWorker()` فسينتقل البحث إلى خارج الدالة تلك ويأخذ القيمة العمومية كما نرى من السلسلة أعلاه. في تلك الحالة سيكون الناتج `"Ahmad"`.

المنغلقات Clousures

هناك مصطلح عام يُستعمل في البرمجة باسم "المُنغلِق" Clousure ويُفترض أن يعلم به المطوّرون.

المُنغلِق هو دالة تتذكّر متغيراتها الخارجية كما ويمكنها أن تصل إليها. هذا الأمر -في بعض اللغات- مستحيل، أو أنّه يلزم كتابة الدالة بطريقة معيّنة ليحدث ذلك. ولكن كما شرحنا أعلاه ففي لغة JavaScript، كلّ الدوال مُنغلِقات بطبيعتها (وطبعًا ثمّة استثناء واحد أوحد نشرحه في فصل "صياغة الدالة الجديدة").

يعني ذلك بأنّ الدوال تتذكّر أين أُنشئت باستعمال خاصية `[[Environment]]` المخفية، كما ويمكن للدوال كافة الوصول إلى متغيراتها الخارجية.

لو كنت عزيزي مطوّر الواجهات في مقابلة وأتاك السؤال "ما هو المُنغلِق؟" فيمكنك أن تقدّم تعريفه شرحًا، كما وتُضيف بأنّ الدوال في JavaScript كلّها مُنغلِقات، وربما شيء من عندك تفاصيل تقنية مثل خاصية `[[Environment]]` وطريقة عمل البيئات المُعجمية.

6.3.5 كُتل الشفرات والحلقات، تعابير الدوال الآنية

ركّزتُ في الأمثلة أعلاه على الدوال، إلا أنّ البيئة المُعجمية موجودة لكلّ كتلة شيفرات `{...}`. تُنشأ البيئة المُعجمية حين تعمل أيّ كتلة شيفرات فيها متغيرات تُعدّ محلية لهذه الكتلة. إليك بعض الأمثلة.

أ. الجملة الشرطية If

في المثال أسفله نرى المتغير `user` موجودًا فقط داخل كتلة `if`:

```
let phrase = "Hello";
if (true) {
    let user = "Ahmad";
    alert(`${phrase}, ${user}`); // Hello, Ahmad
}
alert(user); // خطأ! لا أرى هذا المتغير!
```

```
let phrase = "Hello";

if (true) {
  let user = "John";

  alert(`${phrase}, ${user}`);
}

alert(user); // Error, no such variable!
```

user: "John" ← الخارجية ← phrase: "Hello" ← الخارجية ← null

حين تصل عملية التنفيذ داخل كتلة `if` يُنشئ المحرك البيئة المُعجمية "فقط وفقط إذا كذا...".

لهذه البيئة إشارة إلى البيئة الخارجية، بهذا يمكن أن تجد المتغير `phrase`. ولكن على العكس فالمتغيرات وتعابير الدوال المصرَّح عنها داخل `if` في تلك البيئة المُعجمية لا يمكن أن نراها من الخارج.

فمثلًا بعدما تنتهي إفادة `if` لن يرى التابع `alert` أسفلها متغير `user`، وهذا سبب الخطأ.

ب. حلقة "كرّر طالما"

لكلّ دورة في حلقة التكرار بيئة مُعجمية خاصة بها. وأيضًا لو صرّحت عن متغير في `for(let ...)` فسيكون موجودًا فيها:

```
for (let i = 0; i < 10; i++) {
  // لكلّ دورة بيئة مُعجمية خاصة بها
  // {i: value}
}
```

```
alert(i); // خطأ، ما من متغير كهذا
```

لاحظ كيف أنّ الإفادة `let i` خارج كتلة `{...}` بصريًا. مُنشئ حلقة `for` خاص نوعًا ما: لكلّ دورة من الحلقة بيئة مُعجمية خاصة بها تحمل قيمة `i` الحالية فيها أيضًا. وكما مع `if` فبعد انتهاء الحلقة لا نرى `i` خارجها.

ج. كُتل الشفرات

يمكننا أيضًا استعمال كتلة شفرات `{…}` "مجرّدة" لنعزل المتغيرات في "نطاق محلي" خاص بها.

فمثلًا في متصفّح الوب تتشارك كل السكربتات (عدا التي فيها `type="module"`) نفس المساحة العمومية. لذا لو أنشأنا متغيرًا عموميًا في واحد من السكربتات يمكن أن تراه البقية. هذا الأمر يتسبب بمشكلة لو استعمل سكربتان اثنان نفس اسم المتغير وبدأ كلّ منهما بتعويض الذي عند الآخر.

يمكن أن يحدث هذا لو كان اسم المتغير كلمة شائعة (مثلًا `name`) ولا يدري مطورو السكربتات ما يفعله الغير. فيمكن أن نستعمل كتلة شيفرات لغول السكربت كاملًا أو جزءًا منه حتى لو أردنا تجنّب هذه المشكلة:

```
{
  // نُجري أمرًا على المتغيرات المحلية يُمنع على ما خارجنا رؤيته

  let message = "Hello";

  alert(message); // Hello
}

alert(message); // خطأ: message غير معرّف
```

لا ترى الشيفرات خارج تلك الكتلة (أو حتى الموجودة في سكربت آخر) المتغيرات داخل الكتلة إذ أنّ لها بيئتها المُعجمية الخاصة بها.

د. تعابير الدوال الآنية IIFE

سابقًا لم تكن هناك بيئات مُعجمية للكُتل في JavaScript. وكما "الحاجة أمّ الاختراع"، فكان على المطوّرين حلّ ذلك، وهكذا صنعوا ما سمّوه "تعابير الدوال آنيّة الاستدعاء" (Immediately-Invoked Function Expressions).

لست تريد أن تكتب هذا النوع من الدوال في وقتنا الآن، ولكن يمكن أن تصادفك وأنت تطالع السكربتات القديمة فالأفضل لو تعرف كيف تعمل من الآن.

إليك شكل الدوال الآنية هذه:

```
(function() {

  let message = "Hello";

  alert(message); // Hello

})();
```

هنا أنشأنا تعبير دالة واستدعيناه مباشرةً/آنيًا. لذا فتعمل الشيفرة في لحظتها كما وفيها متغيراتها الخاصة بها.

الطريقة هي أن نُحيط تعبير الدالة بأقواس `(function {...})` إذ أنّ محرّك JavaScript حين يقابل `"function"` في الشيفرة الأساسية يفهمها وكأنّها تصريح عن دالة. ولكن، التصريح عن الدوال يحتاج اسمًا لها، بهذا فهذه الشيفرة ستَسبّب بخطأ:

```
// نحاول التصريح عن الدالة واستدعائها آنيًا
function() { // <-- Error: Unexpected token (

  let message = "Hello";

  alert(message); // Hello

}();
```

حتّى لو قلنا "طيب لنضيف ذلك الاسم" فلن ينفع إذ أنّ محرّك JavaScript لا يسمح باستدعاء التصاريح عن الدوال آنيًا:

```
// خطأ صياغي بسبب الأقواس أسفله
function go() {

}(); // <-- لا يمكن أن نستدعي التصريح عن الدوال آنيًا
```

إذًا، فالأقواس حول الدالة ما هي إلا خدعة لنضحك على محرّك JavaScript ونُقنعه بأنّا أنشأنا الدالة في سياق تعبير آخر وبهذا فهي تعبير دالة... أي لا تحتاج اسمًا ويمكن استدعائها آنيًا.

ثمّة طرق أخرى دون الأقواس لنُقنع المحرّك بأنّ ما نعني هو تعبير الدالة:

```
// طرائق إنشاء هذه التعابير الآنية
```

```
(function() {
  alert("أقواس تحيط بالدالة");
}) ();

(function() {
  alert("أقواس تحيط بكامل الجملة");
}() );

! function() {
  alert("أوّل التعبير NOT عملية الأعداد الثنائية");
}();

+ function() {
  alert("عملية الجمع الأحادية أوّل التعبير");
}();
```

في كلّ الحالات أعلاه: صرّحنا عن تعبير دالة واستدعيناها آنيًا. لنوضّح ذلك ثانيةً: لم يعد هناك أيّ داعٍ لنكتب هكذا شيفرات في وقتنا الحاضر.

6.3.6 كنس المهملات

عادةً ما تُمسح وتُحذف البيئة المُعجمية بعدما تعمل الدالة. مثال:

```
function f() {
  let value1 = 123;
  let value2 = 456;
}

f();
```

هنا القيمتين (تقنيًا) خاصيتين للبيئة المُعجمية. ولكن حين تنتهي `f()` لا يمكن أن نصل إلى تلك البيئة بأيّ طريقة فتُحذف من الذاكرة.

لكن لو كانت هناك دالة متداخلة يمكن أن نصل إليها بعدما تنتهي `f` (ولديها خاصية `[[Environment]]` التي تُشير إلى البيئة المُعجمية الخارجية)، لو كانت فيمكن أن نصل إليها:

```
function f() {
  let value = 123;
```

```
  function g() { alert(value); }

  return g; // (*)
}

let func = f(); // g يمكن أن تصل func الآن بإشارة إلى
// بذلك تبقى في الذاكرة، ومعها بيئتها المُعجمية الخارجية
```

لاحظ بأنّه لو استدعينا `f()` أكثر من مرة، فسوف تُحفظ الدوال الناتجة منها وتبقى كائنات البيئة المُعجمية لكلّ واحدة منها في الذاكرة. إليك ثلاثة منها في الشيفرة أدناه:

```
function f() {
  let value = Math.random();

  return function() { alert(value); };
}
// في المصفوفة ثلاث دوال تُشير كلّ منها إلى البيئة المُعجمية
// في عملية التنفيذ f() المقابلة لكلّ واحدة
let arr = [f(), f(), f()];
```

يموت كائن البيئة المُعجمية حين لا يمكن أن يصل إليه شيء (كما الحال مع أيّ كائن آخر). بعبارة أخرى فهو موجود طالما ثمّة دالة متداخلة واحدة (على الأقل) في الشيفرة تُشير إليه.

في الشيفرة أسفله، بعدما تصير `g` محالة الوصول تُمسح بيئتها المُعجمية فيها (ومعها متغير `value`) من الذاكرة:

```
function f() {
  let value = 123;

  function g() { alert(value); }

  return g;
}

// طالما يمكن أن تصل func بإشارة إلى g، ستظلّ تشغل حيّزًا في الذاكرة
let func = f();
```

```
// ...والآن لم تعد كذلك ونكون قد نظّفنا الذاكرة
func = null;
```

ا. التحسينات على أرض الواقع

كما رأينا، فنظريًا طالما الدالة "حيّة تُرزق" تبقى معها كل متغيراتها الخارجية. ولكن عمليًا تُحاول محرّكات JavaScript تحسين أداء ذلك. فهي تحلّل استعمال المتغيرات فلو كان واضحًا لها في الشيفرة بأنّ المتغير الخارجي لم يعد مستعملًا، تحذفه.

ثمّة -في محرّك V8 (كروم وأوبرا)- تأثير مهمّ ألا وهو أنّ هذا المتغير لن يكون مُتاحًا أثناء التنقيح.

جرّب تشغيل المثال الآتي في "أدوات المطوّرين" داخل متصفّح كروم. ما إن يُلبث تنفيذ الشيفرة، اكتب `alert(value)` في الطرفية.

```
function f() {
  let value = Math.random();

  function g() {
    debugger; // اكتب في الطرفية: alert(value); ما من متغير كهذا!
  }

  return g;
}

let g = f();
g();
```

كما رأينا، ما من متغير كهذا! يُفترض نظريًا أن نصل إليه ولكنّ المحرّك حسّن أداء الشيفرة وحذفه.

يؤدّي ذلك أحيانًا إلى مشاكل مضحكة (هذا إن لم تجلس عليها اليوم بطوله لحلّها) أثناء التنقيح. إحدى هذه المشاكل هي أن نرى المتغير الخارجي بدل الذي توقّعنا أن نراه (يحمل كلاهما نفس الاسم):

```
let value = "Surprise!";

function f() {
  let value = "the closest value";

  function g() {
    debugger; // اكتب في المِعراض: alert(value); إليك Surprise!
```

```
    }

    return g;
  }

  let g = f();
  g();
```

من المفيد معرفة هذه الميزة في معيار V8. متى ما بدأت التنقيح في كروم أو أوبرا، فستراها شئت أم أبيت.

هذه ليست علّة في المنقّح بل هي ميزة خاصة في معيار V8. ربما تتغير لاحقًا من يدري. يمكنك أن تتحقّق منها متى أردت بتجربة الأمثلة في هذه الصفحة.

6.3.7 تمارين

ا. هل العدّادات مستقلة عن بعضها البعض؟

الأهمية: ★★★★★

صنعنا هنا عدّادين اثنين counter و counter2 باستعمال ذات الدالة makeCounter. هل هما مستقلان عن بعضهما البعض؟ ما الذي سيعرضه العدّاد الثاني؟ 0,1 أم 2,3 أم ماذا؟

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
let counter2 = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1

alert( counter2() ); // ؟
alert( counter2() ); // ؟
```

الحل:

الإجابة هي: 0,1.

صنعنا الدالتين `counter` و `counter2` باستدعاءين `makeCounter` مختلفين تمامًا. لذا فلكلّ منهما بيئات مُعجمية خارجية مستقلة عن بعضها، ولكلّ منهما متغير `count` مستقل عن الثاني.

ب. كائن عد

الأهمية: ★★★★★

هنا صنعنا كائن عدّ بمساعدة دالة بانية (Constructor Function).

هل ستعمل؟ ماذا سيظهر؟

```
function Counter() {
  let count = 0;

  this.up = function() {
    return ++count;
  };
  this.down = function() {
    return --count;
  };
}

let counter = new Counter();

alert( counter.up() ); // ؟
alert( counter.up() ); // ؟
alert( counter.down() ); // ؟
```

الحل:

طبعًا، ستعمل كما يجب.

صُنعت الدالتين المتداخلتين في نفس البيئة المُعجمية الخارجية، بهذا تتشاركان نفس المتغير `count` وتصلان إليه:

```
function Counter() {
  let count = 0;
```

```
  this.up = function() {
    return ++count;
  };

  this.down = function() {
    return --count;
  };
}

let counter = new Counter();

alert( counter.up() ); // 1
alert( counter.up() ); // 2
alert( counter.down() ); // 1
```

ج. دالة في شرط if

طالِع الشيفرة أسفله. ما ناتج الاستدعاء في آخر سطر؟

```
let phrase = "Hello";

if (true) {
  let user = "Ahmad";

  function sayHi() {
    alert(`${phrase}, ${user}`);
  }
}

sayHi();
```

الحل:

الناتج هو: خطأ.

صُرّح عن الدالة `sayHi` داخل الشرط `if` وتعيش فيه فقط لا غير. ما من دالة `sayHi` خارجية.

د. المجموع باستعمال المُنغلِقات

الأهمية: ★★★★☆

اكتب الدالة `sum` لتعمل هكذا: `sum(a)(b) = a+b`.

نعم عينك سليمة، هكذا تمامًا باستعمال قوسين اثنين (ليست خطأً مطبعيًا).

مثال:

```
sum(1)(2) = 3
sum(5)(-1) = 4
```

الحل:

ليعمل القوسين الثانيين، يجب أن يُعيد الأوليين دالة.

هكذا:

```
function sum(a) {

  return function(b) {
    return a + b; // تأخذ "a" من البيئة المُعجمية الخارجية
  };

}

alert( sum(1)(2) ); // 3
alert( sum(5)(-1) ); // 4
```

ه. الترشيح عبر دالة

الأهمية: ★★★★★

نعلم بوجود التابع `arr.filter(f)` للمصفوفات. ووظيفته هي ترشيح كلّ العناصر عبر الدالة `f`. لو أرجعت `true` فيُعيد التابع العنصر في المصفوفة الناتجة.

اصنع مجموعة مرشّحات "جاهزة لنستعملها مباشرة":

- `inBetween(a, b)`: بين `a` و`b` بما فيه الطرفين (أي باحتساب `a` و`b`).
- `inArray([...])`: في المصفوفة الممرّرة.

هكذا يكون استعمالها:

- `arr.filter(inBetween(3,6))`: تحدّد القيم بين 3 و6 فقط.
- `arr.filter(inArray([1,2,3]))`: تحدّد العناصر المتطابقة مع أحد عناصر `[1,2,3]` فقط.

مثال:

```
// .. inArray و inBetween شيفرة الدالتين
let arr = [1, 2, 3, 4, 5, 6, 7];

alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6

alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

(تجربة حية للتمرين)

الحل:

- المرشّح `inBetween`

```
function inBetween(a, b) {
  return function(x) {
    return x >= a && x <= b;
  };
}

let arr = [1, 2, 3, 4, 5, 6, 7];
alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6
```

- المرشّح `inArray`

```
function inArray(arr) {
  return function(x) {
    return arr.includes(x);
  };
}

let arr = [1, 2, 3, 4, 5, 6, 7];
alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

(تجربة حية للحل)

و. الترشيح حسب حقل الاستمارة

الأهمية: ★★★★★

أمامنا مصفوفة كائنات علينا ترتيبها:

```
let users = [
  { name: "Ahmad", age: 20, surname: "Ahmadson" },
  { name: "Pete", age: 18, surname: "Peterson" },
  { name: "Ann", age: 19, surname: "Hathaway" }
];
```

الطريقة الطبيعية هي الآتي:

```
// (Ann, Ahmad, Pete) حسب الاسم
users.sort((a, b) => a.name > b.name ? 1 : -1);

// (Pete, Ann, Ahmad) حسب العمر
users.sort((a, b) => a.age > b.age ? 1 : -1);
```

هل يمكن أن تكون بحروف أقل، هكذا مثلًا؟

```
users.sort(byField('name'));
users.sort(byField('age'));
```

أي، بدل أن نكتب دالة، نضع `byField(fieldName)` فقط.

اكتب الدالة `byField` لنستعملها هكذا.

(تجربة حية للتمرين)

الحل:

```
let users = [
  { name: "Ahmad", age: 20, surname: "Ahmadson" },
  { name: "Pete", age: 18, surname: "Peterson" },
  { name: "Ann", age: 19, surname: "Hathaway" }
];

function byField(field) {
  return (a, b) => a[field] > b[field] ? 1 : -1;
```

```
}

users.sort(byField('name'));
users.forEach(user => alert(user.name)); // Ann, Ahmad, Pete

users.sort(byField('age'));
users.forEach(user => alert(user.name)); // Pete, Ann, Ahmad
```

(تجربة حية للحل)

ز. جيش عرمرم من الدوال

الأهمية: ★★★★★

تصنع الشيفرة الآتية مصفوفة من مُطلقي النار `shooters`.

يفترض أن تكتب لنا كلّ دالة رقم هويّتها، ولكن ثمّة خطب فيها...

```
function makeArmy() {
  let shooters = [];

  let i = 0;
  while (i < 10) {
    let shooter = function() { // دالة مُطلق النار
      alert( i ); // المفترض أن ترينا رقمها
    };
    shooters.push(shooter);
    i++;
  }

  return shooters;
}

let army = makeArmy();

army[0](); // مُطلق النار بالهويّة 0 يقول أنّه 10
army[5](); // مُطلق النار بالهويّة 5 يقول أنّه 10...
// ... كلّ مُطلقي النار يقولون 10 بدل هويّاتهم 0 فَـ 1 فَـ 2 فَـ 3 ...
```

لماذا هويّة كلّ مُطلق نار نفس البقية؟ أصلِح الشيفرة لتعمل كما ينبغي أن تعمل.

(تجربة حية للتمرين)

الحل:

لنُجري مسحًا شاملًا على ما يجري في `makeArray`، حينها يظهر لنا الحل جليًا.

- تُنشئ مصفوفة `shooters` فارغة:

```
let shooters = [];
```

- تملأ المصفوفة في حلقة عبر `shooters.push(function...)`.

كلّ عنصر هو دالة، بهذا تكون المصفوفة الناتجة هكذا:

```
shooters = [
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); }
];
```

- تُعيد الدالة المصفوفة.

لاحقًا، يستلم استدعاء `army[5]()` العنصر `army[5]` من المصفوفة، وهي دالة فيستدعيها.

الآن، لماذا تعرض كلّ هذه الدوال نفس الناتج؟

يعزو ذلك إلى عدم وجود أيّ متغير محلي باسم `i` في دوال `shooter`. فحين تُستدعى هذه الدالة تأخذ المتغير `i` من البيئة المُعجمية الخارجية.

وماذا ستكون قيمة `i`؟

لو رأينا مصدر القيمة:

```
function makeArmy() {
  ...
```

```
    let i = 0;
    while (i < 10) {
      let shooter = function() { // دالة مُطلق النار
        alert( i ); // المفترض أن ترينا رقمها
      };
      ...
    }
    ...
}
```

كما نرى... "تعيش" القيمة في البيئة المُعجمية المرتبطة بدورة `makeArmy()` الحالية. ولكن متى استدعينا `army[5]()`، تكون دالة `makeArmy` قد أنهت مهمّتها فعلًا وقيمة `i` هي آخر قيمة، أي `10` (قيمة نهاية حلقة `while`).

وبهذا تأخذ كلّ دوال `shooter` القيمة من البيئة المُعجمية الخارجية، ذات القيمة الأخيرة `i=10`.

يمكن أن نُصلح ذلك بنقل تعريف المتغير إلى داخل الحلقة:

```
function makeArmy() {

  let shooters = [];

  // (*)
  for(let i = 0; i < 10; i++) {
    let shooter = function() { // دالة مُطلق النار
      alert( i ); // المفترض أن ترينا رقمها
    };
    shooters.push(shooter);
  }

  return shooters;
}

let army = makeArmy();

army[0](); // 0
army[5](); // 5
```

الآن صارت تعمل كما يجب إذ في كلّ مرة تُنفّذ كتلة الشيفرة في `for (let i=0...) {...}`، يُنشئ المحرّك بيئة مُعجمية جديدة لها فيها متغير `i` المناسب لتلك الكتلة.

إذًا لنلخّص: قيمة `i` صارت "تعيش" أقرب للدالة من السابق. لم تعد في بيئة `makeArmy()` المُعجمية بل الآن في تلك البيئة المخصّصة لدورة الحلقة الحالية. هكذا صارت تعمل كما يجب.

```
shooters = [
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  ...
  function () { alert(i); }
];
```


أعدنا كتابة الشيفرة هنا وعوّضنا `while` بحلقة `for`.

يمكننا أيضًا تنفيذ حيلة أخرى. لنراها لنفهم الموضوع أكثر:

```
function makeArmy() {
  let shooters = [];

  let i = 0;
  while (i < 10) {
    let j = i; // (*)
    let shooter = function() { // دالة مُطلق النار
      alert( j ); // (*) المفترض أن ترينا رقمها
    };
    shooters.push(shooter);
    i++;
  }

  return shooters;
}

let army = makeArmy();

army[0](); // 0
army[5](); // 5
```

كما حلقة `for`، فحلقة `while` تصنع بيئة مُعجمية جديدة لكلّ دورة، وهكذا نتأكّد بأن تكون قيمة `shooter` صحيحة.

باختصار ننسخ القيمة `let j = i` وهذا يصنع المتغير `j` المحلي داخل الحلقة وينسخ قيمة `i` إلى نفسه. تُنسخ الأنواع الأولية "حسب قيمتها" By value، لذا بهذا نأخذ نسخة كاملة مستقلة تمامًا عن `i`، ولكنّها مرتبطة بالدورة الحالية في الحلقة.

(تجربة حية للحل)

6.4 إفادة "var" القديمة

ذكرنا في أوائل الفصول حين تكلمنا عن المتغيرات - ذكرنا ثلاث طرائق للتصريح عنها:

1. `let`
2. `const`
3. `var`

تتصرّف كلا الإفادتين `let` و`const` بذات الطريقة (بالمقايسة مع البيئات المُعجمية).

بينما `var` فهو وحش آخر مختلف جذريًا ويعود في أصله إلى قرون سحيقة. لا نستعمله عادةً في السكربتات الحديثة ولكنّك ستجده حتمًا خلف إحدى صخور السكربتات القديمة.

لو لم ترغب بالتعرّف على هذه السكربتات فيمكنك تخطّي هذا الفصل أو تأجيله لوقت لاحق. ولكن لا تنسَ احتمالية ندمك لاحقًا فيغدر بك هذا الوحش.

من أول وهلة نرى بأنّ تصرّف `var` يشابه تصرّف `let`، أي أنّه يُصرّح (مثل الثاني) عن متغير:

```
function sayHi() {
  var phrase = "Hello"; // "let" بدل "var" متغير محلي، استعملنا

  alert(phrase); // Hello
}

sayHi();

alert(phrase); // خطأ، phrase غير معرَف
```

ولكن... ما خفي كان أعظم. إليك الفروق.

6.4.1 ليس لإفادة "var" نطاقًا كُتليًا

حين نصرّح عن المتغيرات باستعمال `var` نكون جعلناها معروفة للدالة كاملةً (لو كانت في دالة) أو عمومية في السكربت. يمكنك أن ترى تلك المتغيرات إن اخترقت "جدران" الكُتل.

مثال:

```
if (true) {
  var test = true; // "var" بدل "let" نستعمل
}

alert(test); // الناتج true، أي أنّ المتغير "حيّ يُرزق" بعد إفادة if
```

تجاهل var كتل الشيفرة، وبهذا صار متغير test عموميًا.

لو استعملنا test let بدل test var فسيكون المتغير ظاهرًا لباقي الشيفرة داخل إفادة if فقط لا غير:

```
if (true) {
  let test = true; // "let" نستعمل
}

alert(test); // خطأ: لم يُعرّف عن test
```

يسري الأمر ذاته على الحلقات فلا يمكن أن يكون var محليًا حسب الكتلة أو حسب الحلقة:

```
for (var i = 0; i < 10; i++) {
  // ...
}

alert(i); // 10، ظهر "i" بعد الحلقة فهو متغير عمومي
```

لو كتبت كتلة شيفرة في دالة فسيصير var متغيرًا على مستوى الدالة كاملةً.

```
function sayHi() {
  if (true) {
    var phrase = "Hello";
  }

  alert(phrase); // يمكننا فعل هذا
}

sayHi();
alert(phrase); // خطأ: phrase غير معرّف (طالع مِعراض المطوّر)
```

كما نرى فإفادة `var` تخترق كُتل `if` و`for` وغيرها من كُتل شيفرة. يعزو ذلك إلى أنّه في الزمن الماضي الجميل لم تكن لكُتل جافاسكربت بيئات مُعجمية. و`var` إحدى آثار ذلك الزمن.

6.4.2 تعالج التصريحات باستعمال var عند بدء الدالة

تُعالج التصريحات باستعمال `var` متى ما بدأت الدالة (أو بدأ السكربت، للمتغيرات العمومية).

أي أنّ متغيرات `var` تُعرّف من بداية الدالة مهما كان مكان تعريفها (هذا لو لم يكن التعريف في دالة متداخلة أخرى).

يعني ذلك أنّ هذه الشيفرة:

```
function sayHi() {
  phrase = "Hello";

  alert(phrase);

  var phrase;
}
sayHi();
```

متطابقة تقنيًا مع هذه (بتحريك `var phrase` إلى أعلى):

```
function sayHi() {
  var phrase;

  phrase = "Hello";

  alert(phrase);
}
sayHi();
```

أو حتى هذه (لا تنسَ بأنّ كُتل الشيفرات مُهملة):

```
function sayHi() {
  phrase = "Hello"; // (*)
  if (false) {
    var phrase;
  }
```

```
    alert(phrase);
  }
  sayHi();
```

يدعو الناس هذا السلوك بسلوك "الطفو" hoisting (أو الرفع) إذ أنّ متغيرات `var` "تطفو" إلى أعلى الدالة (أو ترتفع إلى أعلاها).

أي أنّه في المثال أعلاه، الفرع `if (false)` من الإفادة لا يعمل قط ولكن هذا ليس بمهم، إذ أنّ `var` داخله سيُعالج في بداية الدالة، وحين تصل عملية التنفيذ إلى `(*)` سيكون المتغير موجودًا لا محالة.

التصريحات تطفو صحيح، ولكنّ ليس عبارات الإسناد.

الأفضل لو نمثّل ذلك في هذا المثال:

```
function sayHi() {
  alert(phrase);

  var phrase = "Hello";
}

sayHi();
```

في السطر `var phrase = "Hello"` إجراءان اثنان:

1. التصريح عن المتغير باستعمال `var`
2. إسناد قيمة للمتغير باستعمال `=`.

يتعامل المحرّك مع التصريحات متى بدء تنفيذ الدالة (إذ التصريحات تطفو)، ولكنّ عبارة الإسناد لا تعمل إلّا حيثما ظهرت، فقط. إذًا فالشيفرة تعمل بهذا النحو فعليًا:

```
function sayHi() {
  var phrase; // بادئ ذي بدء، يعمل التصريح...

  alert(phrase); // غير معرّف

  phrase = "Hello"; // ...هنا.
}
sayHi();
```

يُعالج المحرّك التصريحات `var` حين تبدأ الدوال، وبهذا يمكننا الإشارة إليها أينما أردنا في الشيفرة. ولكن انتبه فالمتغيرات غير معرّفة حتى تُسند إليها قيم.

في الأمثلة أعلاه عمل التابع `alert` دون أيّ أخطاء إذ أن المتغير `phrase` موجود. ولكن لم تُسند فيه قيمة بعد فعرض `undefined`.

6.4.3 الخلاصة

هناك فرقين جوهرين بين `var` موازنةً بِـ `let/const`:

1. ليس لمتغيرات `var` نطاقًا كتليًا وأصغر نطاق لها هو في الدوال.
2. تُعالج التصريحات باستعمال `var` عند بدء الدالة (أو بدء السكربت، للمتغيرات العمومية).

هناك فرق آخر صغير يتعلّق بالكائن العمومي وسنشرحه في الفصل التالي.

بهذا، غالبًا ما يكون استعمال `var` أسوأ بكثير من `let` بعدما عرفت الفروق بينها، فالمتغيرات على مستوى الكُتل أمر رائع جدًا ولهذا السبب تمامًا أُضيفت `let` إلى معيار اللغة منذ زمن وصارت الآن الطريقة الأساسية (هي و`const`) للتصريح عن متغير.

6.5 الكائن العمومي Global object

تقدّم الكائنات العمومية متغيراتَ ودوال يمكن استعمالها من أي مكان. هذه الكائنات مضمّنة في بنية اللغة أو البيئة مبدئيًا. في المتصفّحات تُدعى بالنافذة `window` وفي Node.js تُدعى بالعموميات `global` وفي باقي البيئات تُدعى بأيّ اسم مناسب يراه مطوّروها.

أُضيف حديثًا الكائن `globalThis` إلى اللغة ليكون اسم قياسيًا للكائن العمومي على أن تدعمه كلّ البيئات. ولكن بعض المتصفّحات (وبالخصوص عدا Chromium Edge) لا تدعم هذا الكائن بعد، ولكن يمكنك "ترقيعه تعدّديًا" بسهولة تامة.

سنستعمل هنا `window` على فرضية بأنّ البيئة هي المتصفّح نفسه. لو كنت ستشغّل السكربت الذي تكتبه في بيئات أخرى فربما تستعمل `globalThis` بدل النافذة تلك.

يمكننا طبعًا الوصول إلى كافة خصائص الكائن العمومي مباشرةً:

```
alert("Hello");
// تتطابق تمامًا مع
window.alert("Hello");
```

يمكنك في المتصفّحات التصريح عن الدوال العمومية والمتغيرات باستعمال `var` (وليس `let/const`) لتصير خاصيات للكائن العمومي:

```
var gVar = 5;
alert(window.gVar); // 5 (تصير خاصية من خاصيات الكائن العمومي)
```

ولكن أرجوك ألا تعتمد على هذا الأمر! هذا السلوك موجود للتوافقية لا غير. تستعمل السكربتات الحديثة "وحداتَ جافاسكربت" (نشرحها في وقت لاحق) حيث لا يحدث هكذا أمر.

لن يحدث هذا لو استعملنا `let` هنا:

```
let gLet = 5;

alert(window.gLet); // غير معرّف (لا تصير خاصية للكائن العمومي)
```

لو كانت القيمة هامّة جدًا جدًا وأردت أن تدخل عليها من أيّ مكان عمومي فاكتبها على أنّها خاصية مباشرةً:

```
// نجعل من معلومات المستخدم الحالي عمومية لتصل إليها كلّ السكربتات
window.currentUser = {
  name: "Ahmad"
```

```
};

// وفي مكان آخر يريدها أحد
alert(currentUser.name);  // Ahmad

// أو (لو كان هناك المتغير المحلي ذا الاسم "currentUser"
// فنأخذها جهارةً من النافذة (وهذا آمن!)
alert(window.currentUser.name); // Ahmad
```

نختم هنا بأنّ استعمال المتغيرات العمومية غير محبّذ بالمرة ويجب أن يكون عددها بأقل ما يمكن. يُعدّ مبدأ تصميم الشيفرات حين تأخذ الدالة المتغيرات "الداخلة" وتُعطينا "نواتج" معيّنة - يُعدّ هذا المبدأ أفضل وأقلّ عُرضة للأخطاء وأسهل للاختبار موازنةً بالمتغيرات الخارجية أو العمومية.

6.5.1 استعمالها للترقيع تعدديًا

المجال الذي نستعمل الكائنات العمومية فيه هو اختبار لو كانت البيئة تدعم مزايا اللغة الحديثة.

فمثلًا يمكننا اختبار لو كانت كائنات الوعود `Promise` المضمّنة في اللغة مضمّنة حقًا (لم تكن كذلك في المتصفحات العتيقة):

```
if (!window.Promise) {
  alert("Your browser is really old!"); // تستعمل يا صاح متصفّحًا من زمن الطيبين!
}
```

لو لم نجد هذه الكائنات (مثلًا نستعمل متصفّحًا قديمًا) فيمكننا "ترقيعه تعدّديًا": أي إضافة الدوال التي لا تدعمها البيئة بينما هي موجودة في معيار اللغة الحديث.

```
if (!window.Promise) {
  window.Promise = ... // شيفرة نكتبها بنفسنا تؤدّي الميزة الحديثة في اللغة هذه
}
```

6.5.2 الخلاصة

- يحمل الكائن العمومي تلك المتغيرات التي يلزم أن نصل إليها أينما كنّا في الشيفرة.
- تشمل المتغيرات هذه كل ما هو مضمّن في بنية لغة جافاسكربت مثل المصفوفات `Array` والقيم المخصّصة للبيئة مثل `window.innerHeight` (ارتفاع نافذة المتصفّح).
- للكائن العمومي اسم عام في المواصفة: `globalThis`.

- ولكن... دومًا ما نُشير إليه بالأسماء "الأثرية" حسب كل بيئة مثل `window` (في المتصفحات) و`global` (في Node.js)، إذ أنّ `globalThis` هو مُقترح جديد على اللغة وليس مدعومًا في المتصفّحات عدة Chromium Edge (ولكن يمكننا ترقيعه تعدّديًا).
- علينا ألا نخزّن القيم في الكائن العمومي إلّا لو كانت حقًا وفعلًا عمومية للمشروع الذي نعمل عليه. كما ويجب أن يبقى عددها بأقل ما يمكن.
- حين نطوّر لاستعمال الشيفرات في المتصفّحات (لو لم نستعمل الوحدات)، تصير الدوال العمومية والمتغيرات باستعمال `var` خاصيات للكائن العمومي.

علينا استعمال خاصيات الكائن العمومي مباشرةً (مثل `window.x`) لتكون الشيفرة سهلة الصيانة مستقبلًا وأسهل فهمًا.

6.6 كائنات الدوال Function object وتعابير الدوال المسمّاة NFE

كما نعلم فالدوال في لغة جافاسكربت تُعدّ قيمًا. ولكلّ قيمة في هذه اللغة نوع. ولكن ما نوع الدالة نفسها؟

تُعدّ الدوال كائنات في جافاسكربت. يمكننا تخيّل الدوال على أنّها "كائنات إجرائية" يمكن استدعائها. لا يتوقّف الأمر عند الاستدعاء أيضًا بل يمكن حتّى أن نُعاملها معاملة الكائنات فنُضيف الخاصيات ونُزيلها، أو نمرّرها بالإشارة وغيرها من أمور.

6.6.1 خاصية الاسم name

تحتوي كائنات الدوال على خاصيات يمكننا استعمالها، فمثلًا يمكن أن نعرف اسم الدالة من خاصية الاسم name:

```
function sayHi() {
  alert("Hi");
}

alert(sayHi.name); // sayHi
```

والمُضحك في الأمر هو أنّ منطق اللغة في إسناد الاسم ذكيّ (الذكاء الاصطناعي مبهر)، فهو يُسند اسم الدالة الصحيح حتّى لو أنشأناها بدون اسم ثمّ أسندناها إلى متغير مباشرةً:

```
let sayHi = function() {
  alert("Hi");
};

alert(sayHi.name); // (للدالة اسم!) sayHi
```

كما ويعمل المنطق أيضًا لو كانت عملية الإسناد عبر قيمة مبدئية:

```
function f(sayHi = function() {}) {
  alert(sayHi.name); // (تعمل أيضًا!) sayHi
}

f();
```

تُدعى هذه الميزة في توصيف اللغة "بالاسم السياقي". فلو لم تقدّم الدالة اسمًا لها فيحاول المحرّك معرفته من السياق مع أوّل عملية إسناد.

كما ولتوابِع الكائنات أسماء أيضًا:

```
let user = {

  sayHi() {
    // ...
  },

  sayBye: function() {
    // ...
  }
}

alert(user.sayHi.name); // sayHi
alert(user.sayBye.name); // sayBye
```

ولكن ليس للسحر مكان هنا، فهناك حالات يستحيل على المحرّك معرفة الاسم الصحيح منها، بهذا تكون خاصية الاسم فارغة، كما في هذه الشيفرة:

```
// نُنشئ دالة في مصفوفة
let arr = [function() {}];

alert( arr[0].name ); // <سلسلة نصية فارغة>
// ما من طريقة يعرف بها المحرّك الاسم الصحيح، فباختصار، ليس هناك اسم!
```

ولكن عمليًا، لكل الدوال أسماء أغلب الوقت.

6.6.2 خاصية الطول "length"

توجد خاصية أخرى مضمّنة في اللغة باسم `length` وهي تُعيد عدد مُعاملات الدالة. مثال:

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2
```

نرى هنا بأن المُعاملات البقية لم تُحسب.

يستعمل المطوّرون خاصية `length` أحيانًا لإجراء التحقّق الداخلي داخل الدوال التي تعتمد في تشغيلها على التحكّم بدوال أخرى. في الشيفرة أسفله، تقبل دالة `ask` سؤالًا `question` تطرحه وعددًا غير محدّد من دوال المعالجة `handler` لتستدعيها دالة السؤال مع الإجابة التي يعطيها المستخدم. يمكننا تمرير نوعين اثنين من المُعالجات هذه:

- دالة ليس لها وُسطاء لا تُنشأ إلا عندما يُعطيها المستخدم إجابة بالإيجاب.
- دالة لها وُسطاء تُستدعى في بقية الحالات وتُعيد إجابة المستخدم.

علينا فحص خاصية `handler.length` لنستدعي `handler` بالطريقة السليمة. الفكرة هنا هي أن تستعمل الشيفرة صياغة مُعالجة بسيطة وبدون وُسطاء لحالات الإيجاب (وهذا الشائع)، إضافةً على دعم المُعالجات العامة أيضًا:

```
function ask(question, ...handlers) {
  let isYes = confirm(question);

  for(let handler of handlers) {
    if (handler.length == 0) {
      if (isYes) handler();
    } else {
      handler(isYes);
    }
  }

}

// يُعاد كِلا المُعالجان لو كانت الإجابة بالإيجاب
// ولو كانت بالسلب، فالثاني فقط
ask("Question?", () => alert('You said yes'), result =>
alert(result));
```

هذه حالة من حالات التعدّدية الشكلية، أي حين يتغيّر تعاملنا مع الوُسطاء حسب أنواعها... في حالتنا فهي حسب أطوالها `length`. لهذه الفكرة استعمال فعليّ في مكتبات جافاسكربت.

6.6.3 خاصيات مخصّصة

يمكننا أيضًا إضافة ما نريد من خاصيات. فهنا نُضيف خاصية العدّاد counter ليسجّل إجمالي عدد الاستدعاءات:

```
function sayHi() {
  alert("Hi");

  // لنعدّ كم من مرّة شغّلناه
  sayHi.counter++;
}
sayHi.counter = 0; // القيمة الأولية

sayHi(); // Hi
sayHi(); // Hi

alert( `Called ${sayHi.counter} times` ); // Called 2 times
```

الخاصيات ليست متغيرات

لا تعرّف الخاصية المُسندة إلى الدوال مثل sayHi.counter = 0 متغيرًا محليًا فيها (counter في حالتنا). أي أنّ لا علاقة تربط الخاصية counter بالمتغير let counter البتة.

يمكننا التعامل مع الدوال على أنها كائنات فنخزّن فيها الخاصيات، ولكن هذا لا يؤثّر على طريقة تنفيذها. ليست المتغيرات خاصيات للدالة ولا العكس، كلاهما منفصلين يسيران في خطّين مُحال أن يتقاطعا.

يمكننا أحيانًا استعمال خاصيات الدوال بدل المُنغلِقات. فمثلًا يمكن إعادة كتابة تمرين دالة العدّ في فصل "المُنغلِقات" فنستعمل خاصية دالة:

```
function makeCounter() {
  // بدل:
  // let count = 0

  function counter() {
    return counter.count++;
  };

  counter.count = 0;

```

```
    return counter;
  }

  let counter = makeCounter();
  alert( counter() ); // 0
  alert( counter() ); // 1
```

هكذا خزّنا الخاصية count في الدالة مباشرةً وليس في بيئتها المُعجمية الخارجية.

أهذه أفضل أم المنغلقات أفضل؟ الفرق الرئيس هو: لو كانت قيمة count "تحيا" في متغير خارجي فلا يمكن لأي شيفرة خارجية الوصول إليها، بل الدوال المتداخلة فقط من يمكنها تعديلها، ولو ربطناها بدالة فيصير هذا ممكنًا:

```
function makeCounter() {

  function counter() {
    return counter.count++;
  };

  counter.count = 0;

  return counter;
}

let counter = makeCounter();

// هذا
counter.count = 10;
alert( counter() ); // 10
```

إذًا فالخيار يعود لنا: ماذا نريد وما الغاية.

6.6.4 تعابير الدوال المسماة

كما اسمها، فتعابير الدوال المسمّاة (Named Function Expression) هي تعابير الدوال التي لها اسم... بسيطة. لنأخذ مثلًا تعبير دالة مثل أي تعبير تراه في حياتك البرمجية التعيسة:

```
let sayHi = function(who) {
  alert(`Hello, ${who}`);
};
```

والآن نُضيف اسمًا له:

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};
```

هل حللنا أزمة عالمية هنا؟ ما الداعي من هذا الاسم `"func"`؟ أولًا، ما زال أمامنا تعبير دالة، فإضافة الاسم `"func"` بعد `function` لم يجعل الجملة تصريحًا عن دالة إذ ما زلنا نصنع الدالة داخل جزء من تعبير إسناد.

كما وإضافة الاسم لم يُعطب الدالة بأي شكل. يمكن أن ندخل الدالة هكذا `sayHi()`:

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};

sayHi("Ahmad"); // Hello, Ahmad
```

ثمّة أمرين مميزين بهذا الاسم `func` وهما السبب وراء كل هذا:

1. يتيح الاسم بأن تُشير الدالة إلى نفسها داخليًا.
2. ولا يظهر الاسم لما خارج الدالة.

فمثلًا تستدعي الدالة `sayHi` أسفله نفسها ثانيةً بالوسيط `"Guest"` لو لم نمرّر لها `who` من البداية:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // نستعمل func لنستدعي نفسنا ثانيةً
  }
};

sayHi(); // Hello, Guest

// ولكن هذا لن يعمل:
```

```
func(); // ويعطينا خطأً بأنّ func غير معرّفة (فالدالة لا تظهر لما خارجها)
```

ولكن لمَ نستعمل func أصلًا؟ ألا يمكن أن نستعمل sayHi لذلك الاستدعاء المتداخل؟ للصراحة، يمكن ذلك في حالات عديدة:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest");
  }
};
```

مشكلة تلك الشيفرة هي احتمالية تغيّر sayHi في الشيفرة الخارجية. فلو أُسندت الدالة إلى متغير آخر بدل ذاك فستبدأ الأخطاء تظهر:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest"); // خطأ: sayHi ليست بدالة
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // خطأ: لم يعد الاستدعاء المتداخل sayHi يعمل بعد الآن!
```

سبب ذلك هو أنّ الدالة تأخذ sayHi من بيئتها المُعجمية الخارجية إذ لا تجد sayHi محليًا فيها فتستعمل المتغير الخارجي. وفي لحظة الاستدعاء تلك يكون sayHi الخارجي قد صار null.

وهذا الداعي من ذلك الاسم الذي نضعه في تعبير الدالة، أن يحلّ هذه المشاكل بذاتها.

هيا نُصلح شيفرتنا:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
```

```
    } else {
      func("Guest"); // الآن تعمل كما يجب
    }
  };

  let welcome = sayHi;
  sayHi = null;

  welcome(); // Hello, Guest (الاستدعاءات المتداخلة تعمل)
```

الآن صارت تعمل إذ الاسم "func" محليّ للدالة فقط ولا تأخذها من الخارج (ولا تظهر للخارج أيضًا). تضمن لنا مواصفات اللغة بأنّها ستُشير دومًا وأبدًا إلى الدالة الحالية.

مع ذلك فما زالت الشيفرة الخارجية تملك المتغيرين sayHi وwelcome، بينما func هو "اسم الدالة داخليًا" أي كيف تستدعي الدالة نفسها من داخلها.

ما من ميزة كهذه للتصريح عن الدوال

ميزة "الاسم الداخلي" هذه التي شرحناها هنا مُتاحة لتعابير الدوال فقط وليست متاحة للتصريحات عن الدوال. فليس لهذه الأخيرة أية صياغة برمجية لإضافة اسم "داخلي" لها.

لكن أحيانًا نرى حاجة بوجود اسم داخلي نعتمد عليه، حينها يكون السبب وجيهًا بأن نُعيد كتابة التصريح عن الدالة إلى صيغة تعبير الدالة المسمّى.

6.6.5 الخلاصة

تُعدّ الدوال كائنات. شرحنا في الفصل خصائصها:

- اسمها name: غالبًا ما يأتي من تعريف الدالة. لكن لو لم يكن هناك واحد فيحاول المحرّك تخمينه من السياق (مثلًا من عبارة الإسناد).
- عدد مُعاملتها في تعريف الدالة length: لا تُحسب المُعاملات البقية.

لو عرّفنا الدالة باستعمال تعبير عن دالة (وليس في الشيفرة الأساس)، وكان لهذه الدالة اسم فنُسمّيها بتعبير الدالة المسمّى.

كما يمكن أن تحمل الدوال خاصيات إضافية، وتستغل الكثير من مكتبات جافاسكربت المعروفة هذه الميزة أيّما استغلال. إذ تُنشئ دالة "رئيسة" بعدها تُرفق دوال أخرى "مُساعِدة" إليها. فمثلًا تُنشئ مكتبة jQuery الدالة بالاسم $، وتُنشئ مكتبة lodash الدالة بالاسم _ ثمّ تُضيف خاصياتها _.clone و _.keyBy وغيرها (طالع

توثيقها متى أردت معرفتها أكثر). ما تفعله هذه الدوال يعود إلى أنّها (في الواقع) تحاول حدّ "التلوّث" في المجال العمومي فلا تستعمل المكتبة إلّا متغيرًا عموميًا واحدًا. وهذا يُقلّل من أدنى إمكانية لتضارب الأسماء.

إذًا، فالدالة تؤدي عملًا رائعًا كما هي، وأيضًا تحوي على وظائف أخرى خاصيات لها.

6.6.6 تمارين

ا. ضبط قيمة العدّاد وإنقاصها

الأهمية: ★★★★★

عدّل شيفرة الدالة `makeCounter()` بحيث يُنقص العدّاد قيمتها إضافةً إلى ضبطها:

- على `counter()` إعادة العدد التالي (كما في الأمثلة السابقة).
- على `counter.set(value)` ضبط قيمة العدّاد لتكون `value`.
- على `counter.decrease()` إنقاص قيمة العدّاد واحدًا (1).

طالِع الشيفرة أدناه كي تعرف طريقة استعمال الدالة:

```
function makeCounter() {
  let count = 0;

  // ... شيفرتك هنا ...
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1

counter.set(10); // نضبط العدد الجديد

alert( counter() ); // 10

counter.decrease(); // نُنقص العدد واحدًا 1

alert( counter() ); // 10 (بدل 11)
```

> يمكنك استعمال مُنغلِق أو خاصية دالة لحفظ العدد الحالي، أو لو أردت فاكتب الحل بالطريقتين.

الحل:

يستعمل الحل المتغير `count` محليًا، كما وتوابع أخرى نكتبها داخل الدالة `counter`. تتشارك هذه التوابع ذات البيئة المُعجمية الخارجية كما وترى أيضًا قيمة `count` الحالية.

ب. مجموع ما في الأقواس أيًا كان عدد الأقواس

الأهمية: ★★★★★

اكتب الدالة `sum` لتعمل كالآتي:

```
sum(1)(2) == 3; // 1 + 2
sum(1)(2)(3) == 6; // 1 + 2 + 3
sum(5)(-1)(2) == 6
sum(6)(-1)(-2)(-3) == 0
sum(0)(1)(2)(3)(4)(5) == 15
```

تريد تلميحًا؟ ربما تكتب كائنًا مخصّصًا يُحوّل الأنواع الأولية لتُناسب الدالة.

1. أيّما كانت الطريقة التي سنستعملها ليعمل هذا الشيء، فلا بدّ أن تُرجع `sum` دالة.
2. على تلك الدالة أن تحفظ القيمة الحالية بين كلّ استدعاء والآخر داخل الذاكرة.
3. حسب المهمّة المُعطاة، يجب أن تتحول الدالة إلى عدد حين نستعملها في `==` حيث الدوال كائنات لـذا فعملية التحويل ستنفع كما شرحنا في فصل "التحويل من كائن إلى قيمة أولية"، ويمكن أن نقدّم تابِعًـا خاصًا يُعيد ذلك العدد.

إلى الشيفرة:

```
function sum(a) {

  let currentSum = a;

  function f(b) {
    currentSum += b;
    return f;
  }

  f.toString = function() {
    return currentSum;
  };
```

```
  return f;
}

alert( sum(1)(2) ); // 3
alert( sum(5)(-1)(2) ); // 6
alert( sum(6)(-1)(-2)(-3) ); // 0
alert( sum(0)(1)(2)(3)(4)(5) ); // 15
```

لاحظ بأنّ دالة `sum` تعمل مرّة واحدة فقط لا غير، وتُعيد الدالة `f`. وبعدها في كلّ استدعاء يليها، تُضيف `f` المُعامل إلى المجموع `currentSum` وتُعيد نفسها. لا نستعمل التعاود في آخر سطر من `f`.

هذا شكل التعاود:

```
function f(b) {
  currentSum += b;
  return f(); // <-- استدعاء تعاودي
}
```

بينما في حالتنا نُعيد الدالة دون استدعائها:

```
function f(b) {
  currentSum += b;
  return f; // <-- لا تستدعي نفسها، بل تُعيد نفسها
}
```

وستُستعمل `f` هذه في الاستدعاء التالي، وتُعيد نفسها ثانيةً مهما لزم. وبعدها حين نستعمل العدد أو السلسلة النصية، يُعيد التابع `toString` المجموع `currentSum`. يمكن أيضًا أن نستعمل `Symbol.toPrimitive` أو `valueOf` لإجراء عملية التحويل.

6.7 صياغة "الدالة الجديدة" new Function

توجد (أيضًا) طريقة أخرى لإنشاء الدوال. صحيح هي نادرة الاستعمال ولكن لا مفرّ منها في حالات معيّنة.

6.7.1 الصياغة

إليك صياغة إنشاء الدالة:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

نصنع الدالة بالوُسطاء `arg1...argN` ونمرّر متنها `functionBody`.

"هات الشيفرة وقلّل ثرثرتك"... صحيح، هذا أسهل. إليك الدالة وفيها وسيطين اثنين:

```
let sum = new Function('a', 'b', 'return a + b');

alert( sum(1, 2) ); // 3
```

وهنا دالة بلا وُسطاء فيها متنها فقط:

```
let sayHi = new Function('alert("Hello")');

sayHi(); // Hello
```

الفرق الأساس بين هذه الطريقة والطرائق الأخرى هي أنّا نصنع الدالة هنا (كما لاحظت) من سلسلة نصية حرفيًا، ونمرّرها في وقت تشغيل الشيفرة.

ألزمتنا التصريحات السابقة كلها - ألزمتنا نحن المطوّرين أن نكتب شيفرة الدالة في السكربت. ولكن صياغة `new Function` تسمح لنا بأن نحوّل أيّ سلسلة نصية لتصير دالة. فمثلًا يمكن أن نستلم دالة جديدة من أحد الخوادم وننفّذها:

```
let str = ... نستلم الشيفرة ديناميكيًا من الخادم ...

let func = new Function(str);
func();
```

لا نستعمل هذه إلا في حالات خاصّة، مثل لو استلمنا الشيفرة من الخادم أو صنعنا الدالة ديناميكًا من قالب (في تطبيقات الوِب المعقّدة).

6.7.2 المُنغلِقات closures

عادةً ما تتذكّر الدالة مكان ولادتها في الخاصية المميّزة `[[Environment]]`، فتُشير إلى البيئة المُعجمية حين صُنعت الدالة (شرحنا هذا في فصل "المُنغلِقات"). ولكن حين نصنع الدالة باستعمال `new Function` فتُضبط خاصية `[[Environment]]` على البيئة المُعجمية العمومية لا الحالية.

أي أنّ هذه الدوال لا يمكن أن ترى المتغيرات الخارجية بل تلك العمومية فقط.

```
function getFunc() {
  let value = "test";

  let func = new Function('alert(value)');

  return func;
}

getFunc()(); // خطأ: value غير معرّف
```

وازن بين هذا والسلوك الطبيعي:

```
function getFunc() {
  let value = "test";

  let func = function() { alert(value); };

  return func;
}

getFunc()(); // "test"، من بيئة getFunc المُعجمية
```

صحيح أنّ الميزة الخاصة للصياغة `new Function` غريبة بعض الشيء، ولكنها عمليًا مفيدة جدًا.

تخيّل الآن بأنّنا صنعنا دالة من سلسلة نصية. شيفرة هذه الدالة ليست معروفة ونحن نكتب السكربت (ولهذا لم نستعمل الدوال العادية)، بل ستكون معروفة حين تنفيذه. كما أسلفنا يمكن أن نستلم الدالة من الخادم أو أيّ مكان آخر.

الآن، على دالتنا هذه التفاعل مع السكربت الرئيس. لكن ماذا لو أمكن لها أن ترى المتغيرات الخارجية؟

المشكلة هي أنّه قبل أن ننشر شيفرة جافاسكربت لنستعملها، نستعمل المُصغِّرات minifiers لضغطها. تقلّص هذه المُصغِّرات حجم الشيفرة بإزالة التعليقات والمسافات الزائدة، كما (وهذا مهم) تُغيّر تسمية المتغيرات المحلية إلى أسماء أقصر.

فمثلًا لو كان في الدالة `let userName` فيستبدلها المُصغِّر إلى `let a` (أو أيّ حرف آخر لو هناك من أخذ الاسم)، وينفّذ هذا في كلّ مكان آخر. عادةً لا يضرّ ذلك إذ أنّ المتغير محلي ولا يمكن لما خارج الدالة رؤيته، بينما يستبدل المُصغِّر كلّ مرة يرد فيها المتغير داخل الدالة. هذه الأدوات ذكية فهي تحلّل بنية الشيفرة لألا تُعطبها، وليست كأدوات البحث والاستبدال الهمجية.

لذا لو أرادت `new Function` أن تستعمل المتغيرات الخارجية فلن تعرف بوجود `userName` الذي تغيّر اسمه. لو أمكن للدوال `new Function` أن ترى المتغيرات الخارجية لكانت ستواجه مشاكل جمّة مع المُصغِّرات. كما وأنّ الشيفرات من هذا النوع ستكون سيّئة من حيث البنية وعُرضة للأخطاء والمشاكل. لو أردت تمرير شيء للدالة `new Function` فعليك استعمال مُعاملاتها.

6.7.3 الخلاصة

الصياغة:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

ويمكن تمرير المُعاملات (لأسباب تاريخية أيضًا) في قائمة مفصولة بفواصل.

هذه التصريحات الثلاث لا تختلف عن بعضها البعض:

```
new Function('a', 'b', 'return a + b'); // الصياغة الأساس
new Function('a,b', 'return a + b'); // مفصولة بفواصل
new Function('a , b', 'return a + b'); // مفصولة بفواصل ومسافات
```

تُشير خاصية `[[Environment]]` للدوال `new Function` إلى البيئة المُعجمية العمومية لا الخارجية. بهذا لا يمكن لهذه الدوال استعمال المتغيرات الخارجية. إلّا أنّ ذلك أمر طيّب إذ تؤمّن لنا خطّ حماية لألا نصنع الأخطاء والمشاكل، فتمرير المُعاملات جهارةً أفضل بكثير من حيث بنية الشيفرة ولا تتسبّب مشاكل مع المُصغِّرات.

6.8 الجدولة: المُهلة setTimeout والفترة setInterval

وأنت تكتب الشيفرة، ستقول في نفسك "أريد تشغيل هذه الدالة بعد قليل وليس الآن. هذا ما نسمّيه "بجدولة الاستدعاءات" "scheduling a call".

إليك تابِعين اثنين لهذه الجدولة:

- يتيح لك `setTimeout` تشغيل الدالة مرّة واحدة بعد فترة من الزمن.
- يتيح لك `setInterval` تشغيل الدالة تكراريًا يبدأ ذلك بعد فترة من الزمن ويتكرّر كلّ فترة حسب تلك الفترة التي حدّدتها.

صحيح أنّ هذين التابعين ليسا في مواصفة لغة جافاسكربت إلّا أنّ أغلب البيئات فيها مُجدوِل داخلي يقدّمهما لنا. وللدقّة، فكلّ المتصّفحات كما وNode.js تدعمهما.

6.8.1 تابِع تحديد المهلة setTimeout

الصياغة:

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

المُعاملات:

- `func|code`: ما يجب تنفيذه أكان دالة أو سلسلة نصية فيها شيفرة. عادةً هي دالة ولكن كعادة الأسباب التاريخية (أيضًا) يمكن تمرير سلسلة نصية فيها شيفرة، ولكنّ ذلك ليس بالأمر المستحسن.
- `delay`: التأخير قبل بدء التنفيذ بالمليثانية (1000 مليثانية = ثانية واحدة). مبدئيًا يساوي 0.
- `arg1`, `arg2`...: وُسطاء الدالة (ليست مدعومة في IE9-)

إليك هذه الشيفرة التي تستدعي `sayHi()` بعد ثانيةً واحدة:

```
function sayHi() {
  alert('Hello');
}
setTimeout(sayHi, 1000);
```

مع المُعاملات:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}
```

```
setTimeout(sayHi, 1000, "Hello", "Ahmad"); // Hello, Ahmad
```

لو كان المُعامل الأول سلسلة نصية فستصنع جافاسكربت دالة منها.

أي أنّ هذا سيعمل:

```
setTimeout("alert('Hello')", 1000);
```

ولكن استعمال السلاسل النصية غير مستحسن. استعمل الدوال السهمية بدلًا عنها:

```
setTimeout(() => alert('Hello'), 1000);
```

مرّر الدالة لكن لا تشغّلها

يُخطئ المبرمجون المبتدئون أحيانًا فيُضيفون أقواس `()` بعد الدالة:

```
// خطأ هذا!
setTimeout(sayHi(), 1000);
```

لن يعمل ذلك إذ يتوقّع `setTimeout` إشارة إلى الدالة، بينما هنا `sayHi()` يشغّل الدالة وناتج التنفيذ هو الذي يُمرّر إلى `setTimeout`. في حالتنا ناتج `sayHi()` ليس معرّفًا `undefined` (إذ لا تُعيد الدالة شيئًا)، ويعني ذلك أنّ عملنا ذهب سدًى ولم نُجدول أي شيء.

ا. الإلغاء باستعمال clearTimeout

نستلمُ حين نستدعي `setTimeout` "هويّةَ المؤقّت" `timerId` ويمكن استعمالها لإلغاء عملية التنفيذ.

صياغة الإلغاء:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

في الشيفرة أسفله نُجدول الدالة ثمّ نُلغيها (غيّرنا الخطّة العبقرية)، بهذا لا يحدث شيء:

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // هويّة المؤقّت

clearTimeout(timerId);
alert(timerId); // (لا تصير null بعد الإلغاء) ذات الهويّة
```

يمكن أن نرى من ناتج التابع `alert` أنّ هويّة المؤقّت (في المتصفّحات) هي عدد. يمكن أن تكون في البيئات الأخرى أيّ شيء آخر. فمثلًا في Node.js نستلم كائن مؤقّت فيه توابِع أخرى.

نُعيد بأن ليس هناك مواصفة عالمية متّفق عليها لهذه التوابع، فما من مشكلة في هذا.

يمكنك مراجعة مواصفة HTML5 للمؤقّتات (داخل المتصفّحات) في فصل المؤقّتات.

6.8.2 setInterval

صياغة التابع `setInterval` هي ذات `setTimeout`:

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

ولكلّ المُعاملات ذات المعنى. ولكن على العكس من `setTimeout` فهذا التابع يشغّل الدالة مرّة واحدة ثمّ أخرى وأخرى وأخرى تفصلها تلك الفترة المحدّدة.

يمكن أن نستدعي `clearInterval(timerId)` لنُوقف الاستدعاءات اللاحقة. سيعرض المثال الآتي الرسالة كلّ ثانيتين اثنتين، وبعد خمس ثوان يتوقّف ناتجها:

```
// نكرّر التنفيذ بفترة تساوي ثانيتين
let timerId = setInterval(() => alert('tick'), 2000);

// وبعد خمس ثوان نُوقف الجدولة
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

الوقت لا يتوقف حين تظهر alert مُنبثقة

تُواصل عقارب ساعة المؤقّت الداخلي (في أغلب المتصفّحات بما فيها كروم وفَيَرفُكس) بالمضيّ حتّى حين عرض `alert/confirm/prompt`، لذا متى ما شغّلت الشيفرة أعلاه ولم تصرف نافذة `alert` بسرعة، فسترى نافذة `alert` الثانية بعد ذلك مباشرةً، بذلك تكون الفترة الفعلية بين التنبيهين أقلّ من ثانيتين.

6.8.3 تداخل setTimeout

لو أردنا تشغيل أمر كلّ فترة، فهناك طريقتين اثنتين. الأولى هي `setInterval`. والثانية هي `setTimeout` متداخلة هكذا:

```
/** بدل كتابة:
let timerId = setInterval(() => alert('tick'), 2000);
*/
let timerId = setTimeout(function tick() {
  alert('tick');
  timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

تابع `setTimeout` أعلاه يُجدول الاستدعاء التالي ليحدث بعد نهاية الأول، لاحظ `(*)`.

كتابة توابع `setTimeout` متداخلة يعطينا شيفرة مطواعة أكثر من `setInterval`. بهذه الطريقة يمكن تغيير جدولة الاستدعاء التالي حسب ناتج الحالي.

فمثلًا علينا كتابة خدمة تُرسل طلب بيانات إلى الخادم كلّ خمس ثوان، ولكن لو كان الخادم مُثقلًا بالعمليات فيجب أن تزداد الفترة إلى 10 فـ 20 فـ 40 ثانية وهكذا... إليك فكرة عن الشيفرة:

```
let delay = 5000;

let timerId = setTimeout(function request() {
  // ...نُرسل الطلب...
  if (لو فشل الطلب لوجود ضغط على الخادم) {
    // نزيد الفترة حتى الطلب التالي
    delay *= 2;
  }

  timerId = setTimeout(request, delay);
}, delay);
```

ولو كانت الدوال التي نُجدولها ثقيلة على المعالج فيمكن أن نقيس الزمن الذي أخذتها عملية التنفيذ الحالية ونؤجّل أو نقدّم الاستدعاء التالي. يتيح لنا تداخل التوابع `setTimeout` بضبط الفترة بين عمليات التنفيذ بدقّة أعلى ممّا تقدّمه `setInterval`.

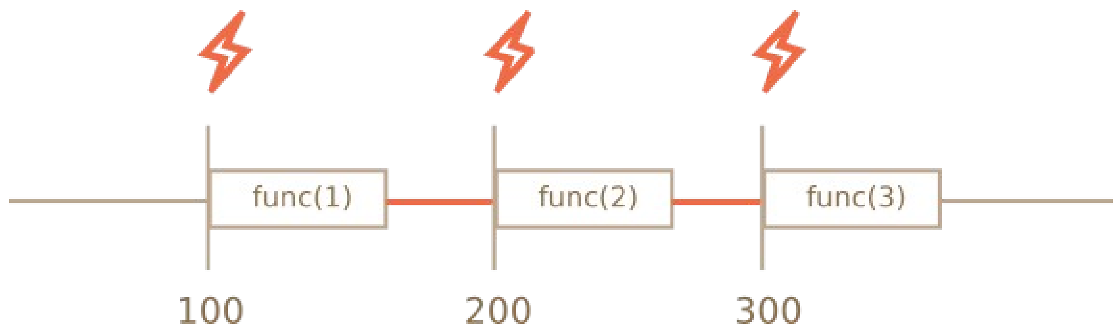
لنرى الفرق بين الشيفرتين أسفله. الأولى تستعمل `setInterval`:

```
let i = 1;
setInterval(function() {
  func(i++);
}, 100);
```

الثانية تستعمل `setTimeout` متداخلة:

```
let i = 1;
setTimeout(function run() {
  func(i++);
  setTimeout(run, 100);
}, 100);
```

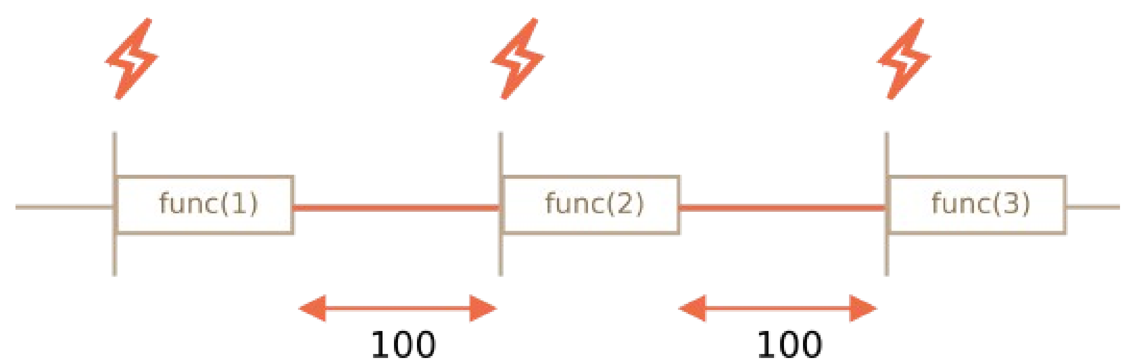
سيُشغّل المُجدول الداخلي `func(i++)` كلّ 100 مليثانية حسب `setInterval`:



هل لاحظت ذلك؟ التأخير الفعلي بين استدعاءات `func` التي ينفّذها `setInterval` أقل مما هي عليه في الشيفرة! هذا طبيعي إذ أنّ الوقت الذي يأخذه تنفيذ `func` يستهلك بعضًا من تلك الفترة أيضًا.

يمكن أيضًا بأن يصير تنفيذ `func` أكبر ممّا توقعناه على حين غرّة ويأخذ وقتًا أطول من 100 مليثانية. في هذه الحال ينتظر المحرّك انتهاء `func` ثمّ يرى المُجدول: لو انقضى الوقت يشغّل الدالة مباشرةً.

دومًا ما تأخذ الدالة وقتًا أطول من `delay` مليثانية في هذه الحالات الهامشية، إذ تجري الاستدعاءات واحدةً بعد الأخرى دون هوادة. وإليك صورة `setTimeout` المتداخلة:



تضمن `setTimeout` المتداخلة لنا التأخير الثابت (100 مليثانية في حالتنا). ذلك لأنّ الاستدعاء التالي لا يُجدول إلا بعد انتهاء السابق.

كنس المهملات وردود نداء التابعين setInterval و setTimeout

تُنشأ إشارة داخلية إلى الدالة (وتُحفظ في المُجدول) متى مرّرتها إلى إلى `setInterval/setTimeout`، وهذا يمنع كنس الدالة على أنّها مهملات، حتّى لو لم تكن هناك إشارات إليها.

```
// .clearInterval تبقى الدالة في الذاكرة حتى يُستدعى
setTimeout(function() {...}, 100);
```

ولكن هناك تأثير جانبي لذلك كالعادة، فالدوال تُشير إلى بيئتها المُعجمية الخارجية. لذا طالما "تعيش"، تعيش معها المتغيرات الخارجية أيضًا، وهي أحيانًا كبيرة تأخذ ذاكرة أكبر من الدالة ذاتها. لذا، متى ما لم ترد تلك الدالة المُجدولة فالأفضل أن تُلغيها حتّى لو كانت صغيرة جدًا.

6.8.4 جدولة setTimeout بتأخير "صفر"

إليك الحالة الخاصة: `setTimeout(func, 0)` أو `setTimeout(func)`. يُجدول هـذا التـابع ليحـدث تنفيذ `func` بأسر ع ما يمكن، إلّا أن المُجدول لن يشغّلها إلا بعد انتهاء السكربت الذي يعمل حاليًـا. أي أنّ الدالـة تُجدول لأن تعمل "مباشرةً بعد" السكربت الحالي.

فمثلًا تكتب هذه الشيفرة "Hello" ثم مباشرة "World":

```
setTimeout(() => alert("World"));
alert("Hello");
```

يعني السطر الأوّل "ضع الاستدعاء في التقويم بعد 0 مليثانية"، إلّا أنّ المُجدول لا "يفحص تقويمه" إلّا بعـد انتهاء السكربت الحالي، بهذا تصير `"Hello"` أولًا وبعدها تأتي `"World"`.

كما أنّ هناك استعمالات متقدّمة خصّيصًا للمتصفّحات للمهلة بالتأخير صفر هذه، وسنشـرحها في الفصـل "حلقة الأحداث: المهام على المستويين الجُسيمي والذرّي".

في الواقع، فالتأخير الصفر هذا ليس صفرًا (في المتصفّحات)، فتحدّ المتصفّحات من التـأخير بين تشـغيل المؤقّتات المتداخلة. تقول مواصفة HTML5: "بعد المؤقّتات المتداخلة الخمسة الأولى، تُجبر الفترة لتكـون أربـع مليثوان على الأقل.".

لنرى ما يعني ذلك بهذا المثال أسفله. يُعيد استدعاء `setTimeout` جدولة نفسه بمدّة تأخير تساوي صـفر، ويتـذكّر كـل اسـتدعاء الـوقت الفعلي بينـه وبين آخـر اسـتدعاء في مصـفوفة `times`. ولكن، مـا هي التـأخيرات الفعلية؟ لنرى بأعيننا:

```
let start = Date.now();
let times = [];
setTimeout(function run() {
  times.push(Date.now() - start); // نحفظ التأخير من آخر استدعاء
  if (start + 100 < Date.now()) alert(times); // نعرض التأخيرات بعد 100 م.ث
  else setTimeout(run); // وإلّا نُعيد الجدولة
});
// إليك مثالًا عن الناتج:
// 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,100
```

تعمل المؤقّتات الأولى مباشرةً (كما تقول المواصفة)، وبعدها نـرى `9, 15, 20, 24...`. تلـك الأربـع مليثوان الإضافية هي التأخير المفروض بين الاستدعاءات.

حتّى مع `setInterval` بدل `setTimeout`، ذات الأمر: تعمل الدالة `setInterval(f)` أوّل `f` مرّة بمدّة تأخير صفر، وبعدها تزيد أربع مليثوان لباقي الاستدعاءات. سبب وجود هذا الحدّ هو من العصور الحجرية (متعوّدة دائمًا) وتعتمد شيفرات كثيرة على هذا السلوك. بينما مع نسخة الخواديم من جافاسكربت فهذا الحدّ ليس له وجود، وهناك أيضًا طُرق أخرى لبدء المهام التزامنية مباشرةً، مثل setImmediate في بيئة Node.js، هذا قلنا بأنّ هذا يخصّ المتصفّحات فقط.

6.8.5 الخلاصة

- يتيح لنا التابعان `setTimeout` و`setInterval` تشغيل الدالة `func` مرّة أو كلّ فترة حسب كذا مليثانية (`delay`).
- لإلغاء التنفيذ علينا استدعاء `clearTimeout/clearInterval` بالقيمة التي أعاداها `setTimeout/setInterval`.
- يُعدّ استدعاء الدوال `setTimeout` تداخليًا خيارًا أفضل من `setInterval` إذ يُتيح لنا ضبط الوقت بين كلّ عملية استدعاء بدقّة.
- الجدولة بضبط التأخير على الصفر باستعمال `setTimeout(func, 0)` أو `setTimeout(func)` يكون حين نريدها "بأقصى سرعة ممكنة، متى انتهى السكربت الحالي".
- يَحدّ المتصفّح من أدنى تأخير بعد استدعاء `setTimeout` أو `setInterval` المتداخل الخامس (أو أكثر) - يَحدّه إلى 4 مليثوان، وهذا لأسباب تاريخية.

لاحظ بأنّ توابِع الجدولة لا تضمن التأخير كما هو حرفيًا. فمثلًا يمكن أن تكون مؤقّتات المتصفّحات أبطأ لأسباب عديدة:

- المعالج مُثقل بالعمليات.
- لسان المتصفّح يعمل في الخلفية.
- يعمل الحاسوب المحمول على البطارية.

يمكن لهذا كله رفع دقّة المؤقّت الدنيا (أي أدنى تأخير ممكن) لتصير 300 مليثانية أو حتى 1000 مليثانية حسب المتصفّح وإعدادات الأداء في نظام التشغيل.

6.8.6 تمارين

ا. اكتب الناتج كلّ ثانية

الأهمية: ★★★★★

اكتب الدالة `printNumbers(from, to)` لتكتب عددًا كلّ ثانية بدءًا بِـ `from` وانتهاءً بِـ `to`.

اصنع نسختين من الحل.

1. واحدةً باستعمال `setInterval`.
2. واحدةً باستعمال `setTimeout` متداخلة.

الحل:

باستعمال `setInterval`:

```
function printNumbers(from, to) {
  let current = from;

  let timerId = setInterval(function() {
    alert(current);
    if (current == to) {
      clearInterval(timerId);
    }
    current++;
  }, 1000);
}

// الاستعمال:
printNumbers(5, 10);
```

باستعمال `setTimeout` متداخلة:

```
function printNumbers(from, to) {
  let current = from;

  setTimeout(function go() {
    alert(current);
    if (current < to) {
```

```
        setTimeout(go, 1000);
      }
      current++;
    }, 1000);
}

// الاستعمال:
printNumbers(5, 10);
```

لاحظ كلا الحلّين: هناك تأخير أولي قبل أول عملية كتابة إذ تُستدعى الدالة بعد `1000ms` في أوّل مرة.

لو أردت تشغيل الدالة مباشرةً فعليك كتابة استدعاء إضافي في سطر آخر هكذا:

```
function printNumbers(from, to) {
  let current = from;

  function go() {
    alert(current);
    if (current == to) {
      clearInterval(timerId);
    }
    current++;
  }

  go(); // هنا
  let timerId = setInterval(go, 1000);
}

printNumbers(5, 10);
```

ب. ماذا سيعرض setTimeout؟

الأهمية: ★★★★★

جدول أحدهم في الشيفرة أسفله استدعاء `setTimeout`، وثمّ كتب عملية حسابية ثقيلة لتعمل (وهي تأخذ أكثر من 100 مليثانية حتى تنتهي).

متى ستعمل الدالة المُجدولة؟

1. بعد الحلقة؟

2. قبل الحلقة؟

3. في بداية الحلقة؟

ما ناتج `alert`؟

```
let i = 0;

setTimeout(() => alert(i), 100); // ?

// عَدّ بأنّ الوقت اللازم لتنفيذ هذه الدالة يفوق 100 مليثانية
for(let j = 0; j < 100000000; j++) {
  i++;
}
```

الحل:

لن يُشغّل أيّ تابِع `setTimeout` إلا بعدما تنتهي الشيفرة الحالية.

ستكون قيمة `i` هي القيمة الأخيرة: `100000000`.

```
let i = 0;

setTimeout(() => alert(i), 100); // 100000000

// عَدّ بأنّ الوقت اللازم لتنفيذ هذه الدالة يفوق 100 مليثانية
for(let j = 0; j < 100000000; j++) {
  i++;
}
```

6.9 المزخرفات والتمرير: التابعان call و apply

تقدّم لنا لغة جافاسكربت مرونة عالية غير مسبوقة في التعامل مع الدوال، إذ يمكننا تمريرها أو استعمالها على أنّها كائنات. والآن سنرى كيف نُمرّر الاستدعاءات بينها وكيف نُزخرِفها.

6.9.1 خبيئة من خلف الستار

لنقل بأنّ أمامنا الدالة الثقيلة على المعالج `slow(x)` بينما نتائجها مستقرة، أي لنقل بأنّنا لو مرّرنا ذات `x`، فسنجد ذات النتيجة دومًا.

لو استدعينا هذه الدالة مرارًا وتكرارًا، فالأفضل لو خبّئنا (أي تذكّرنا) ناتجها لئلا يذهب الوقت سدًى لإجراء ذات الحسابات. ولكن، بدل إضافة هذه الميزة في دالة `slow()` نفسها، ستُنشئ دالة غالِفة تُضيف ميزة الخبيئة هذه. سنرى أسفله مدى فوائد هذا الأمر.

إليك الشيفرة أولًا، وبعدها الشرح:

```
function slow(x) {
  // هنا مهمّة ثقيلة تُهلك المعالج
  alert(`Called with ${x}`);
  return x;
}

function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {    // لو وجدنا هذا المفتاح في الخبيئة
      return cache.get(x); // نقرأ النتيجة منها
    }

    let result = func(x);  // وإلّا نستدعي الدالة

    cache.set(x, result);  // ثمّ نُخبّئ (نتذكّر) ناتجها
    return result;
  };
}
```

```
slow = cachingDecorator(slow);

alert( slow(1) ); // خبّأنا slow(1)
alert( "Again: " + slow(1) ); // ذات الناتج

alert( slow(2) ); // slow(2) خبّأنا
alert( "Again: " + slow(2) ); // ذات ناتج السطر السابق
```

في الشيفرة أعلاه، ندعو cachingDecorator بالمُزخرِف (decorator): وهي دالة خاصّة تأخذ دالة أخرى مُعاملًا وتعدّل على سلوكها.

الفكرة هنا هي استدعاء cachingDecorator لأيّ دالة أردنا، وستُعيد لنا غِلاف الخبيئة ذاك. الفكرة هذه رائعة إذ يمكن أن نكون أمام مئات من الدوال التي يمكن أن تستغلّ هذه الميزة، وكلّ ما علينا فعله هو إضافة cachingDecorator عليها.

كما وأنّا نحافظ على الشيفرة أبسط بفصل ميزة الخبيئة عن مهمّة الدالة الفعلية حيث ناتج cachingDecorator(func) هو "غِلاف" يُعيد الدالة function(x) التي "تُغلّف" استدعاء func(x) داخل شيفرة الخبيئة:

```
function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }

    let result = func(x);

    cache.set(x, result);
    return result;
  };
}
```


الشيفرات الخارجية لا ترى أيّ تغيير على دالة slow المُغلّفة. ما فعلناه هو تعزيز سلوكها بميزة الخبيئة.

إذًا نُلخّص: ثمّة فوائد عدّة لاستعمال cachingDecorator منفصلًا بدل تعديل شيفرة الدالة slow نفسها:

- إعادة استعمال cachingDecorator، فنُضيفه على دوال أخرى.
- فصل شيفرة الخبيئة فلا تزيد من تعقيد دالة slow نفسها (هذا لو كانت معقّدة).
- إمكانية إضافة أكثر من مُزخرف decorator عند الحاجة (سنرى ذلك لاحقًا).

6.9.2 استعمال func.call لأخذ السياق

لا ينفع مُزخرِف الخبيئة الذي شرحناه مع توابِع الكائنات. فمثلًا في الشيفرة أسفله، سيتوقّف عمل `worker.slow()` بعد هذه الزخرفة:

```
// هيّا نُضف ميزة الخبيئة إلى worker.slow
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    // أمامنا مهمّة ثقيلة على المعالج هنا
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

// نفس الشيفرة أعلاه
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func(x); // (**)
    cache.set(x, result);
    return result;
  };
}

alert( worker.slow(1) ); // التابع الأصلي يعمل كما ينبغي

worker.slow = cachingDecorator(worker.slow); // وقت الخبيئة

alert( worker.slow(2) ); // لاااا! خطأ: تعذّرت قراءة الخاصية 'someMethod' في undefined
```

مكان الخطأ هو السطر `(*)` الذي يحاول الوصول إلى `this.someMethod` ويفشل فشلًا ذريعًا. هل تعرف السبب؟

السبب هو أنّ الغِلاف يستدعي الدالة الأصلية هكذا `func(x)` في السطر `(**)`. وحين نستدعيها هكذا تستلم الدالة `this = undefined`. سنرى ما يشبه هذا الخطأ لو شغّلنا هذه الشيفرة:

```
let func = worker.slow;
func(2);
```

إذًا... يُمرّر الغِلاف الاستدعاء إلى التابع الأصلي دون السياق `this`، بهذا يحصل الخطأ.

حان وقت الإصلاح! ثمّة تابِع دوال مضمّن في اللغة باسم func.call(context, ...args) يتيح لنا استدعاء الدالة. صياغته هي:

```
func.call(context, arg1, arg2, ...)
```

يُشغّل التابع الدالةَ `func` بعد تمرير المُعامل الأول (وهو `this`) وثمّ مُعاملاتها. للتبسيط، هذين الاستدعاءين لا يفرقان بشيء في التنفيذ:

```
func(1, 2, 3);
func.call(obj, 1, 2, 3)
```

فكلاهما يستدعي `func` بالمُعاملات 1 و2 و3. الفرق الوحيد هو أنّ `func.call` تضبط قيمة `this` على `obj` علاوةً على ذلك.

لنأخذ مثالًا. في الشيفرة أسفله نستدعي `sayHi` بسياق كائنات أخرى: يُشغّل `sayHi.call(user)` الدالةَ `sayHi` ويُمرّر `this=user`، ثمّ في السطر التالي يضبط `this=admin`:

```
function sayHi() {
  alert(this.name);
}

let user = { name: "Ahmad" };
let admin = { name: "Admin" };

// نستعمل call على أنها لنمرّر مختلف الكائنات this:
sayHi.call( user ); // this = Ahmad
sayHi.call( admin ); // this = Admin
```

وهنا نستدعي `call` لتستدعي `say` بالسياق والعبارة المُمرّرتين:

```
function say(phrase) {
  alert(this.name + ': ' + phrase);
}

let user = { name: "Ahmad" };

// الكائن user يصير this وتصير Hello المُعامل الأول
say.call( user, "Hello" ); // Ahmad: Hello
```

في حالتنا نحن، يمكن استعمال call في الغِلاف ليُمرّر السياق إلى الدالة الأصلية:

```
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func.call(this, x); // هكذا نُمرّر "this" كما ينبغي
    cache.set(x, result);
    return result;
  };
}

worker.slow = cachingDecorator(worker.slow); // والآن نُضيف الخبيئة
alert( worker.slow(2) ); // يعمل
alert( worker.slow(2) ); // يعمل ولا يستدعي التابع الأصلي (إذ القيمة مُخبّأة)
```

الآن يعمل كلّ شيء كما نريد!

لنوضّح الأمر أكثر، لنرى بالتفصيل المملّ تمريرات `this` من هنا إلى هناك:

1. بعد الزخرفة، يصير `worker.slow` الغِلاف `function (x) { ... }`.
2. لذا حين نُنفّذ `worker.slow(2)`، يأخذ الغِلاف القيمةَ `2` وسيطًا ويضبط `this=worker` (وهو الكائن قبل النقطة).
3. في الغِلاف (باعتبار أنّ النتيجة لم تُخبّأ بعد)، تُمرّر `func.call(this, x)` قيمة `this` الحالية (وهي `worker`) مع المُعامل الحالي (`2`) - كلّه إلى التابع الأصلي.

6.9.3 استعمال أكثر من وسيط داخل func.apply

الآن صار وقت تعميم `cachingDecorator` على العالم. كنّا إلى هنا نستعملها مع الدوال التي تأخذ مُعاملًا واحدًا فقط.

وماذا لو أردنا تخبئة التابع `worker.slow` الذي يأخذ أكثر من مُعامل؟

```
let worker = {
  slow(min, max) {
    return min + max; // نُعدّها عملية تستنزف المعالج
  }
};

// علينا تذكّر الاستدعاءات بنفس المُعامل هنا
worker.slow = cachingDecorator(worker.slow);
```

كنّا سابقًا نستعمل `cache.set(x, result)` (حين تعاملنا مع المُعامل الوحيد `x`) لنحفظ الناتج، ونستعمل `cache.get(x)` لنجلب الناتج. أمّا الآن فعلينا تذكّر ناتج مجموعة مُعاملات `(min,max)`. الخارطة `Map` لا تأخذ المفاتيح إلّا بقيمة واحدة.

ثمّة أمامنا أكثر من حلّ:

1. كتابة بنية بيانات جديدة تشبه الخرائط (أو استعمال واحدة من طرف ثالث) يمكن استعمالها لأكثر من أمر وتسمح لنا بتخزين أكثر من مفتاح.
2. استعمال الخرائط المتداخلة: تصير `cache.set(min)` خارطة تُخزّن الزوجين `(max, result)`. ويمكن أن نأخذ الناتج `result` باستعمال `cache.get(min).get(max)`.

3. دمج القيمتين في واحدة. في حالتنا هذه يمكن استعمال السلسلة النصية `"min,max"` لتكون مفتاح `Map`. ويمكن أن نقدّم للمُزخرِف دالة عنونة Hashing يمكنها صناعة قيمة من أكثر من قيمة، فيصير الأمر أسهل.

أغلب التطبيقات العملية تَعدّ الحل الثالث كافيًا، ولهذا سنستعمله هنا.

علينا أيضًا استبدال التابع `func.call(this, x)` بالتابع `func.call(this, ...arguments)` كي نُمرّر كلّ المُعاملات إلى استدعاء الدالة المُغلّفة لا الأولى فقط.

رحّب بالمُزخرف `cachingDecorator` الجديد، أكثر قوة وأناقة:

```
let worker = {
  slow(min, max) {
    alert(`Called with ${min},${max}`);
    return min + max;
  }
};

function cachingDecorator(func, hash) {
  let cache = new Map();
  return function() {
    let key = hash(arguments); // (*)
    if (cache.has(key)) {
      return cache.get(key);
    }

    let result = func.call(this, ...arguments); // (**)

    cache.set(key, result);
    return result;
  };
}

function hash(args) {
  return args[0] + ',' + args[1];
}

worker.slow = cachingDecorator(worker.slow, hash);
```

```
alert( worker.slow(3, 5) ); // يعمل
alert( "Again " + worker.slow(3, 5) ); // نفس الناتج (خبّأناه)
```

الآن صار يعمل مهما كان عدد المُعاملات (ولكن علينا تعديل دالة العنونة لتسمح هي أيضًا بالمُعاملات أيًا كان عددها. سنشره أسفله إحدى الطرائق الجميلة لإنجاز هذه المهمة).

أمامنا تعديلان اثنان:

- في السطر`(*)`، نستدعي `hash` لتصنع مفتاحًا واحدًا من `arguments`. نستعمل هنا دالة "دمج" بسيطة تحوّل المُعاملان `(3, 5)` إلى المفتاح `"3,5"`. لو كانت الحالة لديك أكثر تعقيدًا، فتحتاج إلى دوال عنونة أخرى.
- ثمّ يستعمل `(**)` التابع `func.call(this, ...arguments)` لتمرير السياق وكلّ المُعاملات التي استلمها الغِلاف (وليس الأول فقط) - كله إلى الدالة الأصلية.

يمكننا بدل استعمال `func.call()` استغلال `func.apply()`.

صياغة هذا التابع المبني في اللغة func.apply هي:

```
func.apply(context, args)
```

يُشغّل التابع الدالةَ `func` بضبط `this=context` واستعمال الكائن الشبيه بالمصفوفات `args` قائمةً بالمُعطيات للدالة. الفارق الوحيد بين `call` و`apply` هي أنّ الأوّل يتوقّع قائمة بالمُعطيات بينما الثاني يأخذ كائنًا شبيهًا بالمصفوفات يحويها.

أي أنّ الاستدعاءين الآتين متساويين تقريبًا:

```
func.call(context, ...args); // نمرّر الكائن قائمةً بمُعامل التوزيع
func.apply(context, args);   // نفس الفكرة باستعمال apply
```

ولكن هناك فرق بسيط واحد:

- يُتيح لنا مُعامل التوزيع `...` تمرير المُتعدَّد `args` قائمةً إلى `call`.
- لا يقبل `apply` إلّا مُعامل `args` شبيه بالمصفوفات.

أي أنّ هذين الاستدعاءين يُكمّلان بعضهما البعض. لو توقّعنا وصول مُتعدَّد فنستعمل `call`، ولو توقّعنا شبيهًا بالمصفوفات نستعمل `apply`.

أمّا الكائنات المُتعدَّدة والشبيهة بالمصفوفات (مثل المصفوفات الحقيقية)، فيمكننا نظريًا استعمال أيّ من الاثنين، إلّا أنّ apply سيكون أسرع غالبًا إذ أنّ مُعظم محرّكات جافاسكربت تحسّن أدائه داخليًا أكثر من call.

يُدعى تمرير كافة المُعاملات (مع السياق) من دالة إلى أخرى بتمرير الاستدعاء. إليك أبسط صوره:

```
let wrapper = function() {
  return func.apply(this, arguments);
};
```

حين تستدعي أيّ شيفرة خارجية wrapper محال أن تفرّق بين استدعائها واستدعاء الدالة الأصلية func.

6.9.4 استعارة التوابع borrowing a method

أمّا الآن لنحسّن دالة العنونة قليلًا:

```
function hash(args) {
  return args[0] + ',' + args[1];
}
```

لا تعمل الدالة حاليًا إلّا على مُعاملين اثنين، وسيكون رائعًا لو أمكن أن ندمج أيّ عدد من args.

أوّل حلّ نفكّر به هو استعمال التابع arr.join:

```
function hash(args) {
  return args.join();
}
```

ولكن... للأسف فهذا لن ينفع، إذ نستدعي hash(arguments) بتمرير كائن المُعاملات arguments المُتعدَّد والشبيه بالمصفوفات... إلّا أنّه ليس بمصفوفة حقيقية.

بذلك استدعاء join سيفشل كما نرى أسفله:

```
function hash() {
  alert( arguments.join() ); // خطأ: arguments.join ليست بدالة
}

hash(1, 2);
```

مع ذلك فما زال هناك طريقة سهلة لضمّ عناصر المصفوفة:

```
function hash() {
  alert( [].join.call(arguments) ); // 1,2
}

hash(1, 2);
```

ندعو هذه الخدعة باستعارة التوابع. فيها نأخذ (أي نستعير) تابِع الضمّ من المصفوفات العادية (`[].join`) ونستعمل `[].join.call` لتشغيله داخل سياق `arguments`.

ولكن، لمَ تعمل أصلًا؟

هذا بسبب بساطة الخوارزمية الداخلية للتابِع الأصيل `arr.join(glue)` في اللغة.

أقتبس -بتصرّف خفيف جدًا- من مواصفات اللغة:

1. لمّا أنّ `glue` هو المُعامل الأول، ولو لم تكن هناك مُعاملات فهو `","`.
2. لمّا أنّ `result` هي سلسلة نصية فارغة.
3. أضِف `this[0]` إلى نهاية `result`.
4. أضِف `glue` و`this[1]`.
5. أضِف `glue` و`this[2]`.
6. ...كرّر حتّى ينتهي ضمّ العناصر ذات الطول `this.length`.
7. أعِد `result`.

إذًا فهو يأخذ `this` ويضمّ `this[0]` ثمّ `this[1]` وهكذا معًا. كتب المطوّرون التابع بهذه الطريقة عمدًا ليسمح أن تكون `this` أيّ شبيه بالمصفوفات (ليست مصادفة إذ تتبع كثير من التوابع هذه الممارسة). لهذا يعمل التابع حين يكون `this=arguments`.

6.9.5 المُزخرِفات decorators وخاصيات الدوال function properties

استبدال الدوال أو التوابع بأخرى مُزخرفة هو أمر آمن عادةً، ولكن باستثناء صغير: لو احتوت الدالة الآلية على خاصيات (مثل `func.calledCount`) فلن تقدّمها الدالة المُزخرفة، إذ أنّها غِلاف على الدالة الأصلية. علينا بذلك أن نحذر في هذه الحالة. نأخذ المثال أعلاه مثالًا، لو احتوت الدالة `slow` أيّ خاصيات فلن يحتوي الغِلاف `cachingDecorator(slow)` عليها.

يمكن أن تقدّم لنا بعض المُزخرِفات خاصيات خاصة بها. فمثلًا يمكن أن يعدّ المُزخرِف كم مرّة عملت الدالة وكم من وقت أخذ ذلك، وتقدّم لنا خاصيات لنرى هذه لمعلومات.

توجد طريقة لإنشاء مُزخرِفات تحتفظ بميزة الوصول إلى خاصيات الـدوال، ولكنّهـا تطلب اسـتعمال الكـائن الوسيط Proxy لتغليف الدوال. سنشرح هذا الكائن لاحقًا في قسم "تغليف الدوال: apply".

6.9.6 الخلاصة

تُعدّ المُزخرِفات أغلفة حول الدوال فتعدّل سلوكها، بينما المهمة الأساس مرهونة بالدالـة نفسـها. يمكن عـدّ المُزخرِفات "مزايا" نُضيفها على الدالة، فنُضيف واحدة أو أكثر، ودون تغيير أيّ سطر في الشيفرة!

رأينا التوابع الآتية لنعرف كيفية إعداد المُزخرِف cachingDecorator:

- func.call(context, arg1, arg2…): يستدعي func حسب السياق والمُعاملات الممرّرة.
- func.apply(context, args): يستدعي func حيث يُمرّر context بصـفته this والكـائن الشـبيه بالمصفوفات args في قائمة المُعاملات.

عادةً ما نكتب تمرير الاستدعاءات باستعمال apply:

```
let wrapper = function() {
  return original.apply(this, arguments);
};
```

كما رأينا مثالًا عن استعارة التوابع حيث أخذنا تابِعًا من كائن واسـتدعيناه call في سـياق كـائن آخـر غـيره. يشـيع بين المطـوّرين أخـذ توابِـع المصـفوفات وتطبيقهـا على المُعـاملات arguments. لـو أردت بـديلًا لـذلك فاستعمل كائن المُعاملات البقية إذ هو مصفوفة حقيقية.

ستجد في رحلتك المحفوفة بالمخاطر مُزخرِفات عديدة. حاوِل التمرّس عليها بحلّ تمارين هذا الفصل.

6.9.7 تمارين

ا. مُزخرِف تجسّس

الأهمية: ★★★★★

أنشِئ المُزخرِف spy(func) ليُعيد غِلافًا يحفظ كلّ استدعاءات تلك الدالة في خاصية calls داخله.

احفظ كلّ استدعاء على أنّه مصفوفة من الوُسطاء.

مثال:

```
function work(a, b) {
  alert( a + b ); // ليست work إلّا دالة أو تابِعًا لسنا نعرف أصله
}
```

```
work = spy(work); // (*)

work(1, 2); // 3
work(4, 5); // 9

for (let args of work.calls) {
  alert( 'call:' + args.join() ); // "call:1,2", "call:4,5"
}
```

تذكر أننا نستفيد من هذا المُزخرِف أحيانًا لاختبار الوحـدات. يمكن عـدّ `sinon.spy` في المكتبـة Sinon.JS صورةً متقدّمةً عنه.

الحل:

سيُخزّن الغِلاف الذي أعادته `spy(f)` كلّ الوُسطاء، بعدها يستعمل `f.apply` لتمرير الاستدعاء.

```
function spy(func) {

  function wrapper(...args) {
    // استعملنا ...arg بدلًا من arguments للحصول على مصفوفة حقيقية في wrapper.calls
    wrapper.calls.push(args);
    return func.apply(this, args);
  }

  wrapper.calls = [];

  return wrapper;
}
```

جرب الحل في بيئة تجريبية.

ب. مُزخرِف تأخير

الأهمية: ★★★★★

أنشِئ المُزخرف `delay(f, ms)` ليُؤخّر كلّ استدعاء من `f` بمقدار `ms` مليثانية.

مثال:

```
function f(x) {
  alert(x);
}

// أنشِئ الغِلافات
let f1000 = delay(f, 1000);
let f1500 = delay(f, 1500);

f1000("test"); // يعرض "test" بعد 1000 مليثانية
f1500("test"); // يعرض "test" بعد 1500 مليثانية
```

أي أنّ المُزخرِف `delay(f, ms)` يُعيد نسخة عن `f` "تأجّلت `ms`". الدالة `f` في الشيفرة أعلاه تقبل وسيطًا واحدًا، ولكن على الحل الذي ستكتبه تمرير كلّ الوُسطاء والسياق `this` كذلك.

الحل:

```
function delay(f, ms) {

  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
  };

}

let f1000 = delay(alert, 1000);

f1000("test"); // يعرض test بعد 1000 مليثانية
```

لاحظ بأنّا استعملنا الدالة السهمية هنا. كما نعلم فالدوال السهمية لا تملك لا `this` ولا `arguments`، لذا يأخذ `f.apply(this, arguments)` كِلا `this` و`arguments` من الغِلاف.

لو مرّرنا دالة عادية فسيستدعيها `setTimeout` بدون المُعاملات ويضبط `this=window` (باعتبار أنّا في بيئة المتصفّح).

مع ذلك يمكننا تمرير قيمة `this` الصحيحة باستعمال متغيّر وسيط ولكنّ ذلك سيكون تعبًا لا داعٍ له:

```
function delay(f, ms) {
  return function(...args) {
```

```
    let savedThis = this; // خزّنه في متغير وسيط
    setTimeout(function() {
      f.apply(savedThis, args); // استعمل الوسيط هنا
    }, ms);
  };

}
```

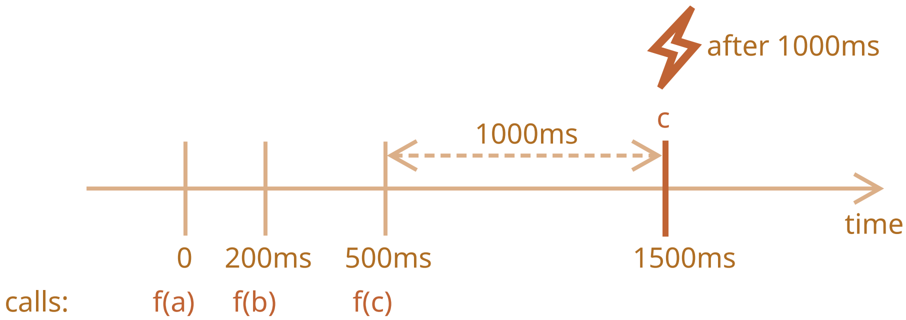
جرب الحل في بيئة تجريبية.

ج. مُزخرف إزالة ارتداد

الأهمية: ★★★★★

اصنع المُزخرِف `debounce(f, ms)` ليُعيد غِلافًا يُمرّر الاستدعاء إلى `f` مرّة واحدة كلّ `ms` مليثانية.

بعبارة أخرى: حين ندعو الدالة "بأنّ ارتدادها أُزيل" Debounce فهي تضمن لنا بأنّ الاستدعاءات التي ستحدث في أقلّ من `ms` مليثانية بعد الاستدعاء السابق - ستُهمل.



إليك مثال (مأخوذ من مكتبة Lodash):

```
let f = _.debounce(alert, 1000);

f("a");
setTimeout( () => f("b"), 200);
setTimeout( () => f("c"), 500);
// alert("c") تنتظر 1000 م.ث بعد آخر استدعاء ثم تُنفذ
```

عمليًا في الشيفرات، نستعمل `debounce` للدوال التي تستلم أو تُحدّث شيئًا ما نعرف مسبقًا بأنّ لا شيء جديد سيحدث له في هذه الفترة القصيرة، فالأفضل أن نُهمله ولا نُهدر الموارد. مثلًا، لنفكر بحالة عملية عندما نريد أن نرسل طلبًا للخادم كلما أدخل المستخدم حرفًا (للبحث مثلًا) فلا حاجة إلى إرسال طلب عند إدخال كل

حرف والأفضل أن ننتظر قليلًا ريثما ينتهي المستخدم وبذلك نعالج الكلمة المدخلة كلها. وهذا يُطبق على كل حالات المعالجة للمدخلات لحقول الإدخال `input`.

إن فكرت بالحل، قبل النظر للشيفرة التالية، فهو عبارة عن بضعة سطور!

الحل:

```
function debounce(func, ms) {
  let timeout;
  return function() {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, arguments), ms);
  };
}
```

استدعاء `debounce` يُعيد غِلافًا. عند استدعائها، تجدول الدالة الأصلية بعد عدد الثواني المحدد وتمسح أي استدعاء سابق مُجدول.

د. مُزخرِف خنق

الأهمية: ★★★★★

أنشِئ مُزخرِف "الخنق/throttle" `throttle(f, ms)` ليُعيد غِلافًا يُمرّر الاستدعاء إلى `f` مرّة كلّ `ms` مليثانية. والاستدعاءات التي تحدث في فترة "الراحة" تُهمل.

الفرق بين هذه وبين `debounce` هي أنّه لو كان الاستدعاء المُهمل هو آخر الاستدعاءات أثناء فترة الراحة، فسيعمل متى انتهت تلك الفترة. لنطالع هذا التطبيق من الحياة العملية لنعرف أهمية هذا الشيء الغريب العجيب وما أساسه أصلًا.

لنقل مثلًا أنّا نريد تعقّب تحرّك الفأرة. يمكن أن نضبط دالة (في المتصفّح) لتعمل كلّما تحرّكت الفأرة وتأخذ مكان المؤشّر أثناء هذه الحركة. لو كنت تستعمل الفأرة فعادةً ما تعمل الدالة هذه بسرعة (ربما تكون 100 مرّة في الثانية، أي كلّ 10 مليثوان).

نريد تحديث بعض المعلومات في صفحة الوِب أثناء حركة المؤشّر ولكن تحديث الدالة `update()` عملية ثقيلة ولا تنفع لكلّ حركة فأرة صغيرة. كما وليس منطقيًا أصلًا التحديث أكثر من مرّة كلّ 100 مليثانية.

لذا نُغلّف الدالة في مُزخرف: نستعمل `throttle(update, 100)` على أنّها دالة التشغيل كلّما تحرّكت الفأرة بدلًا من الدالة `update()` الأصلية. سيُستدعى المُزخرِف كثيرًا صحيح، ولكنّها لن يمرّر الاستدعاءات هذه إلى `update()` إلّا مرّة كلّ 100 مليثانية.

هكذا سيظهر للمستخدم:

1. في أوّل تحريك للفأرة، تُمرّر نسختنا المُزخرفة من الدالة الاستدعاء مباشرةً إلى `update`، وهذا مهمّ إذ يرى المستخدم كيف تفاعلت الصفحة مباشرةً مع تحريكه الفأرة.
2. ثمّ يُحرّك المستخدم الفأرة أكثر، ولا يحدث شيء طالما لم تمرّ `100ms`. نسختنا المُزخرفة الرائعة تُهمل تلك الاستدعاءات.
3. بعد نهاية `100ms` يعمل آخر استدعاء `update` حاملًا الإحداثيات الأخيرة.
4. وأخيرًا تتوقّف الفأرة عن الحراك. تنتظر الدالة المُزخرفة حتى تمضي `100ms` وثمّ تشغّل `update` حاملةً آخر الإحداثيات. وهكذا نُعالج آخر حركة للفأرة، وهذا مهم

مثال عن الشيفرة:

```
function f(a) {
  console.log(a);
}

// تمرّر f1000 الاستدعاءات إلى f كحدّ أقصى مليثانية 1000 مرّة كلّ
let f1000 = throttle(f, 1000);

f1000(1); // 1 تعرض
f1000(2); // (مخنوقة، لم تمض 1000 مليثانية بعد)
f1000(3); // (مخنوقة، لم تمض 1000 مليثانية بعد)

// حين تمضي 1000 مليثانية...
// ...تطبع 3، إذ القيمة 2 الوسطية أُهملت
```

يجب تمرير المُعاملات والسياق `this` المُمرّرة إلى `f1000`- تمريرها إلى `f` الأصلية.

الحل:

```
function throttle(func, ms) {

  let isThrottled = false,
    savedArgs,
    savedThis;

```

```
  function wrapper() {

    if (isThrottled) { // (2)
      savedArgs = arguments;
      savedThis = this;
      return;
    }

    func.apply(this, arguments); // (1)

    isThrottled = true;

    setTimeout(function() {
      isThrottled = false; // (3)
      if (savedArgs) {
        wrapper.apply(savedThis, savedArgs);
        savedArgs = savedThis = null;
      }
    }, ms);
  }

  return wrapper;
}
```

يُعيد استدعاء `throttle(func, ms)` الغِلاف `wrapper`.

1. أثناء الاستدعاء الأول، يُشغّل `wrapper` ببساطة الدالة `func` ويضبط حالة الراحة (`isThrottled = true`).
2. في هذه الحالة نحفظ كلّ الاستدعاءات في `savedArgs/savedThis`. لاحظ بأنّ السياق والوُسطاء مهمّان ويجب حفظهما كلاهما، فنحتاج إليهما معًا لنُعيد ذلك الاستدعاء كما كان ونستدعيه حقًا.
3. بعد مرور `ms` مليثانية، يعمل `setTimeout`، بهذا تُزال حالة الراحة (`isThrottled = false`) ولو كانت هناك استدعاءات مُهملة، نُنفّذ `wrapper` بآخر ما حفظنا من وُسطاء وسياق.

لا نشغّل في الخطوة الثالثة `func` بل `wrapper` إذ نريد تنفيذ `func` إضافةً إلى دخول حالة الراحة ثانيةً وضبط المؤقّت لتصفيرها.

6.10 ربط الدوال Function binding

ثمّة مشكلة معروفة تواجهنا متى مرّرنا توابِع الكائنات على أنّها ردود نداء (كما نفعل مع `setTimeout`)، هي ضياع هويّة الأنا `this`.

سنرى في هذا الفصل طرائق إصلاح هذه المشكلة.

6.10.1 ضياع الأنا (الكلمة المفتاحية this)

رأينا قبل الآن أمثلة كيف ضاعت قيمة `this`. فما نلبث أن مرّرنا التابع إلى مكان آخر منفصلًا عن كائنه، ضاع `this`.

إليك ظواهر هذه المشكلة باستعمال `setTimeout` مثلًا:

```
let user = {
  firstName: "Ahmad",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(user.sayHi, 1000); // Hello, undefined!
```

كما رأينا في ناتج الشيفرة، لم نرحّب بالأخ "Ahmad" (كما أردنا باستعمال `this.firstName`)، بل بالأخ غير المعرّف `undefined`!

هذا لأنّ التابع `setTimeout` استلم الدالة `user.sayHi` منفصلةً عن كائنها. يمكن أن نكتب السطر الأخير هكذا:

```
let f = user.sayHi;
setTimeout(f, 1000); // user ضاع سياق المستخدم
```

بالمناسبة فالتابع `setTimeout` داخل المتصفّحات يختلف قليلًا، إذ يضبط `this=window` حين نستدعي الدالة (بينما في Node.js يصير `this` هو ذاته كائن المؤقّت، ولكنّ هذا ليس بالأمر المهم الآن). يعني ذلك بأنّ `this.firstName` هنا هي فعليًا `window.firstName`، وهذا المتغير غير موجود. عادةً ما تصير `this` غير معرّفة `undefined` في الحالات الأخرى.

كثيرًا ما نواجه هذه المسألة ونحن نكتب الشيفرة: نريد أن نمرّر تابِع الدالة إلى مكان آخر (مثل هنا، مرّرناه للمُجدول) حيث سيُستدعى من هناك. كيف لنا أن نتأكّد بأن يُستدعى في سياقه الصحيح؟

6.10.2 الحل رقم واحد: نستعمل دالة مغلفة

أسهل الحلول هو استعمال دالة غالِفة Wrapping function:

```
let user = {
  firstName: "Ahmad",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(function() {
  user.sayHi(); // Hello, Ahmad!
}, 1000);
```

الآن اكتملت المهمة إذ استلمنا المستخدم user من البيئة المُعجمية الخارجية، وثمّ استدعينا التابع كما العادة. إليك ذات المهمة بأسطر أقل:

```
setTimeout(() => user.sayHi(), 1000); // Hello, Ahmad!
```

ممتازة جدًا، ولكن ستظهر لنا نقطة ضعف في بنية الشيفرة.

ماذا لو حدث وتغيّرت قيمة user قبل أن تعمل setTimeout؟ (لا تنسَ التأخير، ثانية كاملة!) حينها سنجد أنّا استدعينا الكائن الخطأ دون أن ندري!

```
let user = {
  firstName: "Ahmad",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(() => user.sayHi(), 1000);
// ...تغيّرت قيمة user خلال تلك الثانية
user = {
  sayHi() { alert("Another user in setTimeout!"); }
};
// هناك مستخدم آخر داخل التابع setTimeout!
```

الحل الثاني سيضمن لنا ألّا تحدث هكذا أمور غير متوقّعة.

6.10.3 الحل رقم اثنين: ربطة

تقدّم لنا الدوال تابِعًا مضمّنًا في اللغة باسم bind يتيح لنا ضبط قيمة this. إليك صياغته الأساسية:

```
// ستأتي الصياغة المعقّدة لاحقًا لا تقلق
let boundFunc = func.bind(context);
```

ناتِج التابع func.bind(context) هو "كائن دخيل" يشبه الدالة ويمكن لنا استدعائه على أنّه دالة، وسيمرّر هذا الاستدعاء إلى func بعدما يضبط this=context من خلف الستار.

أي بعبارة أخرى، لو استدعينا boundFunc فكأنّما استدعينا func بعدما ضبطنا قيمة this. إليك مثالًا تمرّر فيه funcUser الاستدعاء إلى func بضبط this=user:

```
let user = {
  firstName: "Ahmad"
};

function func() {
  alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // Ahmad
```

رأينا "النسخة الرابطة" من func، func.bind(user) بعد ضبط this=user.

كما أنّ المُعاملات كلّها تُمرّر إلى دالة func الأصلية "كما هي". مثال:

```
let user = {
  firstName: "Ahmad"
};
function func(phrase) {
  alert(phrase + ', ' + this.firstName);
}
// user إلى this نربط
let funcUser = func.bind(user);
funcUser("Hello"); // (this=user كما وُضبط "Hello" مُرّر المُعامل) Hello, Ahmad
```

فلنجرّب الآن مع تابع لكائن:

```
let user = {
  firstName: "Ahmad",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

let sayHi = user.sayHi.bind(user); // (*)

// يمكن أن نشغّلها دون وجود كائن
sayHi(); // Hello, Ahmad!
setTimeout(sayHi, 1000); // Hello, Ahmad!

// حتّى لو تغيّرت قيمة user خلال تلك الثانية
// فما زالت تستعمل sayHi القيمة التي ربطناها قبلًا
user = {
  sayHi() { alert("Another user in setTimeout!"); }
};
```

أخذنا في السطر `(*)` التابع `user.sayHi` وربطناه مع المستخدم `user`. ندعو الدالة `sayHi` بالدالة "المربوطة" حيث يمكن أن نستدعيها لوحدها هكذا أو نمرّرها إلى `setTimeout`. مهما فعلًا فسيكون السياق صحيحًا كما نريد.

نرى هنا أنّ المُعاملات مُرّرت "كما هي" وما ضبطه هو `bind` قيمة `this` فقط:

```
let user = {
  firstName: "Ahmad",
  say(phrase) {
    alert(`${phrase}, ${this.firstName}!`);
  }
};

let say = user.say.bind(user);
say("Hello"); // (say إلى "Hello" مُرّر المُعامل) Hello, Ahmad!
say("Bye"); // (say إلى "Bye" مُرّر المعامل) Bye, Ahmad!
```

تابِع مفيد: `bindAll`

لو كان للكائن توابِع كثيرة وأردنا تمريرها هنا وهناك بكثرة، فربّما نربطها كلّها في حلقة:

```
for (let key in user) {
  if (typeof user[key] == 'function') {
    user[key] = user[key].bind(user);
  }
}
```

كما تقدّم لنا مكتبات جافاسكربت دوال للربط الجماعي لتسهيل الأمور، مثل ‎_.bindAll(obj)‎ في المكتبة lodash.

6.10.4 الدوال الجزئية partial functions

طوال هذه الفترة لم نُناقش شيئًا إلّا ربط `this`. لنُضيف شيئًا آخر على الطاولة. يمكن أيضًا أن نربط المُعاملات وليس `this` فحسب. صحيح أنّا نادرًا ما نفعل ذلك إلّا أنّ الأمر مفيد في أحيان عصيبة.

صياغة `bind` الكاملة:

```
let bound = func.bind(context, [arg1], [arg2], ...);
```

وهي تسمح لنا بربط السياق ليكون `this` والمُعاملات الأولى في الدالة.

نرى مثالًا: دالة ضرب `mul(a, b)`:

```
function mul(a, b) {
  return a * b;
}
```

فلنستعمل `bind` لنصنع دالة "ضرب في اثنين" `double` تتّخذ تلك أساسًا لها:

```
function mul(a, b) {
  return a * b;
}
let double = mul.bind(null, 2);

alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10
```

يصنع استدعاء `mul.bind(null, 2)` دالةً جديدة `double` تُمرّر الاستدعاءات إلى `mul` وتضبط `null` ليكون السياق و`2` ليكون المُعامل الأول. الباقي من مُعاملات يُمرّر "كما هو".

هذا ما نسمّيه باستعمال الدوال الجزئية أي أن نصنع دالة بعد ضبط بعض مُعاملات واحدة غيرها. لاحظ هنا بأنّا لا نستعمل this هنا أصلًا… ولكنّ التابع bind يطلبه فعلينا تقديم شيء (وكان null مثلًا).

الدالة triple أسفله تضرب القيمة في ثلاثة:

```
function mul(a, b) {
  return a * b;
}
let triple = mul.bind(null, 3);

alert( triple(3) ); // = mul(3, 3) = 9
alert( triple(4) ); // = mul(3, 4) = 12
alert( triple(5) ); // = mul(3, 5) = 15
```

ولكن لماذا نصنع الدوال الجزئية أصلًا، وعادةً؟!

الفائدة هي إنشاء دالة مستقلة لها اسم سهل القراءة (double أو triple)، فنستعملها دون تقديم المُعامل الأول في كلّ مرة إذ ضبطنا قيمته باستعمال bind. وهناك حالات أخرى يفيدنا الاستعمال الجزئي هذا حين نحتاج نسخة أكثر تحديدًا من دالة عامّة جدًا، ليسهُل استعمالها فقط.

فمثلًا يمكن أن نصنع الدالة send(from, to, text). وبعدها في كائن المستخدم user نصنع نسخة جزئية عنها: sendTo(to, text) تُرسل النصّ من المستخدم الحالي.

6.10.5 الدوال الجزئية، بدون السياق

ماذا لو أردنا أن نضبط بعض المُعاملات ولكن دون السياق this؟ مثلًا نستعملها لتابِع أحد الكائنات.

تابِع bind الأصيل في اللغة لا يسمح بذلك، ومستحيل أن نُزيل السياق ونضع المُعاملات فقط. لكن لحسن الحظ فيمكننا صنع دالة مُساعدة partial تربط المُعاملات فقط.

هكذا تمامًا:

```
function partial(func, ...argsBound) {
  return function(...args) { // (*)
    return func.call(this, ...argsBound, ...args);
  }
}
// الاستعمال:
let user = {
  firstName: "Ahmad",
```

```
    say(time, phrase) {
      alert(`[${time}] ${this.firstName}: ${phrase}!`);
    }
  };

  // نُضيف تابِعًا جزئيًا بعد ضبط الوقت
  user.sayNow = partial(user.say, new Date().getHours() + ':' + new
  Date().getMinutes());

  user.sayNow("Hello");
  // وسيظهر ما يشبه الآتي:
  // [10:00] Ahmad: Hello!
```

نـاتِج اسـتدعائنا للدالـة `partial(func[, arg1, arg2...])` هـو غِلاف `(*)` يسـتدعي الدالة `func` هكذا:

- يترك `this` كما هو (فتكون قيمته `user` داخل الاستدعاء `user.sayNow`)
- ثمّ يمرّر لها `...argsBound`: أي المُعاملات من استدعاء `partial` (`"10:00"`)
- وثمّ يمرّر لها `...args`: المُعاملات الممرّرة للغِلاف (`"Hello"`)

ساعدنا مُعامل التوزيع كثيرًا هنا، أم لا؟

كما أنّ هناك شيفرة _.partial في المكتبة lodash.

6.10.6 الخلاصة

يُعيد التابع `func.bind(context, ...args)` "نسخة مربوطة" من الدالة `func` بعد ضبط سياقها `this` ومُعاملاتها الأولى (في حال مرّرناها).

عادةً ما نستعمل `bind` لنضبط `this` داخل تابِع لأحد الكائنات، فيمكن أن نمرّر التابع ذلك إلى مكـان آخـر، مثلًا إلى `setTimeout`.

وحين نضبط بعضًا من مُعاملات إحدى الدوال، يكون الناتج (وهو أكثر تفصيلًا) دالةً ندعوها بالدالة الجزئية أو المطبّقة بنحوٍ جزئي partially applied.

تُفيدنا هذه الدوال الجزئية حين لا نريد تكرار ذات الوسيط مرارًا وتكرارًا، مثل دالة `send(from, to)` حيث يجب أن يبقى `from` كما هو في مهمّتنا هذه، فنأخذ دالة جزئية ونتعامل بها.

6.10.7 تمارين

ا. دالة ربط على أنّها تابِع

الأهمية: ★★★★★

ما ناتج هذه الشيفرة؟

```
function f() {
  alert( this ); // ؟
}

let user = {
  g: f.bind(null)
};

user.g();
```

الحل:

الجواب هو: `null`.

سياق دالة الربط مكتوب في الشيفرة (hard-coded) ولا يمكن تغييره لاحقًا بأيّ شكل من الأشكال.

فحتّى لو شغّلنا `user.g()` فستُستدعى الدالة الأصلية بضبط `this=null`.

ب. ربطة ثانية

الأهمية: ★★★★★

هي يمكن أن نغيّر قيمة `this` باستعمال ربطة إضافية؟

ما ناتج هذه الشيفرة؟

```
function f() {
  alert(this.name);
}
f = f.bind( {name: "Ahmad"} ).bind( {name: "Ann" } );
f();
```

الحل:

الجواب هو: Ahmad.

```
function f() {
  alert(this.name);
}

f = f.bind( {name: "Ahmad"} ).bind( {name: "Pete"} );

f(); // Ahmad
```

لا يتذكّر كائن دالة الربط "الدخيل" (الذي يُعيده `f.bind(...)`) السياق (مع الوُسطاء إن مُرّرت) - لا يتذكّر هذا كلّه إلى وقت إنشاء الكائن.

أي: لا يمكن إعادة ربط الدوال.

ج. خاصية الدالة بعد الربط

الأهمية: ★★★★★

تمتلك خاصية إحدى الدوال قيمة ما. هل ستتغيّر بعد `bind`؟ نعم، لماذا؟ لا، لماذا؟

```
function sayHi() {
  alert( this.name );
}
sayHi.test = 5;

let bound = sayHi.bind({
  name: "Ahmad"
});

alert( bound.test ); // ما الناتج؟ لماذا؟
```

الحل:

الجواب هو: `undefined`.

ناتِج `bind` هو كائن آخر، وليس في هذا الكائن خاصية `test`.

د. أصلِح هذه الدالة التي يضيع "this" منها

الأهمية: ★★★★★

على الاســـــتدعاء `askPassword()` في الشـــــيفرة أســـــفله فحص كلمـــــة الســـــر، ثمّ اســـــتدعاء `user.loginOk/loginFail` حسب نتيجة الفحص.

ولكن أثناء التنفيذ نرى خطأً. لماذا؟

أصلِح الجزء الذي فيه `(*)` لتعمل الشيفرة كما يجب (تغيير بقية الأسطر ممنوع).

```
function askPassword(ok, fail) {
  let password = prompt("Password?", '');
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'Ahmad',

  loginOk() {
    alert(`${this.name} logged in`);
  },

  loginFail() {
    alert(`${this.name} failed to log in`);
  },

};

askPassword(user.loginOk, user.loginFail); // (*)
```

الحل:

سبب الخطأ هو أنّ الدالة `ask` تستلم الدالتين `loginOk/loginFail` دون كائنيهما.

فمتى ما استدعتهما، تُعدّ `this=undefined` بطبيعتها.

علينا ربط السياق!

```
function askPassword(ok, fail) {
  let password = prompt("Password?", '');
```

```
    if (password == "rockstar") ok();
    else fail();
  }

  let user = {
    name: 'Ahmad',

    loginOk() {
      alert(`${this.name} logged in`);
    },

    loginFail() {
      alert(`${this.name} failed to log in`);
    },

  };

  // (*)\maskPassword(user.loginOk.bind(user),
  user.loginFail.bind(user));
```

الآن صارت تعمل. أو، بطريقة أخرى:

```
//...
askPassword(() => user.loginOk(), () => user.loginFail());
```

هذه الشيفرة تعمل وعادةً ما تكون سهلة القراءة أيضًا.

ولكنّها في حالات أكثر تعقيدًا تصير أقلّ موثوقية، مثل لو تغيّر المتغير `user` بعدما استُدعيت الدالة `askPassword` وقبل أن يُجيب الزائر على الاستدعاء `() => user.loginOk()`.

ه. استعمال الدوال الجزئية لولوج المستخدم

هذا التمرين معقّد أكثر من سابقه، بقليل. هنا تعدّل كائن `user`، فصار فيه بدل الدالتين `loginOk/loginFail` دالة واحدة `user.login(true/false)`.

ما الأشياء التي نمرّرها إلى `askPassword` في الشيفرة أسفله فتستدعي `user.login(true)` باستعمال `ok` وتستدعي `user.login(false)` باستعمال `fail`؟

```
function askPassword(ok, fail) {
```

```
  let password = prompt("Password?", '');
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'Ahmad',

  login(result) {
    alert( this.name + (result ? ' logged in' : ' failed to log in')
);
  }
};

askPassword(?, ?); // ؟ (*)
```

يجب أن تعدّل الجزء الذي عليه `(*)` فقط لا غير.

الحل:

نستعمل دالة غالِفة... سهمية لو أردنا التفصيل:

```
askPassword(() => user.login(true), () => user.login(false));
```

هكذا تأخذ `user` من المتغيرات الخارجية وتُشغّل الدوال بالطريقة العادية.

أو نصنع دالة جزئية من `user.login` تستعمل `user` سياقًا لها ونضع مُعاملها الأول كما يجب:

```
```
askPassword(user.login.bind(user, true), user.login.bind(user,
false));
```
```

6.11 الحديث عن الدوال السهمية Arrow functions مرة أخرى

لنرى الدوال السهمية مرّة أخرى.

لو ظننت الدوال السهمية هي طريقة مختصرة لكتابة الشيفرات القصيرة، فأنت مخطئ، إذ لهذه الدوال مزايا تختلف عن غيرها وتفيدنا جدًا. كثيرًا ما نواجه المواقف (في جافاسكربت بالتحديد) التي نريد أن نكتب فيها دالة صغيرة وننفّذها في مكان آخر.

مثال:

- `arr.forEaoch(func)`: تُنفّذ `forEach` الدالة `func` لكلّ عنصر في المصفوفة.
- `setTimeut(func)`: يُنفّذ المجدول الداخلي في البيئة دالة `func`.
- …وغيرها وغيرها.

هذا هو جوهر اللغة، أن نصنع دالة في مكان ونمرّرها إلى مكان آخر. وفي هذه الدوال عادةً ما لا نريد أن نترك سياقها الحالي، وهنا تأتي الفائدة المخفية للدوال السهمية.

6.11.1 ليس للدوال السهمية مفهوم this

كما ذكرنا من فصل "الدوال في الكائنات واستعمالها this" فليس في الدوال السهمية مفهوم `this`، ولو حاولت الوصول إلى قيمة `this` فستأخذها الدالة من الخارج.

فمثلًا يمكننا استعمالها للمرور على العناصر داخل تابِع للكائن:

```
let group = {
  title: "Our Group",
  students: ["Ahmad", "Pete", "Alice"],

  showList() {
    this.students.forEach(
      student => alert(this.title + ': ' + student)
    );
  }
};

group.showList();
```

استعملنا هنا في `forEach` الدالة السهمية، وقيمة `this.title` فيها هي تمامًا القيمة التي يراها التابع الخارجي `showList`، أي `group.title`.

لو استعملنا هنا الدوال العادية فسنواجه خطأً:

```
let group = {
  title: "Our Group",
  students: ["Ahmad", "Pete", "Alice"],

  showList() {

    this.students.forEach(function(student) {
      // خطأ: تعذرت قراءة الخاصية 'title' لغير المعرَّف undefined
      alert(this.title + ': ' + student)
    });
  }

};

group.showList();
```

سبب هذا الخطأ هو أنّ التابع `forEach` يشغّل الدوال بتمرير `this=undefined` مبدئيًا، وبذلك تحاول الشيفرة الوصول إلى `undefined.title`.

ليس لهذا أيّ تأثير على الدوال السهمية إذ ليس لها `this` أساسًا.

لا يمكن تشغيل الدوال السهمية باستعمال new

بطبيعة الحال فدون `this` تواجه حدًّا آخر: لا يمكنك استعمال الدوال السهمية على أنّها مُنشِئات دوال، أي لا يمكنك استدعاءها باستعمال new.

انتبه إلى أن هناك فرق بسيط بين الدالة السهمية `=>` والدالة العادية التي نستدعيها باستعمال `.bind(this)`:

- يُنشئ التابع `.bind(this)` "نسخة مربوطة" من تلك الدالة.
- لا يصنع السهم `=>` أيّ نوع من الربطات. الدالة ليس فيها `this`، فقط. يبحث المحرّك عن قيمة `this` كما يبحث عن أيّ قيمة متغير آخر: في البيئة المُعجمية الخارجية للدالة السهمية.

6.11.2 ليس للدوال السهمية "مُعاملات"

كما وأنّ الدوال السهمية ليس فيها متغير مُعاملات `arguments`.

وهذا أمر رائع حين نتعامل مع المُزخرِفات إذ نُمرّر الاستدعاء حاملًا قيمة `this` الحالية مع المُعاملات `arguments`. فمثلًا هنا تأخذ `defer(f, ms)` دالةً وتُعيد غِلافًا (Wrapper) عليها تُؤجّل الاستدعاء بالمليثوان `ms` الممرّرة:

```
function defer(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms)
  };
}

function sayHi(who) {
  alert('Hello, ' + who);
}

let sayHiDeferred = defer(sayHi, 2000);
sayHiDeferred("Ahmad"); // بعد مرور ثانيتين Hello, Ahmad
```

يمكن كتابة نفس الشيفرة دون استعمال دالة سهمية هكذا:

```
function defer(f, ms) {
  return function(...args) {
    let ctx = this;
    setTimeout(function() {
      return f.apply(ctx, args);
    }, ms);
  };
}
```

هنا لزم أن نصنع المتغيرين الإضافيين `args` و `ctx` لتقدر الدالة في `setTimeout` على أخذ قيمهما.

6.11.3 الخلاصة

ليس للدوال السهمية:

- لا `this`.

- ولا `arguments`.
- ولا يمكن استدعائها باستعمال `new`.
- وليس فيها `super`... لم نشرح ذلك بعد ولكنّا سنفعل في الفصل "وراثة الأصناف".

ليس فيها هذا كله لأنّ الغرض منها كتابة شيفرات قصيرة ليس لها سياق تعتمد عليه بل سياقًا تأخذه، وهنا حين "تتألّق" هذه الدوال.

أكبر سوق عربي لبيع وشراء الخدمات المصغرة
اعرض خدماتك أو احصل على ما تريد بأسعار تبدأ من 5$ فقط

تصفح الخدمات

7. ضبط خاصيات الكائنات

يتضمن هذا الفصل الأقسام التالية:

1. رايات الخاصيات وواصِفاتها
2. جالبات الخاصيات Getters وضابطاتها Setters

7.1 رايات الخاصيات وواصِفاتها

كما نعلم فالكائنات تُخزّن داخلها خاصيات properties تصفها. وحتى الآن لم تكن الخاصية إلا زوجًا من مفاتيح وقيم، ولكن خاصية الكائن يمكن أن تكون أكثر قوّة ومرونة من هذا.

سنتعلم في هذا الفصل بعض خصائص الضبط الأخرى، وفي الفصل الّذي يليه سنرى كيف نحوّلها إلى دوال جلب Getters وضبط Setters أيضًا.

7.1.1 رايات الخاصيات

لخصائص الكائنات (إضافةً إلى القيمة الفعلية لها) ثلاث سمات أخرى مميزة (أو "رايات" flags):

- قابلية التعديل: لو كانت بقيمة `true` فيمكننا تغيير القيمة وتعديلها، ولو لم تكن فالقيمة للقراءة فقط.
- قابلية الإحصاء: لو كانت بقيمة `true` فستقدر الحلقات على المرور على عناصرها، وإلا فلن تقدر.
- قابلية إعادة الضبط: لو كانت بقيمة `true` فيمكن حذف الخاصية وتعديل هذه السمات، وإلا فلا يمكن.

لم نتطرّق إلى هذه الرايات قبلًا إذ لا تظهر عادةً في الشيفرات، فحين ننشئ خاصية "بالطريقة العادية" فكلّ هذه السمات بقيمة `true`، ولكن يمكننا طبعًا تغييرها متى أردنا. أولًا لنعرف كيف سنرى هذه الرايات.

يتيح لنا التابع Object.getOwnPropertyDescriptor الاستعلامَ عن المعلومات الكاملة الخاصة بأيّ خاصية. وهذه صياغته:

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

المعاملات:

- `obj`: الكائن الّذي سنجلب معلوماته.
- `propertyName`: اسم الخاصية الّتي نريد.

نسمّي القيمة المُعادة بكائن "واصِف الخاصيات" Property Descriptor، وهو يحتوي على القيمة وجميع الرايات الّتي سبق لنا شرحها. إليك مثالًا:

```
let user = {
  name: "Ahmad"
};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');
```

```
alert( JSON.stringify(descriptor, null, 2 ) );
/* واصف الخاصية:
{
  "value": "Ahmad",
  "writable": true,
  "enumerable": true,
  "configurable": true
}
*/
```

يمكننا استعمال التابع Object.defineProperty لتغيير الرايات. إليك صياغته:

```
Object.defineProperty(obj, propertyName, descriptor)
```

المعاملات:

- `obj` و `propertyName`: الكائن الّذي سنطبّق عليه الواصِف، واسم الخاصية.
- `descriptor`: واصِف الخاصيات الّذي سنطبّقه على الكائن.

لو كانت الخاصية موجودة فسيُحدّث التابع `defineProperty` راياتها. وإلّا فسيُنشئ الخاصية بهذه القيمة الممرّرة والرايات كذلك، وفي هذه الحالة لو لم يجد قيمة لأحد الرايات، فسيعدّه بقيمة `false`.

مثلًا هنا نُنشئ الخاصية `name` حيث تكون راياتها كلّها بقيمة `false`:

```
let user = {};

Object.defineProperty(user, "name", {
  value: "Ahmad"
});

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": "Ahmad",
  // لاحظ هنا
  "writable": false,
```

```
    "enumerable": false,
    "configurable": false
  }
   */
```

وازِن هـذه الخاصـية بتلـك الّـتي صـنعناها أعلاه `user.name` (كالعـادة): وأصـبحت قيمـة جميـع الرايـات `false`. لو لم يكن هذا ما تريده فربّما الأفضل ضبطها على `true` في كائن `descriptor`.

لنرى الآن تأثيرات هذه الرايات في هذا المثال.

7.1.2 منع قابلية التعديل

لنمنـع قابليـة التعـديل على الخاصـية `user.name` (أي اسـتحالة إسـناد قيمـة لهـا) وذلـك بتغيـير راية `writable`:

```
let user = {
  name: "Ahmad"
};

Object.defineProperty(user, "name", {
  writable: false // هنا
});

user.name = "Pete";
// خطأ: لا يمكن إسناد القيم إلى الخاصية `name` إذ هي للقراءة فقط
```

الآن يستحيل على أيّ شخص تعديل اسم هذا المستخدم إلّا لو طبّقوا تابِع `defineProperty` من طرفهم ليُلغي ما فعلناه نحن.

لا تظهر الأخطاء إلا في الوضع الصارم

لا تظهر أي أخطاء عند تعديل قيمة خاصية في وضع غير الصارم إلّا أنها لن تتغير قيمتها بطبيعة الحال، وذلك لأن خطأ خرق الراية لن يظهر إلا في الوضع الصارم.

إليك نفس المثال ولكن دون إنشاء الخاصية من الصفر:

```
let user = { };

Object.defineProperty(user, "name", {
```

```
    value: "Ahmad",
    // لو كانت الخاصيات جديدة فعلينا إسناد قيمها إسنادًا صريحًا
    enumerable: true,
    configurable: true
});

alert(user.name); // Ahmad
user.name = "Pete"; // Error
```

7.1.3 منع قابلية الإحصاء

الآن لنُضيف تابِع toString مخصّص على كائن user!

عادةً لا يمكننا استخدام التابع toString على الكائنات، وذلك لأنها غير قابلة للإحصاء، ولذلك فلا يمكن تمريرها على حلقة for..in. ولكن إن أردنا تغيير ذلك يدويًا (كما في المثال التالي) عندها يمكننا تمريرها إلى حلقة for..in.

```
let user = {
  name: "Ahmad",
  toString() {
    return this.name;
  }
};

// مبدئيًا، ستعرض الشيفرة الخاصيتين معًا:
for (let key in user) alert(key); // name, toString
```

لو لم نرد ذلك فيمكن ضبط enumerable:false حينها لن نستطع أن نمرر الكائن في حلقات for..in كما في السلوك المبدئي:

```
let user = {
  name: "Ahmad",
  toString() {
    return this.name;
  }
};

Object.defineProperty(user, "toString", {
```

```
  enumerable: false // هنا
});

// الآن اختفى تابع toString:
for (let key in user) alert(key); // name
```

كما أنّ التابع Object.keys يستثني الخاصيات غير القابلة للإحصاء:

```
alert(Object.keys(user)); // name
```

7.1.4 منع قابلية إعادة الضبط

أحيانًا ما نرى راية "قابلية إعادة الضبط" ممنوعة (أي configurable:false) في بعض الكائنات والخاصيات المضمّنة في اللغة. لا يمكن حذف هذه الخاصية لو كانت ممنوعة (أي configurable:false).

فمثلًا المتغيّر المضمّن في اللغة Math.PI يمنع قابلية التعديل والإحصاء وإعادة الضبط عليه:

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": 3.141592653589793,
  "writable": false,
  "enumerable": false,
  "configurable": false
}
*/
```

هكذا لا يقدر المبرمج على تغيير قيمة Math.PI ولا الكتابة عليها.

```
Math.PI = 3; // خطأ

// delete Math.PI لن تعمل أيضًا
```

إن تفعيل خاصيّة منع قابلية إعادة الضبط هو قرار لا عودة فيه، فلا يمكننا تغيير الراية (إتاحة قابلية إعادة الضبط) باستعمال التابع defineProperty.

وللدقّة فهذا المنع يضع تقييدات أخرى على defineProperty:

1. منع تغيير راية قابلية إعادة الضبط `configurable`.
2. منع تغيير راية قابلية الإحصاء `enumerable`.
3. منع تغيير راية قابلية التعديل `writable: false` إلى القيمة `true` (ولكن العكس ممكن).
4. منع تغيير ضابط وجالب واصف الوصول `get/set` (ولكن يمكن إسناد قيم إليه).

هنا سنحدّد الخاصية `user.name` لتكون ثابتة للأبد :

```
let user = { };

Object.defineProperty(user, "name", {
  value: "Ahmad",
  writable: false,
  configurable: false
});


Object.defineProperty(user, "name", {writable: true}); // خطأ
```

نلاحظ عدم إمكانية تغيير الخاصيّة `user.name` ولا حتى راياتها ولن نستطيع تطبيق هذه العمليات عليها:

1. الكتابة عليها `user.name = "Pete"`.
2. حذفها `delete user.name`.
3. تغيير قيمتها باستخدام التابع `defineProperty` بالشكل:

```
defineProperty(user, "name", { value: "Pete" })
```

"إن منع قابلية إعادة الضبط" ليس "منعًا لقابلية التعديل" حيث إن فكرة منع قابلية إعـادة الضـبط هي في الحقيقة لمنع تغيير رايات هذه الخاصية أو حذفها، وليس تغيير قيمة الخاصية بحد ذاتها.

> في المثال السابق جعلنا قابلية التعديل ممنوعة يدويًا.

7.1.5 التابع Object.defineProperties

هنــاك أيضًــا التــابع Object.defineProperties إذ يُــتيح تعريــف أكــثر من خاصــية في وقت واحد صياغته هي:

```
Object.defineProperties(obj, {
  prop1: descriptor1,
  prop2: descriptor2
  // ...
});
```

مثال عليه:

```
Object.defineProperties(user, {
  name: { value: "Ahmad", writable: false },
  surname: { value: "Smith", writable: false },
  // ...
});
```

أي أنّنا نقدر على ضبط أكثر من خاصية معًا.

7.1.6 التابع Object.getOwnPropertyDescriptors

يمكننا استعمال التابع Object.getOwnPropertyDescriptors(obj) لجلب كلّ واصفات الخاصيات معًا. ويمكن استعماله بدمجه مع `Object.defineProperties` لنسخ الكائنات "ونحن على علمٍ براياتها":

```
let clone = Object.defineProperties({},
Object.getOwnPropertyDescriptors(obj));
```

فعادةً حين ننسخ كائنًا نستعمل الإسناد لنسخ الخاصيات، هكذا:

```
for (let key in user) {
  clone[key] = user[key]
}
```

ولكن... هذا لا ينسخ معه الرايات. لذا يفضّل استعمال `Object.defineProperties` لو أردنا نُسخةً "أفضل" عن الكائن. الفرق الثاني هو أنّ حلقة `for..in` تتجاهل الخاصيات الرمزية (Symbolic Properties)، ولكنّ التابع `Object.getOwnPropertyDescriptors` يُعيد كلّ واصِفات الخاصيات بما فيها الرمزية.

7.1.7 إغلاق الكائنات على المستوى العام

تعمل واصِفات الخاصيات على مستوى الخاصيات منفردةً. هناك أيضًا توابِع تقصر الوصول إلى الكائن كلّه:

- Object.preventExtensions(obj): يمنع إضافة خاصيات جديدة إلى الكائن.

- Object.seal(obj): يمنع إضافة الخاصيات وإزالتها، فهو يمنع قابلية إعادة الضبط أي `configurable: false` على كلّ الخاصيات الموجودة.
- Object.freeze(obj): يمنع إضافة الخاصيات أو إزالتها أو تغييرها، إذ يمنع قابلية التعديل `writable: false` وقابلية إعادة الضبط `configurable: false` على كلّ الخاصيات الموجودة.

كما أنّ هناك توابِع أخرى تفحص تلك المزايا:

- Object.isExtensible(obj): يُعيد `false` لو كان ممنوعًا إضافة الخاصيات، وإلا `true`.
- Object.isSealed(obj): يُعيد `true` لو كان ممنوعًا إضافة الخاصيات أو إزالتها، وكانت كلّ خاصيات الكائن الموجودة ممنوعة من قابلية إعادة الضبط `configurable: false`.
- Object.isFrozen(obj): يُعيد `true` لو كان ممنوعًا إضافة الخاصيات أو إزالتها أو تغييرها، وكانت كلّ خاصيات الكائن الموجودة ممنوعة أيضًا من قابلية التعديل `writable: false` أو إعادة الضبط `configurable: false`.

أمّا على أرض الواقع، فنادرًا ما نستعمل هذه التوابع.

7.2 جالبات الخاصيات Getters وضابطاتها Setters

يوجد نوعين من الخاصيات. الأوّل هو خاصيات البيانات Data Properties. نعرف جيدًا كيف نعمل مع هذا النوع إذ كلّ ما استعملناه من البداية إلى حدّ الساعة هي خاصيات بيانات.

النوع الثاني هو الجديد، وهو خاصيات الوصول Accessor Properties. هي في الأساس دوال تجلب القيم وتضبطها، ولكن في الشيفرة تظهرُ لنا وكأنها خاصيات عادية.

7.2.1 الجالبات والضابطات

خاصيات الوصول هذه هي توابِع "جلب" getter و"ضبط" setter.

```
let obj = {
  get propName() {
    // جالب، يُستعمَل جلب قيمة الخاصية obj.propName
  },

  set propName(value) {
    // ضابط يُستعمَل لضبط قيمة الخاصية obj.propName إلى value
  }
};
```

يعمل الجالب متى ما طلبت قراءة الخاصية `obj.propName`، والضابط... متى ما أردت إسناد قيمة `obj.propName = value`. لاحظ مثلًا كائن `user` له خاصيتين: اسم `name` ولقب `surname`:

```
let user = {
  name: "Ahmad",
  surname: "Smith"
};
```

الآن نريد إضافة خاصية الاسم الكامل `fullName`، وهي `"Ahmad Smith"`. طبعًا لا نريد نسخ المعلومات ولصقها، لذا ستُنفذها باستخدام خاصية الوصول (get):

```
let user = {
  name: "Ahmad",
  surname: "Smith",

  // لاحظ
```

```
  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

alert(user.fullName); // Ahmad Smith
```

خارج الكائن لا تبدو خاصية الوصول إلا خاصية عادية، وهذا بالضبط الغرض من هذه الخاصيات، فلسنا نريد استدعاء `user.fullName` على أنّها دالة، بل قراءتها فحسب، ونترك الجالب يقوم بعمله خلف الكواليس.

الآن ليس للخاصية `fullName` إلا جالبًا، لو حاولنا إسناد قيمة لها `=user.fullName` فسنرى خطأً:

```
let user = {
  get fullName() {
    return `...`;
  }
};

user.fullName = "Test"; // خطأ (للخاصية جالب فقط)
```

هيا نُصلح الخطأ ونُضيف ضابطًا للخاصية `user.fullName`:

```
let user = {
  name: "Ahmad",
  surname: "Smith",

  get fullName() {
    return `${this.name} ${this.surname}`;
  },
  // هنا
  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  }
};
// نضبط fullName كما النية بتمرير القيمة.
user.fullName = "Alice Cooper";
alert(user.name); // Alice
alert(user.surname); // Cooper
```

وهكذا صار لدينا الخاصية "الوهمية" `fullName`. يمكننا قراءتها والكتابة عليها، ولكنها في واقع الأمر، غير موجودة.

7.2.2 واصِفات الوصول Accessor Descriptor

واصِفات خاصيات الوصول Accessor Properties تختلف عن واصِفات خاصيات البيانات Data Properties. فليس لخاصيات الوصول قيمة `value` أو راية `writable`، بل هناك دالة `get` ودالة `set`.

أي يمكن لواصِف الوصول أن يملك ما يلي:

- `get`: دالة ليس لها وُسطاء تعمل متى ما قُرئت الخاصية.
- `set`:: دالة لها وسيط واحد تُستدعى متى ما ضُبطت الخاصية.
- `enumerable`: خاصية قابلية الإحصاء وهي مشابهة لخاصيّات البيانات.
- `configurable`: خاصية قابلية إعادة الضبط وهي مشابهة لخاصيّات البيانات.

فمثلًا لننشئ خاصية الوصول `fullName` باستعمال التابع `defineProperty`، يمكننا تمرير واصِفًا فيه دالة `get` ودالة `set`:

```
let user = {
  name: "Ahmad",
  surname: "Smith"
};

// هنا
Object.defineProperty(user, 'fullName', {
  get() {
    return `${this.name} ${this.surname}`;
  },

  set(value) {
    [this.name, this.surname] = value.split(" ");
  }
});

alert(user.fullName); // Ahmad Smith
for(let key in user) alert(key); // name, surname
```

أُعيد بأنّ الخاصية إمّا تكون خاصية وصول (لها توابِع `get/set`) أو خاصية بيانات (لها قيمة `value`)، ولا تكون الاثنين معًا. فلو حاولنا تقديم `get` مع `value` في نفس الواصِف، فسنرى خطأً:

```
// خطأ: واصِف الخاصية غير صالح.
Object.defineProperty({}, 'prop', {
  get() {
    return 1
  },

  value: 2
});
```

7.2.3 الجوالب والضوابط الذكيّة

يمكننا استعمال الجوالب والضوابط كأغلفة للخاصيات "الفعلية"، فتكون في يدنا السيطرة الكاملة على العمليات التي تؤثّر عليها.

فمثلًا لو أردنا منع الأسماء القصيرة للاسم `user` فيمكن كتابة الضابِط `name` وترك القيمة في خاصية منفصلة باسم `_name`:

```
let user = {
  get name() {
    return this._name;
  },

  set name(value) {
    if (value.length < 4) {
      alert("Name is too short, need at least 4 characters"); // الاسم قصير جدًا. أقلّ طول هو 4 محارف
      return;
    }
    this._name = value;
  }
};

user.name = "Pete";
alert(user.name); // Pete
user.name = ""; // الاسم قصير جدًا...
```

هكذا نخزّن الاسم في الخاصية `_name` والوصول يكون عبر الجالب والضابط.

عمليًا يمكن للشيفرة الخارجية الوصول إلى الاسم مباشرةً باستعمال `user._name`، ولكن هناك مفهوم شائع بين المطوّرين هو أنّ الخاصيات التي تبدأ بشرطة سفلية "_" هي خاصيات داخلية وممنوع التعديل عليها من خارج الكائن.

7.2.4 استعمالها لغرض التوافقية

إحدى استعمالات خاصيات الوصول هذه هي إتاحة الفرصة للتحكّم بخاصية بيانات "عادية" متى أردنا واستبدالها بدالتي جلب وضبط وتعديل سلوكها.

لنقل مثلًا بأنّا بدأنا المشروع حيث كانت كائنات المستخدمين تستعمل خاصيات البيانات: الاسم `name` والعمر `age`:

```
function User(name, age) {
  this.name = name;
  this.age = age;
}

let john = new User("Ahmad", 25);

alert( john.age ); // 25
```

ولكن الأمور لن تبقى على حالها وإنما ستتغير، عاجلًا أم آجلًا. فبدل العمر `age` نقول بأنّا نريد تخزين تاريخ الميلاد `birthday` إذ هو أكثر دقّة وسهولة في الاستعمال:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;
}

let john = new User("Ahmad", new Date(1992, 6, 1));
```

ولكن... كيف سنتعامل مع الشيفرة القديمة الّتي ما زالت تستعمل خاصية `age`؟

يمكن أن نبحث في كلّ أمكان استخدام الخاصية `age` وتغييرها بخاصية جديدة مناسبة، ولكنّ ذلك يأخذ وقتًا ويمكن أن يكون صعبًا لو عدة مبرمجين يعملون على هذه الشيفرة، كما وأنّ وجود عمر المستخدم كخاصية `age` أمر جيّد، أليس كذلك؟

إذًا لنُبقي الخاصية كما هي، ونُضيف جالبًا للخاصية تحلّ لنا المشكلة:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;

  // العمر هو الفرق بين التاريخ اليوم وتاريخ الميلاد
  Object.defineProperty(this, "age", {
    get() {
      let todayYear = new Date().getFullYear();
      return todayYear - this.birthday.getFullYear();
    }
  });
}

let john = new User("Ahmad", new Date(1992, 6, 1));

alert( john.birthday );   // تاريخ الميلاد موجود
alert( john.age );       // وعمر المستخدم أيضًا...
```

هكذا بقيت الشيفرة القديمة تعمل كما نريد، وأضفنا الخاصية الإضافية.

دورة تطوير واجهات المستخدم

ابدأ مسارك المهني كمطور واجهات المواقع والمتاجر الإلكترونية فور انتهائك من الدورة

التحق بالدورة الآن

8. الوراثة النموذجية

يتضمن هذا الفصل الأقسام التالية:

1. الوراثة النموذجية Prototypal inheritance
2. الوراثة النموذجية بتعمق F.prototype
3. النماذج الأولية الأصيلة Native prototypes
4. توابِع النماذج الأولية والكائنات بلا proto

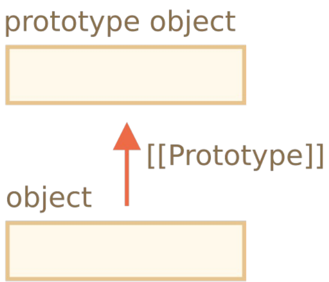
8.1 الوراثة النموذجية Prototypal inheritance

أثناء البرمجة، نرى دائما مواقف حيث تريد أخذ شيء وتوسعته أكثر. فمثلًا لدينا كائن مستخدم `user` له خاصيات وتوابِع، وأردنا إنشاء نسخ عنه (مدراء `admin` وضيوف `guest`) لكن معدّلة قليلًا. سيكون رائعًا لو أعدنا استعمال الموجود في كائن المستخدم بدل نسخه أو إعادة كتابة توابِعه، سيكون رائعًا لو صنعنا كائنًا جديدًا فوق كائن `user`.

الوراثة النموذجية (تدعى أيضًا الوراثة عبر كائن النموذج الأولي prototype) وهي الميزة الّتي تساعدنا في تحقيق هذا الأمر.

8.1.1 الخاصية [[Prototype]]

لكائنات جافاسكربت خاصية مخفية أخرى باسم `[[Prototype]]` (هذا اسمها في المواصفات القياسية للغة جافاسكربت)، وهي إمّا أن تكون `null` أو أن تشير إلى كائن آخر. نسمّي هذا الكائن بِـ"prototype" (نموذج أولي).



إن كائن النموذج الأولي "سحريٌّ" إن صحّ القول، فحين نريد قراءة خاصية من كائن `object` ولا يجدها محرّك جافاسكربت، يأخذها تلقائيًا من كائن النموذج الأولي لذاك الكائن. يُسمّى هذا في علم البرمجة "بالوراثة النموذجية" (Prototypal inheritance)، وهناك العديد من المزايا الرائعة في اللغة وفي التقنيات البرمجية مبنية عليها. الخاصية `[[Prototype]]` هي خاصية داخلية ومخفية، إلّا أنّ هناك طُرق عديدة لنراها. إحداها استعمال `__proto__` هكذا:

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};
rabbit.__proto__ = animal; // هنا
```

يعد `__proto__` هو الجالب والضابط القديم للخاصية `[[Prototype]]`، فكانت تستخدم قـديمًا، ولكن في اللغـة الحديثـة اسـتبدلت بالـدالتين `Object.getPrototypeOf/Object.setPrototypeOf` وهو أيضًا تعمل عمل الجالب والضابط للنموذج الأولي (سندرس هذه الدوال لاحقًا في هذا الفصل).

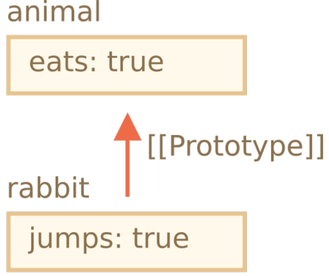
إن المتصفحات هي الوحيدة الّتي تدعم `__proto__` وفقًـا للمواصـفات القياسـية للغـة، ولكن في الواقـع جميع البيئات تدعمها حتى بيئات الخادم وذلك لأنها سهلة وواضحة. وهي الّتي سنستخدمها في الأمثلة.

فمثلاً لو بحثنا الآن عن خاصية ما في كائن `rabbit` ولم تكُ موجودة، ستأخذها لغة جافاسكربت تلقائيًا من كائن `animal`. مثال على ذلك:

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // (*)

// الآن: كلتا الخاصيتين في الأرنب
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true
```

هنا نضبط (في السطر `(*)`) كائن `animal` ليكون النموذج الأولي (Prototype) للكائن `rabbit`.

بعدها مـتى مـا حـاولت التعليمـة `alert` قـراءة الخاصـية `rabbit.eats` (انظـر `(**)`)، ولم يجـدها في كـائن `rabbit` سـتتبع لغـة جافاسـكربت الخاصـية `[[Prototype]]` لمعرفـة مـا هـو كـائن النمـوذج الأولي لكائن `rabbit`، وسيجده كائن `animal` (البحث من أسفل إلى أعلى):

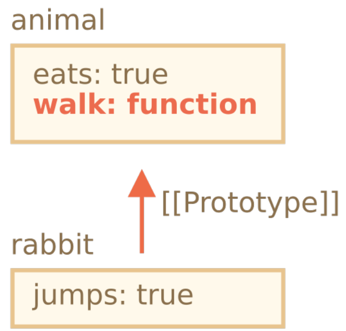


يمكن أن نقول هنا بأنّ الكائن `animal` هو النموذج الأولي للكائن `rabbit`، أو كائن `rabbit` هو نسخة نموذجية من الكائن `animal`. وبهذا لو كان للكائن `animal` خاصيات وتوابِع كثيرة مفيدة، تصير مباشرةً موجودة عند كائن `rabbit`. نسمّي هذه الخاصيات بأنّها "موروثة".

لو كان للكائن `animal` تابِعًا فيمكننا استدعائه في كائن `rabbit`:

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// نأخذ walk من كائن النموذج الأولي
rabbit.walk(); // Animal walk
```

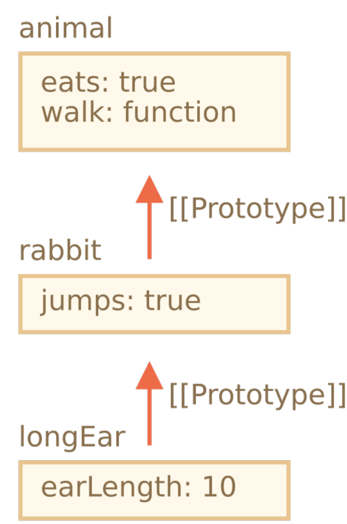
يُؤخذ التابع تلقائيًا من كائن النموذج الأولي، هكذا:



يمكن أيضًا أن تكون سلسلة الوراثة النموذجية (النموذج الأولي) أطول:

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
```

```
};

let rabbit = {
  jumps: true,
  __proto__: animal // (*)
};

let longEar = {
  earLength: 10,
  __proto__: rabbit // (*)
};

// نأخذ الدالة walk من سلسلة الوراثة النموذجية
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (من rabbit)
```



ولكن، هناك مُحددان للوراثة النموذجية وهما:

1. لا يمكن أن تكون سلسلة الوراثة النموذجية دائرية (على شكل حلقة). ما إن تُسند `__proto__` بطريقة دائرية فسترمي لغة جافاسكربت خطأً.

2. يمكن أن تكون قيمة `__proto__` إمّا كائنًا أو `null`، وتتجاهل لغة جافاسكربت الأنواع الأخرى.

ومن الواضح جليًا أيضًا أي كائن سيرث كائن `[[Prototype]]` واحد وواحد فقط، لا يمكن للكائن وراثة كائنين.

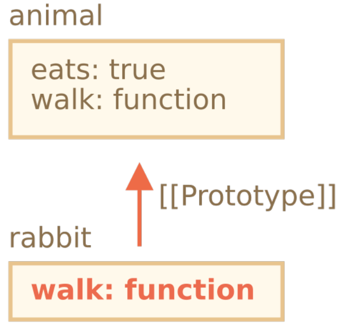
8.1.2 كائن النموذج الأولي للقراءة فقط

لا يمكننا تعديل أو حذف خصائص أو دوالّ من كائن النموذج الأولي وإنما هو للقراءة فقط. وأيّة عمليات كتابة أو حذف تكون مباشرةً على الكائن نفسه وليس على كائن النموذج الأولي.

في المثال أسفله نُسند التابع `walk` إلى الكائن `rabbit`:

```
let animal = {
  eats: true,
  walk() {
    /* لن يستعمل الكائن `rabbit` هذا التابع */
  }
};

let rabbit = {
  __proto__: animal
};

rabbit.walk = function() {
  alert("Rabbit! Bounce-bounce!");
};

rabbit.walk(); // Rabbit! Bounce-bounce!
```

من الآن فصاعدًا فستجد استدعاء التابع `rabbit.walk()` سيكون من داخل كائن `rabbit` مباشرةً وتُنفّذه دون استعمال كائن النموذج الأولي:



ولكن خاصيات الوصول استثناء للقاعدة، إذ يجري الإسناد على يد دالة الضابِط، أي أنّك بالكتابة في هذه الخاصية في الكائن الجديد ولكنّك استدعيت دالة الضابط الخاصة بكائن النموذج الأولي لإسناد هذه القيمة.

لهذا السبب نرى الخاصية `admin.fullName` في الشيفرة أسفله تعمل كما ينبغي لها:

```
let user = {
  name: "Ahmad",
  surname: "Smith",

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  },

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

let admin = {
  __proto__: user,
  isAdmin: true
};

alert(admin.fullName); // Ahmad Smith (*)

// عمل الضابط!
admin.fullName = "Alice Cooper"; // (**)
```

هنا في السطر `(*)` نرى أن `admin.fullName` استدعت الجالِب داخل الكائن `user`، ولهذا استُدعيت الخاصية. وفي السطر `(**)` نرى عملية إسناد للخاصية `admin.fullName` ولهذا استدعيَ الضابِط داخل الكائن `user`.

8.1.3 ماذا عن "this"؟

بعدما تتمعّن في المثال أعلاه، يمكن أن تتساءل ما قيمة `this` داخل `set fullName(value)`؟ أين كُتبت القيم الجديدة `this.name` و `this.surname`؟ داخل الكائن `user` أم داخل الكائن `admin`؟

جواب هذا السؤال المحيّر بسيط: لا تؤثّر كائنات النموذج الأولي على قيمة `this`.

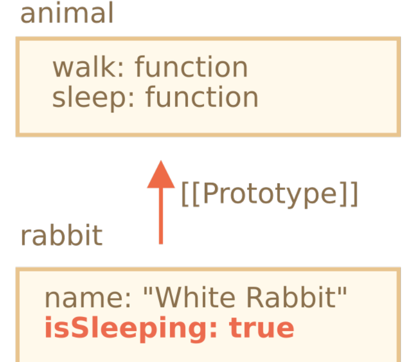
أينما كان التابع موجودًا أكان في الكائن أو في كائن النموذج الأولي، سيكون تأثير `this` على الكائن الّذي قبل النقطة (الكائن المستدعى من خلاله هذه الخاصية) دائمًا وأبدًا. لهذا فالضابِط الّذي يستدعي `admin.fullName=` يستعمل كائن `admin` عوضًا عن `this` وليس الكائن `user`.

في الواقع فهذا أمر مهما جدًا جدًا إذ أنّ لديك ربما كائنًا ضخمًا فيه توابِع كثـيرة جـدًا، وهنـاك كائنـات أخـرى ترثه، وما إن تشغّل تلك الكائنات الموروثة التوابعَ الموروثة، ستعدّل حالتها هي -أي الكائنات- وليس حالة الكـائن الضخم ذاك.

فمثلًا هنـا، يمثّـل كـائن animal "مخـزّنَ توابِـع" وكـائن rabbit يسـتغلّ هـذا المخـزن. فاسـتدعاء rabbit.sleep() يضبط this.isSleeping على كائن rabbit:

```
// للحيوان توابِع
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};

let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

// rabbit.isSleeping يعدّل
rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // غير معرّف (لا يوجد خاصية معرفة في كائن النموذج الأولي بهذا الاسم)
```

الصورة الناتجة:



لو كانت هناك كائنات أخرى (مثل الطيور bird والأفاعي snake وغيرها) ترث الكائن animal، فسيمكنها الوصول إلى توابِع الكائن animal، إلّا أنّ قيمة this في كلّ استدعاء للتوابِع سيكون على الكائن الّذي استُدعيت منه، وستعرِفه لغة جافاسكربت أثناء الاستدعاء (أي سيكون الكائن الّذي قبل النقطة) ولن يكون animal. لذا متى كتبنا البيانات من خلال this، فستُخزّن في تلك الكائنات الّتي استدعيت عليها this.

وبهذا نخلص إلى أنّ التوابع مشتركة، ولكن حالة الكائن ليست مشتركة.

8.1.4 حلقة for..in

كما أنّ حلقة for..in تَمرُّ على الخاصيات الموروثة هي الأخرى. مثال:

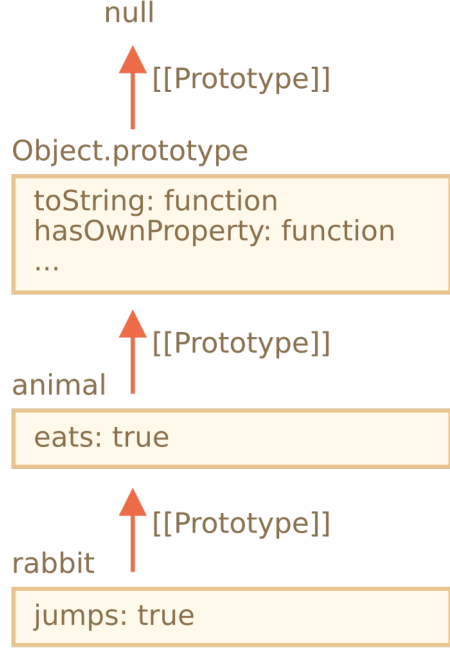
```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// يُعيد التابع Object.keys خصائص الكائن نفسه فقط
alert(Object.keys(rabbit)); // jumps

// تدور حلقة for..in على خصائص الكائن نفسه والخصائص الموروثة معًا
for(let prop in rabbit) alert(prop); // jumps ثمّ eats
```

لو لم تكن هذه النتيجة ما نريد (أي نريد استثناء الخاصيات الموروثة)، فيمكن استعمال التابع obj.hasOwnProperty(key) المضمّن في اللغة: إذ يُعيد true لو كان للكائن obj نفسه (وليس للموروث منه) خاصية بالاسم key.

بهذا يمكننا ترشيح الخاصيات الموروثة (ونتعامل معها على حدة):

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

for(let prop in rabbit) {
  let isOwn = rabbit.hasOwnProperty(prop);

  if (isOwn) {
    alert(`Our: ${prop}`); // تخصّنا: jumps
  } else {
    alert(`Inherited: ${prop}`); // ورثناها: eats
  }
}
```

هنا نرى سلسلة الوراثة الآتية: يرث كائن `rabbit` كائنَ `animal`، والّذي يرثه هكذا `Object.prototype` (إذ أنّه كائن مجرّد `{...}`، وهذا السلوك المبدئي)، وبعدها يرث `null`:



ملاحظة لطيفة في هذا السياق وهي: من أين أتى التابع `rabbit.hasOwnProperty`؟ لم نعرّفه يدويًا! لو تتبّعناه في السلسلة لرأينا بأنّ كائن النموذج الأولي `Object.prototype.hasOwnProperty` هـو من قـدّم التابع، أي بعبارة أخرى، ورث كائن `rabbit` هذا التابع من كائن النموذج الأولي.

ولكن لحظة… لماذا لم يظهـر تـابع `hasOwnProperty` في حلقـة `for..in` كمـا ظهـرت `eats` و `jumps` طالما تُظهر حلقات `for..in` الخاصيات الموروثة؟

الإجابة هنا بسيطة أيضًا: لإنه مُنع من قابلية العدّ (من خلال إسناده لقيمـة الرايـة `enumerable:false`). في النهاية هي مِثل غيرهـا من الخاصـيات في `Object.prototype`- تملـك الرايـة `enumerable:false`، وحلقـة `for..in` لا تمـرّ إلّا على الخاصـيات القابلـة للعـدّ. لهـذا السـبب لم نراهـا لا هي ولا خاصـيات `Object.prototype` الأخرى.

كلّ التوابع الّتي تجلب المفتاح/القيمة تُهمل الخاصيات الموروثة، تقريبًا مثل تـابِع `Object.keys` أو تـابِع `Object.values` ومـا شـابههم. إذ إنهم يتعـاملون مـع خصـائص الكـائن نفسـه ولا يأخـذون بعين الاعتبـار الخصائص الموروثة

8.1.5 الخلاصة

- لكلّ كائنات جافاسكربت خاصية `[[Prototype]]` مخفية قيمتها إمّا أحد الكائنات أو `null`.
- يمكننا استعمال `obj.__proto__` للوصول إلى هذه الخاصية (وهي خاصـية جـالِب/ضـابِطة). هنـاك طرق أخرى سنراها لاحقًا.
- الكائن الّذي تُشير إليه الخاصية `[[Prototype]]` يسمّى كائن النموذج الأولي.
- لو أردنا قراءة خاصية داخل كائن مـا `obj` أو اسـتدعاء تـابِع، ولم تكن موجـودة/يكن موجـودًا، فسـيحاول محرّك جافاسكربت البحث عنه/عنها في كائن النموذج الأولي.
- عمليات الكتابة والحذف تتطبّق مباشرة على الكائن المُسـتدعي ولا تسـتعمل كـائن النمـوذج الأولي (إذ يعدّ أنّها خاصية بيانات وليست ضابِطًا).
- لو استدعينا التابع `obj.method()` وأخذ المحرّك التابع `method` من كائن النموذج الأولي، فلن تتغـير إشارة `this` وسيُشير إلى `obj`، أي أنّ التوابع تعمل على الكـائن الحـالي حتّى لـو كـانت التوابـع نفسـها موروثة.
- تمـرّ حلقـة `for..in` على خاصـيات الكـائن والخاصـيات الموروثـة، بينمـا لا تعمـل توابِـع جلب المفاتيح/القيم إلّا على الكائن نفسه.

8.1.6 تمارين

أ. العمل مع prototype

الأهمية: ★★★★★

إليك شيفرة تُنشئ كائنين وتعدّلها. ما القيم الّتي ستظهر في هذه العملية؟

```
let animal = {
  jumps: null
};
let rabbit = {
  __proto__: animal,
  jumps: true
};

alert( rabbit.jumps ); // ? (1)

delete rabbit.jumps;

alert( rabbit.jumps ); // ? (2)

delete animal.jumps;

alert( rabbit.jumps ); // ? (3)
```

يجب أن هنالك ثلاث إجابات.

الحل:

1. `true`، تأتي من `rabbit`.
2. `null`، تأتي من `animal`.
3. `undefined`، إذ ليس هناك خاصية بهذا الاسم بعد الآن.

ب. خوارزمية بحث

الأهمية: ★★★★★

ينقسم هذا التمرين إلى قسمين.

لديك الكائنات التالية:

```
let head = {
  glasses: 1
};

let table = {
  pen: 3
};

let bed = {
  sheet: 1,
  pillow: 2
};

let pockets = {
  money: 2000
};
```

1. استعمل `__proto__` لإسناد كائنات النموذج الأولي بحيث يكون البحث عن الخاصيات بهذه الطريقة: `pockets` ثمّ `bed` ثمّ `table` ثمّ `head` (من الأسفل إلى الأعلى على التتالي). فمثلًا، قيمة `pockets.pen` تكون 3 (من `table`)، وقيمة `bed.glasses` تكون 1 (من `head`).
2. أجِب عن هذا السؤال: ما الأسرع، أن نجلب `glasses` هكذا `pockets.glasses` أم هكذا `head.glasses`؟ قِس أداء كلّ واحدة لو لزم.

الحل:

لنُضيف خاصيات `__proto__`:

```
let head = {
  glasses: 1
};

let table = {
  pen: 3,
  __proto__: head
};
```

```
let bed = {
  sheet: 1,
  pillow: 2,
  __proto__: table
};

let pockets = {
  money: 2000,
  __proto__: bed
};

alert( pockets.pen ); // 3
alert( bed.glasses ); // 1
alert( table.money ); // undefined
```

حين نتكلّم عن المحرّكات الحديثة، فليس هناك فرق (من ناحية الأداء) لو أخذنا الخاصية من الكائن أو من النموذج الأولي، فهي تتذكّر مكان الخاصية وتُعيد استعمالها عند طلبها ثانيةً.

فمثلًا ستتذكّر التعليمة `pockets.glasses` بأنّها وجدت `glasses` في كائن `head`، وفي المرة التالية ستبحث هناك مباشرة. كما أنّها ذكية لتُحدّث ذاكرتها الداخلية ما إن يتغيّر شيء ما لذا فإن الأداء الأمثل في أمان.

ج. أين سيحدث التعديل؟

الأهمية: ★★★★★

لدينا الكائن `rabbit` يرث من الكائن `animal`. لو استدعينا `rabbit.eat()` فأيّ الكائنين ستُعدل به الخاصية `full`، الكائن `animal` أم الكائن `rabbit`؟

```
let animal = {
  eat() {
    this.full = true;
  }
};
let rabbit = {
  __proto__: animal
};
rabbit.eat();
```

الحل:

الإجابة هي: الكائن `rabbit`.

لأنّ قيمة `this` هي الكائن قبل النقطة، بذلك يُعدّل `rabbit.eat()`.

عملية البحث عن الخاصيات تختلف تمامًا عن عملية تنفيذ تلك الخاصيات.

نجد التابع `rabbit.eat` سيُستدعى أولًا من كائن النموذج الأولي، وبعدها نُنفّذه على أنّ `this=rabbit`.

د. لماذا أصابت التخمة كِلا الهامسترين؟

الأهمية: ★★★★★

لـديناهامسـترين، واحـد سـريع `speedy` وآخـر كسـول `lazy`، والاثـنين يرثـان كـائن الهامسـتر العمومي `hamster`. حين نُعطي أحدهما الطعام، نجد الآخر أُتخم أيضًا. لماذا ذلك؟ كيف نُصلح المشكلة؟

```
let hamster = {
  stomach: [],

  eat(food) {
    this.stomach.push(food);
  }
};

let speedy = {
  __proto__: hamster
};

let lazy = {
  __proto__: hamster
};
// وجد هذا الهامستر الطعامَ قبل الآخر
speedy.eat("apple");
alert( speedy.stomach ); // apple
// هذا أيضًا وجده. لماذا؟ أصلِح الشيفرة.
alert( lazy.stomach ); // apple
```

الحل:

لنرى ما يحدث داخل الاستدعاء `speedy.eat("apple")` بدقّة.

1. نجد التابع `speedy.eat` في كائن النموذج الأولي الهامستر (`hamster=`)، وبعدها ننفّذه بقيمة `this=speedy` (الكائن قبل النقطة).
2. بعدها تأتي مهمة البحث للتابِع `this.stomach.push()` ليجد خاصية المعدة `stomach` ويستدعي عليها `push`. يبدأ البحث عن `stomach` في `this` (أي في `speedy`)، ولكنّه لا يجد شيئًا.
3. بعدها يتبع سلسلة الوراثة ويجد المعدة `stomach` في `hamster`.
4. ثمّ يستدعي `push` عليها ويذهب الطعام في معدة النموذج الأولي.

بهذا تتشارك الهامسترات كلها معدةً واحدة!

أكان `lazy.stomach.push(...)` أم `speedy.stomach.push()`، لا نجد خاصية المعدة `stomach` إلّا في كائن النموذج الأولي (إذ ليست موجودة في الكائن نفسه)، بذلك ندفع البيانات الجديدة إلى كائن النموذج الأولي. لاحظ كيف أنّ هذا لا يحدث لو استعملنا طريقة الإسناد البسيط `this.stomach=`:

```
let hamster = {
  stomach: [],

  eat(food) {
    // نُسند إلى this.stomach بدلًا من this.stomach.push
    this.stomach = [food];
  }
};

let speedy = {
   __proto__: hamster
};

let lazy = {
  __proto__: hamster
};

// وجد الهامستر السريع الطعام
speedy.eat("apple");
alert( speedy.stomach ); // apple
// معدة ذاك الكسول فارغة
alert( lazy.stomach ); // <لا شيء>
```

الآن يعمل كلّ شيء كما يجب، إذ لا تبحث عملية الإسناد `this.stomach=` عن خاصية `stomach`، بل تكتبها مباشرةً في كائن الهامستر الّذي وجد الطعام (المستدعى قبل النقطة).

ويمكننا تجنّب هذه المشكلة من الأساس بتخصيص معدة لكلّ هامستر (كما الطبيعي):

```
let hamster = {
  stomach: [],

  eat(food) {
    this.stomach.push(food);
  }
};

let speedy = {
  __proto__: hamster,
  stomach: [] // هنا
};

let lazy = {
  __proto__: hamster,
  stomach: [] // هنا
};

// وجد الهامستر السريع الطعام
speedy.eat("apple");
alert( speedy.stomach ); // apple

// معدة ذاك الكسول فارغة
alert( lazy.stomach ); // <لا شيء>
```

يكون الحلّ العام هو أن تُكتب الخاصيات كلّها الّتي تصف حالة الكائن المحدّد ذاته (مثل `stomach` أعلاه) - أن تُكتب في الكائن ذاته، وبهذا نتجنّب مشاكل تشارك المعدة.

8.2 الوراثة النموذجية بتعمق: F.prototype

لا تنسَ بأنّك يمكنك إنشاء كائنات جديدة من خلال دالّة الباني (مثل `new F()`). لو كـان `F.prototype` كائن جافاسكربت، فإن المعامِل `new` سيضبط الخاصية `[[Prototype]]` لهذا لكائن الجديد.

من بدايـة تضـمين لغـة جافاسـكربت للوراثـة النموذجيـة جعلتهـا من الممـيزات الأساسـية في اللغـة. ولكن في الماضي لم يكن هنالك القدرة للوصول المباشر للوراثة النموذجية والطريقة الوحيدة الـّتي حلّت محلهـا هي خاصية `"prototype"` في دالّة الباني. سنشرح في هذا الفصل كيفية اسـتخدامها لأنـه مـازال العديـد من الشيفرات البرمجية القديمة تستخدمها.

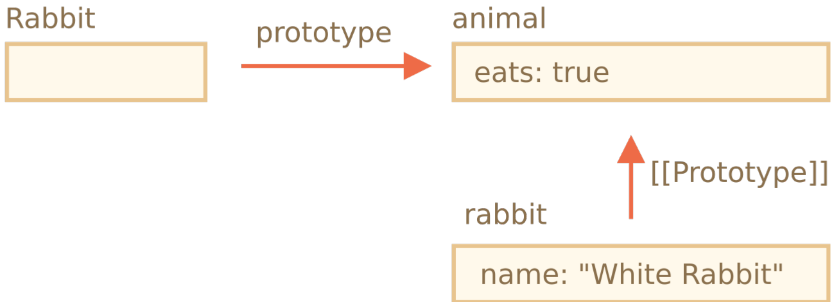
لاحظ بأنّ `F.prototype` هنا تعني وجود خاصية عادية باسم `"prototype"` للكائن `F`. ربما تفكّـر وكأنهـا النموذج الأولي لهذا الكائن، ولكن لا… فهنا نعني حرفيًا أنها خاصية عادية لها هذا الاسم.

إليك مثالًا:

```
let animal = {
  eats: true
};

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = animal; // هنا

let rabbit = new Rabbit("White Rabbit"); //  rabbit.__proto__ ==
animal

alert( rabbit.eats ); // true
```

تعني التعليمة `Rabbit.prototype = animal` حرفيًا الآتي: "ما إن يُنشأ كائن `new Rabbit`، أسنِد خاصية `[[Prototype]]` له لتكون للكائن `animal`".

إليك الصورة الناتجة:



في الصورة نرى `"prototype"` في سهم أفقي (أي أنّها خاصية عادية) بينما `[[Prototype]]` في سهم رأسي (أي أنّها توضّح وراثة كائن `rabbit` للكائن `animal`).

إن الخاصية `F.prototype` تستخدم عند الإنشاء فقط أي عندما تستدعى تعليمة `new F` وتُسند للكائن القيمة المناسبة للخاصية `[[Prototype]]`. في حال تغيرت الخاصية `F.prototype` مثلًا:

```
F.prototype = <another object>
```

عندها ستحصل الكائنات المنشأة بعد هذا التغيير على القيمة الجديدة للخاصية `[[Prototype]]` (أي الكائن الجديد)، ولكن الكائنات القديمة ما زالت تحتفظ بالقيمة القديمة.

8.2.1 القيمة الافتراضية للخاصية prototype في الباني

لكلّ دالة خاصية `"prototype"` حتّى لو لم نقدّمها نحن. إن القيمة الافتراضية للخاصية `"prototype"` تُشير إلى نفس الدالّة. هكذا تمامًا:

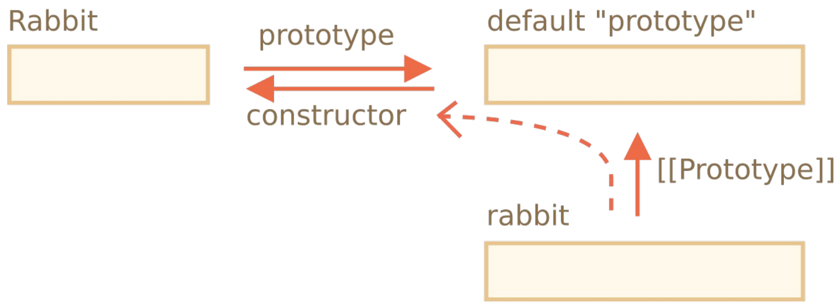
```
function Rabbit() {}
/* prototype كائن
Rabbit.prototype = { constructor: Rabbit };
*/
```

يمكننا فحص ذلك أيضًا:

```
function Rabbit() {}
// مبدئيًا:
// Rabbit.prototype = { constructor: Rabbit }
alert( Rabbit.prototype.constructor == Rabbit ); // true
```

طبيعيًا، إن لم نعدل أي شيء، ستكون خاصية `constructor` مُتاحة لكلّ كائنات `rabbit` من خلال كائن `[[Prototype]]`:

```
function Rabbit() {}
// مبدئيًا:
// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // {constructor: Rabbit} ترث من

alert(rabbit.constructor == Rabbit); // prototype من true
```



يمكننا استعمال الخاصية `constructor` لإنشاء كائن جديد باستعمال نفس الباني الّذي أنشأ الكائن الموجود حاليًا. هكذا:

```
function Rabbit(name) {
  this.name = name;
  alert(name);
}

let rabbit = new Rabbit("White Rabbit");

// انظر
let rabbit2 = new rabbit.constructor("Black Rabbit");
```

يُفيدنا هذا حين نكون أمام كائن ولكن لا نعرف الباني الحقيقي الّذي بناه (ربما أتى من مكتبة خارجية)، وأردنا إنشاء كائن آخر مثله.

ولكن الأمر الأهم الّذي يتعلّق بِـ `"constructor"` هو أنّ لغة جافاسكربت نفسها لا تتأكّد من صحّة قيمة خاصية `"constructor"`.

نعم كما قرأت، الخاصية موجودة في "prototype" للدوالّ، وهذا كلّ ما في الأمر. إذ ستعتمد لغة جافاسكربت علينا فيما سيحدث لاحقًا.

فمثلًا لو أردنا استبدال القيمة الافتراضية للخاصية prototype، فلن يملك الكائن أيّ خاصية "constructor". مثال:

```
function Rabbit() {}
Rabbit.prototype = {
  jumps: true
};

let rabbit = new Rabbit();
// لاحظ
alert(rabbit.constructor === Rabbit); // false
```

ولهذا لنُبقي على خاصية "constructor" الصحيحة يمكننا إضافة الخاصيات وإزالتها من كائن "prototype" الافتراضي بدل الطريقة السابقة. هكذا:

```
function Rabbit() {}

// Rabbit.prototype بدل الكتابة على كلّ
// نُضيف ما نريد إليه
Rabbit.prototype.jumps = true
// هكذا تبقى خاصية Rabbit.prototype.constructor الإفتراضية محفوظة
```

أو يمكننا (لو أردنا) إعادة إنشاء الخاصية constructor يدويًا:

```
Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit // هنا
};

// الآن سيكون المُنشِئ صحيحًا إذ أنّا من أضفناه
```

8.2.2 الخلاصة

شرحنا في هذا الفصل سريعًا طريقة ضبط كائن `[[Prototype]]` للكائنات الّتي أنشأتها بدالّة الباني. سنرى لاحقًا أنماط متقدّمة في البرمجة تعتمد على هذا الطريقة.

ما أخذناه بسيط، ولكن بعض الأمور نوضّحها ثانيةً للتأكّد:

- تضبط الخاصية `F.prototype` (لا تظنّها كائن `[[Prototype]]`) لكائنٍ ما الخاصية `[[Prototype]]` لكلّ الكائنات الجديدة متى استدعيت `new F()`.
- يجب أن تكون قيمة `F.prototype` إمّا كائنًا أو `null`، ولن تعمل أيّة قيم أخرى.
- هذا التأثير للخاصية `"prototype"` موجود فقط حين يُضبط في دالة الباني وحين يُنفّذ بتعليمة `new`.

في الكائنات العادية ليست بخاصية خاصة جدًا:

```
let user = {
  name: "Ahmad",
  prototype: "Bla-bla" // نزعنا السحر
};
```

لكلّ الدوال مبدئيًا `F.prototype = { constructor: F }`، فيمكننا أن نأخذ باني معين من كائن ما بالدخول إلى الخاصية `"constructor"` الخاصة به.

8.2.3 تمارين

أ. تغيير الخاصية prototype

الأهمية: ★★★★★

أنشأنا في الشيفرة أدناه كائنًا جديدًا `new Rabbit` وحاولنا بعدها تعديل الخاصية prototype لهذا الكائن.

بادئ ذي بدء، كانت الشيفرة:

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();
alert( rabbit.eats ); // true
```

وأضفنا سلسلة نصية أخرى (عليها علامة). أولًا، ماذا سيعرض التابع `alert`؟

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

Rabbit.prototype = {}; // (*)

alert( rabbit.eats ); // ?
```

ثانيًا، ماذا لو… كانت الشيفرة كهذه (استبدلنا سطرًا فيها)؟

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

Rabbit.prototype.eats = false; // (*)

alert( rabbit.eats ); // ?
```

ثالثًا، ماذا عن هذه (استبدلنا سطرًا أيضًا)؟

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

delete rabbit.eats; // (*)

alert( rabbit.eats ); // ?
```

رابعًا، وهذه... أيضًا:

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

delete Rabbit.prototype.eats; // (*)

alert( rabbit.eats ); // ?
```

الحل:

الإجابات:

1. `true`: عمليــة الإســناد على `Rabbit.prototype` تضــع الخاصــية `[[Prototype]]` للكائنــات الجديدة، ولكنّها لا تعدّل على الكائنات الموجودة مسبقًا.
2. `False`: عملية الإسناد تكون من خلال الخاصية `Rabbit.prototype`، إن الخاصية المشار إليها هنـا `Rabbit.prototype` ليســت مكــررًا، وإنمــا بقيت يُشــار إليهــا من خلال `Rabbit.prototype` والخاصية `[[Prototype]]` للكائن `rabbit`.

   لذا حين نغيّر المحتوى في الطريقة الأولى سنرى النتائج في الطريقة الثانية.
3. `True`: كل عمليات الحذف تطبق مباشرة على الكائن. تحاول هذه التعليمة `delete rabbit.eats` حذف الخاصية المخصصة للكائن `rabbit` ولكنها ليست لها. لذا العملية لن يكون لها أي تأثير.
4. `Undefined`: حُذفت الخاصية `eats` من كائن prototype وما عادت موجودة بعد الآن.

ب. إنشاء كائن جديد من خلال نفس باني لكائنٍ آخر

الأهمية: ★★★★★

تخيّل بأنّ لدينا الكائن الفريد `obj` وأنشأته بدالة الباني، ولكننا... لا نعرف أيّ دالة هـذه، ولكن مـع ذلـك نريـد استعمال نفس الباني لإنشاء كائن جديد آخر.

أيمكن لهذه الشيفرة إنجاز المهمة؟

```
let obj2 = new obj.constructor();
```

اكتب مثالين باستخدام بانيين للكائن `obj`، واحدًا يعمل مع الشيفرة أعلاه، وواحدًا لا يعمل له.

الحل:

يمكن أن نستعمل هذه الطريقة لو كنّا متأكدين مئة بالمئة بأنّ خاصية `"constructor"` تحمل القيمة الصحيحة. فمثلًا لو لم نعدّل على `"prototype"` المبدئية فستعمل هذه الشيفرة بلا ريب:

```
function User(name) {
  this.name = name;
}

let user = new User('Ahmad');
let user2 = new user.constructor('Pete');

alert( user2.name ); // Pete (عملت!)
```

نفذت الشيفرة تنفيذًا صحيحًا إذ أنّ `User.prototype.constructor == User`.

ولكن... لو أتى أحدهم مثلًا وكتب على `User.prototype` ونسي إعادة إنشاء `constructor` لتُشير إلى كائن المستخدم `User`، فلن تعمل الشيفرة.

مثال:

```
function User(name) {
  this.name = name;
}
User.prototype = {}; // (*)

let user = new User('Ahmad');
let user2 = new user.constructor('Pete');

alert( user2.name ); // undefined
```

لمَ قيمة `user2.name` هي `undefined`؟

إليك طريقة عمل تعليمة `new user.constructor('Pete')`:

1. أولًا، تبحث عن المُنشِئ `constructor` داخل `user`، ولا تجده.

2. ثمّ تتبع سلسلة prototype وتجد prototype الكائن `user` هو `User.prototype`، وأيضًا لا تجده.
3. قيمة `User.prototype` مـا هي إلّا كائنًـا فارغًـا `{}`، و قيمـة الخاصـية prototype لهـذا الكـائن هي `Object.prototype`، وهنا وجدنا `Object.prototype.constructor == Object` بـذلك استعملناه.

وفي نهاية الأمر، لدينا التعليمة `let user2 = new Object('Pete')` إذ أنّ الباني الخاص بالكائن `Object` يتجاهل الوسطاء وينشئ دائمًا كائنًا فارغًا. بطريقة مشابهة جدًا للتعليمة `let user2 = {}` والّتي أنشأت لنا الكائن `user2` في نهاية الأمر.

8.3 النماذج الأولية الأصيلة Native prototypes

كثيرٌ من الكائنات تستعمل الخاصية `"prototype"`، حتّى في محرّك جافاسكربت، إذ تستعملها كلّ البواني المضمّنة في اللغة.

لنرى أولًا تفاصيلها وبعدها كيفية استعمالها لإضافة مزايا جديدة إلى الكائنات المضمّنة.

8.3.1 Object.prototype

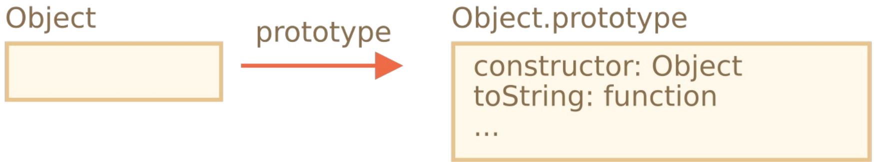
لنقل بأنّا طبعنا كائنًا فارغًا:

```
let obj = {};
alert( obj ); // "[object Object]" ?
```

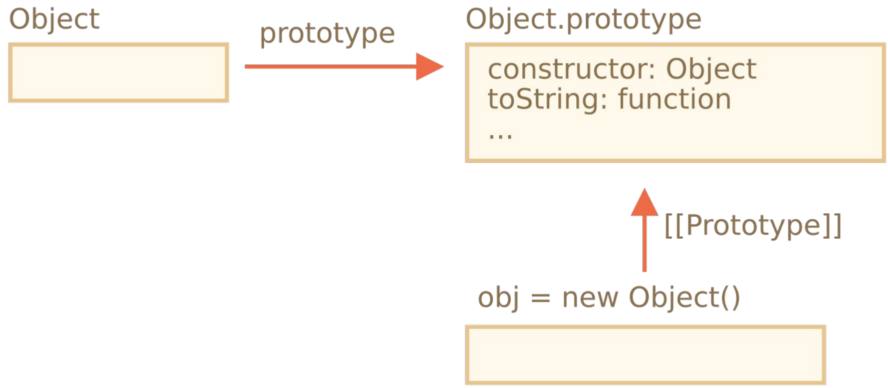
ما هذه الشيفرة الّتي ولّدت النصّ `"[object Object]"`؟ هذه أفعال تابِع `toString` المضمّن في اللغة، ولكن أين هذا التابع فكائن `obj` فارغ!

ولكن لو فكّرنا لحظة... فالاختصار هذا `obj = {}` هو كأنما كتبنا `obj = new Object()`، وهنا `Object` هو الباني المضمّن في اللغة يُشير الخاصية prototype في الكائن إلى كائن آخر ضخم فيه التابع `toString` وغيره من توابِع.

هذا ما يحدث:



متى استدعينا `new Object()` (أو أنشأنا كائن مجرّد `{...}`)، ضُبطت الخاصية `[[Prototype]]` لذاك الكائن إلى `Object.prototype` طبقًا للقاعدة الّتي تحدّثنا عنها في الفصل السابق:



لــذا مــتى حــدث اســتدعاء إلى `obj.toString()` أخــذت لغــة جافاســكربت التــابعَ من `Object.prototype`. يمكننا التأكّد من هذا هكذا:

```
let obj = {};

alert(obj.__proto__ === Object.prototype); // true
// obj.toString === obj.__proto__.toString == Object.prototype.toString
```

لاحظ أنّ لم تعد هناك كائنات `[[Prototype]]` في السلسلة فوق `Object.prototype`:

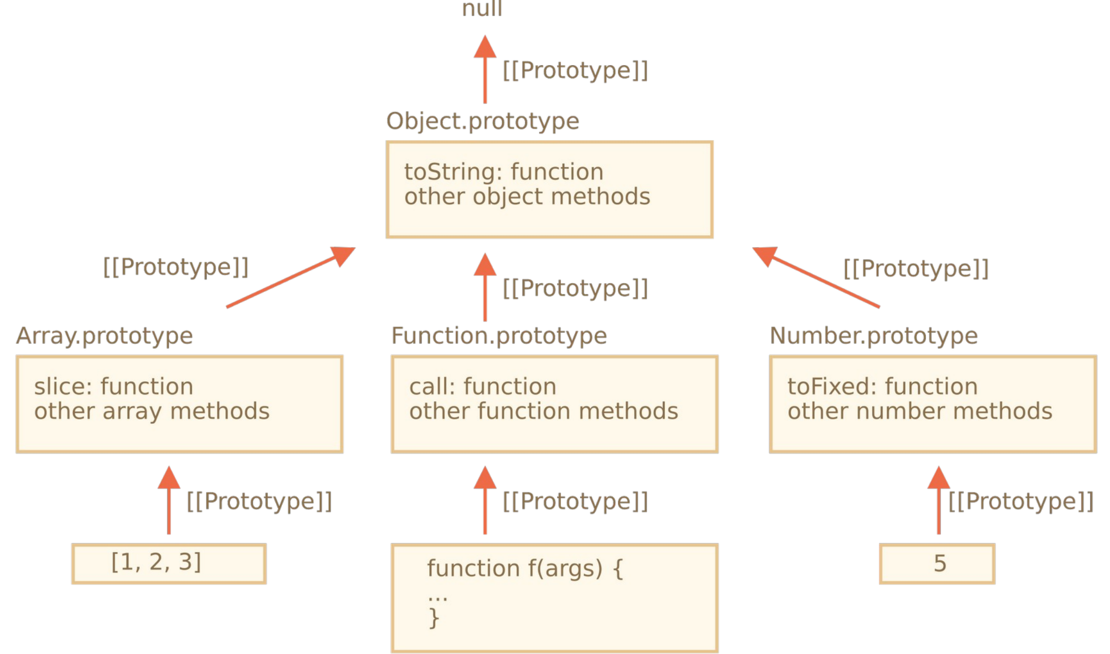
```
alert(Object.prototype.__proto__); // null
```

8.3.2 كائنات النماذج الأولية الأخرى المضمّنة في اللغة

الكائنات الأخرى مثل المصفوفات `Array` والتواريخ `Date` والدوال `Function` تضع هي الأخرى توابِعها في كائنات النماذج الأولية prototype. فمثلًا، حين نُنشئ المصفوفة `[1, 2, 3]` تستدعي لغة جافاسكربت داخليًا البـاني `new Array()` بنفسـها، بـذلك يصـير كـائن `Array.prototype` كـائنَ النمـوذج الأولي (prototype) ويقدّم له التوابع اللازمة. هذا الأمر يزيد من كفاءة الذاكرة.

بحسـب المواصـفات القياسـية للغـة، أنّ لكـلّ كائنـات النمـاذج الأوليـة (prototype) المضـمّنة كـائنَ `Object.prototype` آخر فوقها، ولهذا يقول الناس بأنّ "كلّ شيء يرث الكائنات (Objects)".

إليك صورة تصف هذا كله:



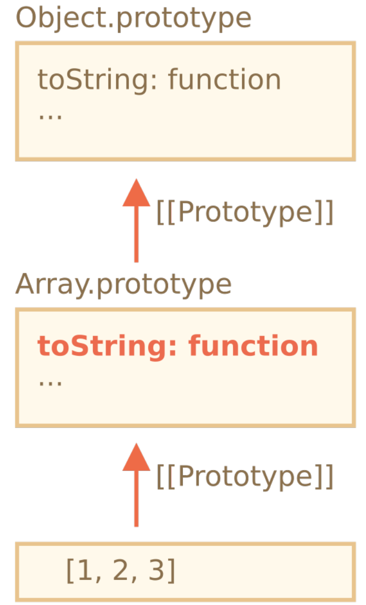
لنرى أمر كائنات النماذج الأولية prototype يدويًا:

```
js runlet arr = [1, 2, 3];

// هل ترث Array.prototype؟
alert( arr.__proto__ === Array.prototype ); // true

// ثمّ من Object.prototype؟
alert( arr.__proto__.__proto__ === Object.prototype ); // true

// وفوق هذا كلّه null.
alert( arr.__proto__.__proto__.__proto__ ); // null
```

أحيانًا تتداخل التوابع في كائنات النماذج الأولية (prototype) مع بعضها. فمثلًا للكائن `Array.prototype` تابِعًا خاصًا فيه `toString` يعرض العناصر بينها فاصلة:

```
let arr = [1, 2, 3]
alert(arr); // 1,2,3 <-- ناتج Array.prototype.toString
```

كما رأينا سابقًا فللكائن `Object.prototype` تابِع `toString` أيضًا، ولكنّ `Array.prototype` أقرب في سلسلة وراثة النموذج الأولي prototype وبذلك تستعمل لغة جافاسكربت تابِع المصفوفة لا الكائن.



كما أنّ الأدوات في المتصفّحات (مثل طرفية كروم للمطوّرين) تعرض الوراثة (إن التعليمة `console.dir` ربّما سنحتاجها للكائنات المضمنة في اللغة).

```
> console.dir([1,2,3])
▼ Array[3] ℹ
    0: 1
    1: 2
    2: 3
    length: 3
  ▼ __proto__: =Array.prototype
    ▶ concat: function concat() { [native code] }
    ▶   ...
    ▶ unshift: function unshift() { [native code] }
    ▼ __proto__: =Object.prototype
      ▶   ...
      ▶ constructor: function Object() { [native code] }
      ▶ hasOwnProperty: function hasOwnProperty() { [native code] }
      ▶ isPrototypeOf: function isPrototypeOf() { [native code] }
      ▶   ...
```

كما أنّ الكائنات الأخرى المضمّنة في اللغة تعمل بنفس الطريقة. حتى الدوالّ هم كائنات مبنية من خلال البواني المخصصة للدوالّ والمضمّنة في اللغة والدوالّ الخاصة بها (مثل الاستدعاء(`call`)/التطبيق(`apply`) وغيرهم من الدوال) مأخوذة من النموذج الأولي للدوالّ `Function.prototype`. ولديهم دالّة `toString` أيضًا، انظر:

```
function f() {}

alert(f.__proto__ == Function.prototype); // true
alert(f.__proto__.__proto__ == Object.prototype); // true، إذ ترث الكائنات
```

8.3.3 الأنواع الأولية

حتّى هنا لا تعقيد، إلّا حين نتعامل مع السلاسل النصية والأعداد والقيم المنطقية، عندها نرى التعقيدات تبدأ. كما نتذكّر من فصول سابقة، فهذه الأنواع ليست كائنات، ولكن لو حاولنا الوصول إلى خاصياتها فسنرى كائنات تغليف أُنشئت مؤقتًا باستعمال البواني المضمّنة في اللغة `String` و `Number` و `Boolean`، وهذه الكائنات تقدم ما نريد من توابع وتختفي.

نرى هذه الكائنات مؤقّتًا إذ تُصنع سريعًا دون معرفتنا!

> لا تملك القيمتان null و undefined أي كائنٍ مغلّف لها، وليس لديها أي دوال ولا خاصيات ولا حتى نماذج أولية.

8.3.4 تغيير كائنات النماذج الأولية الأصيلة

يمكن تعديل كائنات النماذج الأولية الأصيلة. فمثلًا يمكننا إضافة تـابِع إلى `String.prototype` فيصـبح متاحًا لكلّ السلاسل النصية:

```
String.prototype.show = function() {
  alert(this);
};

"BOOM!".show(); // BOOM!
```

يمكن أن يراود المرء (وهو يطوّر) أفكار جديدة لتكـون توابِـع مضـمّنة، ويريـد إضـافتها، بـل نريـد ونتـوق إلى إضافتها إلى كائنات النماذج الأولية الأصيلة، إلّا أنّ هذه (وبصفة عامة) فكرة سيّئة جدًا.

> بما أنّ النماذج الأولية نماذج عامة فمن السهل أن يحدث تضارب. إذ تأتي مكتبتين تُضيفان التابع `String.prototype.show` وتكتب واحدة على تابِع الأُخرى دون قصد، لهذا تعدّ عملية تعديل كائنات النماذج الأولية الأصيلة على أنّها فكرة سيئة.

أمّا في البرمجة الحديثة، فما من طريقة مقبولة لتعـديل كائنـات النمـاذج الأوليـة الأصـيلة إلا طريقـة واحـدة وهي: ترقيع نقص الدعم.

ترقيع نقص الدعم (أو Polyfilling) هو صناعة بـديل عن تـابِع توضّـحه مواصـفة جافاسـكربت ولكنّـه ليس مدعومًا في محرّك جافاسكربت الهدف.

لهذا السبب نكتب التنفيذ يدويًا ونضعه في كائن النموذج الأولي المضمّن له. مثال:

```
if (!String.prototype.repeat) { // لو لم يكن هناك مثل هذا التابع
  // prototype نُضيفه إلى كائن

  String.prototype.repeat = function(n) {
    // نكرّر السلسلة النصية n مرّة
    // في الواقع فالشيفرة الممتازة هي أكثر تعقيدًا من هذه
    // (تجد خوارزميتها الكاملة في المواصفة)
    // ولكن حتى التعويض الناقص يكون كافيًا أحيانًا كثيرة
    return new Array(n + 1).join(this);
  };
}
alert( "La".repeat(3) ); // LaLaLa
```

8.3.5 الاستعارة من كائنات النماذج الأولية

تحدّثنا في الفصل "المُزخرِفات والتمرير، التابعـان call وapply"، عن اسـتعارة التوابـع، أي حين نأخـذ تابِعًـا من كائن وننسخه إلى كائن غيره.

في أحيان كثيرة نستعير بعض توابِع كائنات النماذج الأولية الأصيلة. مثال على ذلك حين نصنع كائنًـا يشـبه المصفوفات ونريد نسخ بعض توابِع المصفوفات Array إليه، انظر مثلًا الشيفرة التالية:

```
let obj = {
"Hello",
"world!",
  length: 2,
};

obj.join = Array.prototype.join; // هنا

alert( obj.join(',') ); // Hello,world!
```

تعمل الشيفرة أعلاه إذ أنّ الخوارزمية الداخلية لتابِع join المضمّن لا يهمها إلا الفهارس الصحيحة وخاصية الطول length، ولا ترى الكائن أهو حقًّا مصفوفة أم لا. تتصرّف توابِع أخرى مضمّنة مثل تصرّف هذا التابع.

يمكننـا أيضًـا الوراثـة بضـبط obj.__proto__ على Array.prototype فتصـير توابِـع المصـفوفات Array مُتاحة للكائن obj تلقائيًا. ولكن مـا إن يـرث obj من أيّ كـائن آخـر (غـير كـائن Array.prototype) يصير هذا مستحيلًا. لا تنسَ بأنّا لا نستطيع الوراثة إلا من كائن واحد فقط لا غير.

تُعدّ هذه الميزة (ميزة استعارة التوابع) ميزةً مرنة إذ تتيح لنا دمج مختلف مزايا الكائنات إن احتجناها.

8.3.6 الخلاصة

- تتبع كافة الكائنات المضمّنة في اللغة هذا النمط:
  - التوابع مخزّنة داخل كـائن النمـوذج الأولي (مثـل Array.prototype و Object.prototype وDate.prototype وغيرها)
  - لا يخزّن الكائن إلّا بياناته (مثل عناصر المصفوفة وخصائص الكائن والتاريخ)
- تخزّن الأنواع الأولية أيضًا توابِعها في كائنات النماذج الأوليـة لكائنـات تغليـف: Number.prototype و String.prototype و Boolean.prototype. فقط undefined و null ليس لهما كائنـات تغليف.

- يمكنك تعديل كائنات النماذج الأولية المضمّنة في اللغة أو إضافة توابِع جديدة لها، ولكنّ تغييرها ليس أمرًا مستحسنًا. ربما تكون إضافة المعايير الجديدة (والّتي لا يدعمها محرّك جافاسكربت بعد) هي الحالة الوحيدة المسموح بها.

8.3.7 تمارين

ا. إضافة التابع "f.defer(ms)" إلى الدوالّ

الأهمية: ★★★★★

أضِف إلى كائن النموذج الأولي المخصص للدوال التابعَ `defer(ms)`، ووظيفته تشغيل الدالة بعد `ms` مِلّي ثانية. بعدما تنتهي، يفترض أن تعمل هذه الشيفرة:

```
function f() {
  alert("Hello!");
}
f.defer(1000); // تعرض "Hello!" بعد ثانية واحدة
```

الحل:

```
Function.prototype.defer = function(ms) {
  setTimeout(this, ms);
};
function f() {
  alert("Hello!");
}
f.defer(1000); // تعرض "Hello!" بعد ثانية واحدة
```

ب. إضافة المُزخرِف "defer()" إلى الدوال

الأهمية: ★★★★☆

أضِف إلى كائن النموذج الأولي المخصص للدوالّ التابعَ `defer(ms)`، ووظيفته إعادة غلاف يُؤخّر الاستدعاء `ms` مِلّي ثانية.

إليك مثالًا عن طريقة عمله:

```
function f(a, b) {
  alert( a + b );
}
f.defer(1000)(1, 2); // يعرض "3" بعد ثانية واحدة
```

لاحِظ أنّ عليك تمرير الوُسطاء إلى الدالة الأصل.

الحل:

```
Function.prototype.defer = function(ms) {
  let f = this;
  return function(...args) {
    setTimeout(() => f.apply(this, args), ms);
  }
};

// نتأكّد
function f(a, b) {
  alert( a + b );
}
f.defer(1000)(1, 2); // يعرض "3" بعد ثانية واحدة
```

لاحظ بأنّا استعملنا `this` في التابع `f.apply` لجعل المزخرف يعمل لكائن الدوال لذا فإن استدعيَ تابع التغليف كدالّة كائن عندها ستمرر `this` إلى الدالة الأصلية `f`.

```
Function.prototype.defer = function(ms) {
  let f = this;
  return function(...args) {
    setTimeout(() => f.apply(this, args), ms);
  }
};

let user = {
  name: "Ahmad",
  sayHi() {
    alert(this.name);
  }
}
user.sayHi = user.sayHi.defer(1000);
user.sayHi();
```

8.4 توابِع النماذج الأولية والكائنات بلا proto

في أوّل فصل من هذا القسم قُلنا بأنّ هناك طرائق حديثة لكتابة الخاصية [[prototype]].

يُعدّ التابع `__proto__` قديمًا وربّما نقول أيضًا لم يعد مستخدمًا (تحديدًا من جهة المتصفح في معايير جافاسكربت)، النسخ الحديثة هي:

- Object.create(proto[, descriptors]): ينشئ كائنًا فارغًا بضبط `proto` الممرّر ليكون كائن `[[Prototype]]` مع واصِفات الخاصيات الاختيارية لو مُرّرت.
- Object.getPrototypeOf(obj): يُعيد كائن `[[Prototype]]` للكائن `obj`.
- Object.setPrototypeOf(obj, proto): يضبط الخاصية `[[Prototype]]` للكائن `obj` لتكون `proto`.

هذه التعليمات يجب عليك استعمالها بدلًا من `__proto__`. مثال:

```
let animal = {
  eats: true
};

// كائنًا جديدًا يكون الكائن animal نصنع كائنًا له نموذج أولي
let rabbit = Object.create(animal);

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal); // true

Object.setPrototypeOf(rabbit, {}); // نغيّر كائن النموذج الأولي للكائن rabbit إلى {}
```

للتابِع `Object.create` وسيط ثانٍ اختياري وهو: واصِفات الخاصيات، إذ يمكننا تقديم خاصيات إضافية إلى الكائن الجديد مباشرةً هكذا:

```
let animal = {
  eats: true
};
```

```
let rabbit = Object.create(animal, {
  jumps: {
    value: true
  }
});

alert(rabbit.jumps); // true
```

تنسيق الواصِفات هو نفس التنسيق الذي شرحناه في فصل رايات الخاصيات وواصفاتها في جافاسكربت.

يمكننا أيضًا استعمال التابع `Object.create` لنسخ الكائنات بتحكّم أكبر من نسخ الخاصيات في حلقة `for..in`:

```
// إنشاء نسخة مماثلة للكائن obj
let clone = Object.create(Object.getPrototypeOf(obj),
Object.getOwnPropertyDescriptors(obj));
```

يصنع هذا الاستدعاء نسخة حقيقية مطابقة عن `obj` بما فيها الخاصيات: قابلية العدّ و منع قابلية العدّ وخاصيات البيانات و الضوابط/الجوالب - ينسخ كلّ شيء مع خاصية `[[Prototype]]`.

8.4.1 لمحة تاريخية

لو عددنا الآن كلّ الطرائق التي نُدير فيها خاصية `[[Prototype]]` لوجدناها لا تُحصى! الكثير من الطرق لمهمة واحدة. ولكن لماذا؟

طبعًا وكالعادة، لأسباب تاريخية.

- بدأت الخاصية `"prototype"` للبواني (constructor) منذ زمن بعيد جدًا.
- ولاحقًا في 2012، ظهر التابع `Object.create` في معيار اللغة. قدّم بذلك إمكانية صناعة الكائنات لها كائن نموذج أولي (prototype)، ولكن لم تقدّم أيّ طريقة لجلبه أو ضبطه. لهذا صنعت المتصفّحات تابع غير قياسي `__proto__` ليصل إلى كائن النموذج الأولي ويُتيح للمستخدم جلبه وضبطه متى أراد.
- بعدها في 2015 أُضيف التابعين `Object.setPrototypeOf` و `Object.getPrototypeOf` إلى المعيار فيقوموا بنفس مقام التابع `__proto__`. ولكن بحكم الأمر الواقع، فقد كان `__proto__` موجودًا في كلّ مكان، وهكذا صار غير مستحسن استخدامه ونزل في المعيار إلى المُلحق باء (Annex B)، أي صار "اختياريًا للبيئات التي ليست متصفّحات".

والآن صرنا نملك هذه الطرائق الثلاث نتنعّم بها.

ولكن لماذا استبدلوا `__proto__` بالدوال `getPrototypeOf/setPrototypeOf`؟ سؤال مهم وعليه سنعرف ما السوء الكبير للتابِع `__proto__`. واصِل القراءة لمعرفة ذلك.

لا تغيّر قيمة الخاصية [[Prototype]] للكائنات الحالية إن كانت السرعة عنصرًا مهمًا

عادة ما نُسند هذه الخاصية `[[Prototype]]` مرة واحدة فقط وذلك عند إنشاء الكائن ولا نعدلها بعد ذلك. ولكن بالطبع يمكننا استخدام الضابِط/الجالِب للخاصية `[[Prototype]]` في أي وقت نريده. الكائن `rabbit` يرث من الكائن `animal` ولن يتغير هذا الأمر لاحقًا.

محرك لغة جافاسكربت محسّن على النحو الأمثل لهذا الغرض. إن عملية تغيير النموذج الأولي "على عجالة" باستخدام التابع `Object.setPrototypeOf` أو `obj.__proto__=` تعدّ عملية بطيئة جدًا لأنها تكسر التحسينات الداخلية لعمليات الوصول إلى الخاصية لهذا الكائن. لذا فإن كانت السرعة تهمك أو كنت تعرف ما عواقب ما تُقدِم عليه فغيره، وإلا فلا.

8.4.2 الكائنات "البسيطة جدًا"

كما نعلم فيمكننا استعمال الكائنات على أنّها مصفوفات مترابطة لتخزين أزواج المفاتيح/القيم.

ولكن... لو أردنا تخزين المفاتيح التي يُعطينا إيّاها المستخدم (مثل قاموس يكتبه المستخدم) فسنرى مشكلة ظريفة: كلّ المفاتيح تعمل عدا `"__proto__"`. انظر هذا المثال:

```
let obj = {};

let key = prompt("What's the key?", "__proto__"); // ما المفتاح؟
obj[key] = "some value";

alert(obj[key]); // [object Object]، وليس "some value"!
```

لو كتب هنا المستخدم `__proto__`، فستُهمل عملية الإسناد!

لا، هذا ليس مفاجئًا، إذ نعرف بأنّ `__proto__` مميّزة عن غيرها: فإمّا تكون كائنًا أو `null`. ممنوع أن تصير السلسلة النصية (String) كائنَ نموذج أولي.

ولكن لم تكن النية أساسًا لتنفيذ هذا السلوك، صح؟ ما نريده هو تخزين أزواج المفاتيح والقيم، وهناك مفتاح اسمه `"__proto__"` لم يُحفظ كما المفترض، والذّي أنشأ مشكلة!

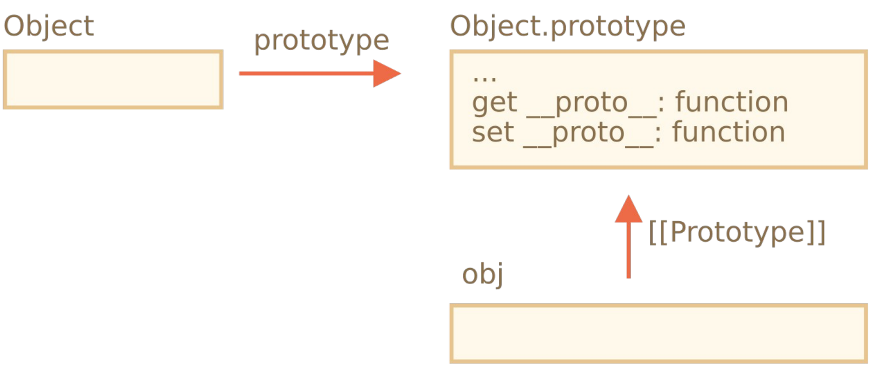
هنا قد لا تكون العواقب وخيمة جدًا، ولكنها في حالات أخرى تكون، مثل لو كنّا نُسند قيم الكائنات، وبعدها تغيرت قيم النماذج الأولية لسبب ما لهذه الكائنات، عندها سيكون ناتج التنفيذ خاطئ وبطريقة غير متوقعة نهائيًا.

الأنكى من هذا هو أنّ المطوّرين -عادةً- لا يفكّرون حتّى بإمكانية حدوث هذا، ما يصنع علل (Bugs) محال ملاحظتها، أو حتّى علل تصير ثغرات أمنية خصوصًا حين نستعمل جافاسكربت من جهة الخوادم.

كما تحدث أيضًا غرائب وعجائب ما إن أسندتَ شيئًا إلى التابع `toString` (وهو دالة بالوضع الافتراضي) وغيره من توابِع أخرى مضمّنة في أصل اللغة. كيف لنا يا تُرى تجنّب هذا؟

أوّلًا، نترك ما نستعمل وننتقل إلى الخارطة `Map`، وهكذا نحلّ كلّ شيء. ولكن الكائنات نفسها `Object` تكفي في هذه الحال إذ أنّ من صنع لغة جافاسكربت فكّر بهذه المشكلة قبل أن نُولد حتّى.

ليست `__proto__` خاصية للكائن، بل خاصية وصول إلى `Object.prototype`:

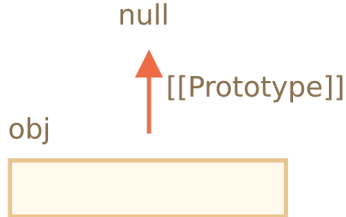


لذا لو قرأنا `obj.__proto__` أو ضبطناها، فسيُستدعى الجالِب (أو الضابِط) من كائن النموذج الأولي prototype لها وتجلب/تضبط كائن `[[Prototype]]`. فكما قلنا في بداية هذا القسم من الكتاب: ليست `__proto__` إلّا طريقة للوصول إلى الخاصية `[[Prototype]]` وليست الكائن نفسه.

الآن بعد هذا كلّه، لو أردنا استعمال الكائن مصفوفةً مترابطة فيمكننا ذلك بخدعة صغيرة:

```
let obj = Object.create(null); // هذه

let key = prompt("What's the key?", "__proto__"); // ما المفتاح؟
obj[key] = "some value";

alert(obj[key]); // "some value"
```

ينشئ التابع `Object.create(null)` كائنًا فارغًا ليس له نموذج أولي prototype (أي أنّ `[[Prototype]]` يساوي `null`):



يعني أنْ ليس هناك أيّ جالِب أو ضابِط موروثان للخاصيـة `__proto__`، فهي الآن خاصيـة بيانـات عاديـة وسيعمل مثالنا أعلاه كما نريد.

نسمّي هذه الكائنات "بالبسيطة جدًا" (very plain) أو "كائنات ليست إلّا قواميس" إذ أنّها أبسط حتّى من الكائن البسيط (العادي) `{...}`. العيب هنا هو أنّ ليس فيها أيّ توابِع كائنات مضمّنة مثل `toString`:

```
let obj = Object.create(null);

alert(obj); // Error (no toString)
```

ولكن ربّما ذلك يكون كافيًا للمصفوفات المترابطة

لاحـظ كيـف أنّ أغلب التوابـع المتعلّقـة بالكائنـات هي بالشـكل `Object.something(...)` (مثـل `Object.keys(obj)`) وليست موجودة في كائن النمـوذج الأولي (prototype)، وسـتعمل كمـا ينبغي لهـذه الكائنات البسيطة:

```
let arabicDictionary = Object.create(null);
chineseDictionary.hello = "hello";
chineseDictionary.bye = "bye";

alert(Object.keys(arabicDictionary)); // hello,bye
```

8.4.3 الخلاصة

التوابع الحديثة لضبط كائنات النماذج الأولية (prototype) والوصول إليها مباشرةً هي:

- Object.create(proto[, descriptors]): ينشـئ كائنًـا فارغًـا بضبـط `proto` الممـرّر ليكـون كـائن `[[Prototype]]` (يمكن أن يحمل القيمة `null`) مع واصِفات الخاصيات الاختيارية لو مُرّرت.
- Object.getPrototypeOf(obj): يُعيد كائن `[[Prototype]]` للكـائن `obj` (مشـابه لعمـل الجـالِب للخاصية `__proto__`).
- Object.setPrototypeOf(obj, proto): يضـبط الخاصـية `[[Prototype]]` للكـائن `obj` لتكـون `proto` (مشابه لعمل الضابِط للخاصية `__proto__`).

ليس من الآمن استعمال الجالِب/الضابِط `__proto__` المضمّن في اللغة إن أردنا وضع مفاتيح صنعها المستخدم في الكائن. ليس ذلك آمنًا إذ يمكن أن يُدخل المستخدم `"__proto__"` مفتاحًا وسنواجه خطأً له عواقب محالة التنبّؤ نأمل أن يُحمد عقباها.

لذا يمكننا إمّا استعمال التابع `Object.create(null)` لإنشاء كائن "بسيط جدًا" بدون استعمال `__proto__` أو يمكننا استغلال كائنات الخرائط `Map` وهو الأفضل. كما يمكننا أيضًا نسخ الكائنات مع جميع واصفاتها من خلال التابع `Object.create`.

```
let clone = Object.create(Object.getPrototypeOf(obj),
Object.getOwnPropertyDescriptors(obj));
```

كما أنّا وضّحنا كيف أنّ `__proto__` ليس إلّا جالِبًا/ضابِطًا للخاصية `[[Prototype]]` وهو موجود تحت ظلّ `Object.prototype` مثل غيره من التوابع.

يمكننا إنشاء كائن ليس له نموذج أولي prototype باستعمال `Object.create(null)`، وهكذا نستعملها على أنّها "خرائط خام (Map)"، والجميل أنّها لا تُمانع قيمة مثل `"__proto__"` لتكون مفتاحًا فيها.

توابِع أخرى:

- Object.keys(obj) / Object.values(obj) / Object.entries(obj): تعيد مصفوفة فيها جميع السلاسل (مفعلٌ بها خاصية قابلية العدّ) على شكل أسماء/قيم/أزواج مفاتيح-قيم.
- Object.getOwnPropertySymbols(obj): تُعيد مصفوفةً فيها جميع الخاصيات الرمزية (symbol properties) الموجودة مباشرةً في الكائن المعطي.
- Object.getOwnPropertyNames(obj): تُعيد مصفوفةً فيها جميع الخاصيات التابعة مباشرةً للكائن المعطي، بما في ذلك الخاصيات غير القابلة للإحصاء (non-enumerable) لكن باستثناء خاصيات الرموز Symbol.
- Reflect.ownKeys(obj): تعيد مصفوفة من جميع المفاتيح الخاصة بها.
- obj.hasOwnProperty(key): تعيد `true` لو كان للكائن `obj` خاصيةً ما مباشرةً (أي أنها لم يرثها).

تُعيد كلّ التوابع التي تُعيد خاصيات الكائن (مثل `Object.keys` وغيرها) خاصياتها "هي". لو أردنا تلك الموروثة فعلينا استعمال حلقة `for..in`.

8.4.4 تمارين

ا. إضافة toString إلى dictionary

الأهمية: ★★★★★

لديك الكائن dictionary حيث أنشأناه باستعمال Object.create(null) ليخزّن أزواج key/value أيًا كانت.

أضِف التابع dictionary.toString() فيه ليُعيد قائمة الخاصيات مفصولة بفواصل. يجب ألا يظهر التابع toString في حلقة for..in عند مرورها على الكائن. إليك طريقة عمل التابع:

```
let dictionary = Object.create(null);

// dictionary.toString اكتب هنا الشيفرة التي تُضيف تابِع

// نُضيف بعض البيانات
dictionary.apple = "Apple";
dictionary.__proto__ = "test"; // ليس __proto__ هنا إلّا خاصية لا أكثر

// في الحلقة apple و __proto__ فقط
for(let key in dictionary) {
  alert(key); // "apple"، ثمّ "__proto__"
}

// الذي كتبته toString حان دور التابع
alert(dictionary); // "apple,__proto__"
```

الحل:

يمكن أن يأخذ التابع كلّ الخاصيات (keys) القابلة للإحصاء باستعمال Object.keys ويطبع قائمة بها.

لنمنع التابع toString من قابلية الإحصاء لنُعرّفه باستعمال واصِف خاصيات. تُتيح لنا صياغة التابع Object.create تقديم الكائن مع واصِفات الخاصيات كوسيط ثاني يمرر للتابِع.

```
// الشيفرة
let dictionary = Object.create(null, {
  toString: { // toString نعرّف الخاصية
    value() { // قيمة الخاصية دالة
```

```
      return Object.keys(this).join();
    }
  }
});

dictionary.apple = "Apple";
dictionary.__proto__ = "test";

// الحلقة في apple و __proto__
for(let key in dictionary) {
  alert(key); // "__proto__" ثمّ ،"apple"
}

// قائمة من الخاصيات طبعها toString مفصولة بفواصل
alert(dictionary); // "apple,__proto__"
```

متى أنشأنا الخاصية باستعمال واصِف فستكون راياتها بقيمة `false` مبدئيًا. إذًا في الشيفرة أعلاه التابع `dictionary.toString` ليس قابلًا للإحصاء.

ألقِ نظرة على فصل "رايات الخاصيات وواصِفاتها" لتُنعش ذاكرتك.

ب. الفرق بين الاستدعاءات

الأهمية: ★★★★★

لنُنشئ ثانيةً كائن `rabbit` جديد:

```
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert(this.name);
};

let rabbit = new Rabbit("Rabbit");
```

هل تؤدّي هذه الاستدعاءات نفس المهمة أم لا؟

```
rabbit.sayHi();
Rabbit.prototype.sayHi();
Object.getPrototypeOf(rabbit).sayHi();
rabbit.__proto__.sayHi();
```

الحل:

في الاستدعاء الأول يكون `rabbit == this`، بينما في البقية يكون `this` مساويًا إلى `Rabbit.prototype` إذ أنّ الكائن الفعلي يكون قبل النقطة.

إذًا فالاستدعاء الأول هو الوحيد الذي يعرض `Rabbit`، بينما البقية تعرض `undefined`:

```
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert( this.name );
}

let rabbit = new Rabbit("Rabbit");

rabbit.sayHi();                        // Rabbit
Rabbit.prototype.sayHi();              // undefined
Object.getPrototypeOf(rabbit).sayHi(); // undefined
rabbit.__proto__.sayHi();              // undefined
```

دورة تطوير تطبيقات الويب باستخدام لغة Ruby

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

9. الأصناف Classes

يتضمن هذا الفصل الأقسام التالية:

1. صياغة الأصناف الأساسية
2. وراثة الأصناف Class inheritance
3. الخاصيات والتوابع الثابتة
4. الخاصيات والتوابع الخاصة والمحمية
5. توسعة الأصناف المضمنة في اللغة
6. فحص الأصناف عبر instanceof
7. المخاليط Mixins

9.1 صياغة الأصناف الأساسية

"في البرمجة كائنية التوجّه، يعدّ الصنف عبارة عن توسعة للشيفرة البرمجية للبرنامج وذلك لإنشاء كائنات كما أنه يهيئ القيم الأولية لمتغيرات الحالة (أو تدعى أحيانًا خاصيات وهي المتغيرات الّتي تكون عضوًا بالصنف) وتطبيق سلوك معين (باستخدام الدوال أو التوابع الأعضاء)" - ويكيبيديا

بعيدًا عن الكلام النظري، ففي الواقع نريد عادةً إنشاء أكثر من كائن تحمل نفس النوع، مثل المستخدمين والبضائع وغيرها.

كما علمنا في الفصل "الباني والعامل new"، يمكن للتعليمة `new function` القيام بهذه المهمة. ولكن، في لغة جافاسكربت الحديثة هناك بوانٍ أكثر تقدّمًا للأصناف إذ تُتيح لنا مزايا جديدة مميّزة تفيدنا حين نُبرمج على طريقة البرمجة كائنية التوجه.

9.1.1 صياغة الأصناف class

إليك الصياغة الأساسية:

```
class MyClass {
  // توابع الصنف
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}
```

بعدها استعمل `new MyClass()` لتُنشِئ كائنًا جديدًا فيه التوابع تلك. يُنادي الباني `constructor()` العبارة `new` تلقائيًا ليهيئ الكائن في المكان المطلوب. إليك مثالًا:

```
class User {

  constructor(name) {
    this.name = name;
  }

  sayHi() {
    alert(this.name);
  }
```

```
}

// الاستعمال:
let user = new User("Ahmad");
user.sayHi();
```

متى استدعينا `new User("Ahmad")`:

1. نكون أنشأنا كائنًا جديدًا.
2. نكون شغّلنا (خلف الكواليس) الباني وممرًا لـه الوسـطاء المناسـبة، وإسـناد هـذه الوسـطاء للمتغـيرات المناسبة. هنا أُسندت الوسيط "Ahmad" إلى المتغير `name` عبر `this.name`.

وبعد ذلك، ننادي توابِع الكائن مثل `user.sayHi()`.

لا يوجد فاصلة بين التوابع

من الشائع بين المطورين المبتدئين وضع فاصلة بين توابع الصنف والّذي سيرمي خطأ في الصياغة، وهذه نقطة مهمة لعدم الخلط بينها والكائنات المجردة. إذًا داخل الصنف الفاصلة غير مطلوبة.

9.1.2 ما الصنف أصلًا؟

هذا جميل، ولكن ما هو الصنف `class` أساسًا؟ لو ظننته كيانًا جديدًا كليًا في اللغة، فأنت مخطئ. هيا معًـا نكشف الأسرار ونعرف ماهيّة الصنف حقًا، بذلك نفهم أمور كثيرة معقّدة، بسهولة.

الصنف -في جافاسكربت- مشابه للدالة نوعًا ما. انظر بنفسك:

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// الإثبات: الصنف User هو دالّة
alert(typeof User); // function
```

إليك ما يفعله الباني `class User {...}` حقًا:

1. يُنشِـئ دالّـة باسـم `User` تصـير هي نـاتج التصـريح عن الصـنف، وتُؤخـذ شـيفرة الدالّـة من البـاني `constructor` (يُعامل الباني الصنف على أنّه فارغ إن لم تكن كتبت دالّة تغيّر ذلك).
2. يُخزّن توابِع الصنف (مثل `sayHi`) في النموذج الأولي للكائن `User.prototype`.

متى ما أُنشأ كائن جديد (`new User`)، واستدعينا تابعًا منه يُؤخذ التابع من كائن النموذج الأولي prototype كما وضّحنا في الفصل "الوراثة النموذجية - 2 -". هكذا يكون للكائن تصريح الوصول إلى توابع الصنف. يمكن أن نوضّح ناتج التصريح `class User` في هذه الصورة:

User
```
constructor(name) {
  this.name = name;
}
```
prototype →

User.prototype
sayHi: function
constructor: User

إليك الشيفرة:

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// الصنف ما هو إلّا دالةً
alert(typeof User); // function

// أو للدّقة هو تابع الباني...
alert(User === User.prototype.constructor); // true

// التوابع موجودة في كائن النموذج الأولي User.prototype، مثال:
alert(User.prototype.sayHi); // alert(this.name);

// في كائن النموذج الأولي prototype تابِعين بالضبط
alert(Object.getOwnPropertyNames(User.prototype)); // constructor,
sayHi
```

9.1.3 ليست مجرد تجميل لغوي

يقول البعض بأنّ الأصناف `class` هي مجرد "تجميل لغوي" (Syntactic sugar)، أي أنّها صياغة أساس تصميمها هو تسهيل القراءة دون تقديم ما هو جديد، إذ يمكننا ببساطة التصريح عن ذات الشيء بدون تلك الكلمة المفتاحية `class`:

```
// نُعيد كتابة صنف المستخدم User باستعمال الدوال فقط

// 1. نُنشِئ دالّة الباني
```

```
function User(name) {
  this.name = name;
}
//عادةً إن أي نموذج أولي لتابِع معين لديه باني
// لذلك لسنا بحاجة لإنشائه

// 2. نضيف التابع إلى النموذج الأولي
User.prototype.sayHi = function() {
  alert(this.name);
};

// طريقة الاستخدام:
let user = new User("Ahmad");
user.sayHi();
```

ناتج هذا التصريح أعلاه يشبه كثيرًا... لا بل يتطابق مع تصريح الصنف. لهذا ففكرة أنّ الأصناف هي حقًا تجميل لغوي لتعريف البواني مع توابِع كائن النموذج الأولي prototype لها - هي فكرة منطقية حقًا. مع ذلك، فهناك فوارق.

أولًا، تُوضع على الدوال الّتي تُنشؤها عبارة `class` كخاصيّة داخلية فريدة لهذا الصنف هكذا `[[FunctionKind]]:"classConstructor"`. لذا فالطريقتين (هذه وإنشائها يدويًا) ليستا تمامًا الشيء نفسه. وعلى عكس الدوال العادية، يلزم عليك استدعاء باني الصنف باستعمال `new`:

```
class User {
  constructor() {}
}

alert(typeof User); // function
User(); // Error: Class constructor User cannot be invoked without 'new'
```

كما وأنّ تمثيل أغلب محرّكات جافاسكربت النصّيّ لباني الصنف يبدأ بالعبارة "...class"

```
class User {
  constructor() {}
}
alert(User); // class User { ... }
```

ثانيًا، توابِع الأصناف غير قابلة للإحصاء/، فيضبط التصريح عن الصنف راية `enumerable` على القيمة `false` لكلّ التوابع في كائن النموذج الأولي للصنف `"prototype"`. هذا جميل إذ لا نريد ظهور توابِع الصنف حين نستعرضهن باستعمال حلقة `for..in`.

ثالثًا، تستعمل الأصناف الوضع الصارم دومًا، فكلّ الشيفرة البرمجية في الباني تكون بالوضع الصارم (سبق وأن تحدثنا في فصل سابق عن الوضع الصارم في لغة جافاسكربت).

كما أنّ صياغة الأصناف تُفيدنا بكثير من المزايا نشرحها لاحقًا.

9.1.4 تعابير الأصناف

كما الدوال فيمكن التعريف عن الأصناف داخل التعابير الأخرى وتمريرها وإعادتها وإسنادها وغيره.

إليك مثال عن تعبير صنف:

```
let User = class {
  sayHi() {
    alert("Hello");
  }
};
```

وكما تعابير الدوال المسمّاة NFE وهي اختصارًا للعبارة Named Function Expressions، يمكن أن نضع اسمًا لتعابير الأصناف. ولو كان لتعبير الصنف اسمًا فسيكون ظاهرًا داخل الصنف فقط لا غير:

```
// "(NCE) "تعبير أصناف مسمَّى
// (ليس في المواصفات القياسية للغة هذا الاسم، لكنّها شبيهة بتعابير الدوال المسمّاة(NFE) )
let User = class MyClass {
  sayHi() {
    alert(MyClass); // لا يظهر اسم الصنف MyClass إلّا داخل الصنف
  }
};

new User().sayHi(); // يعمل ويعرض تعريف MyClass

alert(MyClass); // خطأ اسم تعبير الصنف MyClass غير مرئي خارج الصنف
```

يمكننا حتى جعل الأصناف جاهزة عند الطلب، هكذا:

```
function makeClass(phrase) {
  // declare a class and return it
  return class {
    sayHi() {
      alert(phrase);
    };
  };
}

// إنشاء صنف جديد
let User = makeClass("Hello");

new User().sayHi(); // Hello
```

9.1.5 الجوالِب والضوابِط والاختصارات الأخرى

كما الكائنات المجردة فيمكن أن يكون في الأصناف ضوابِط وجوالِب Gettes/Setters وأسماءً محسوبةً للخاصيات (computed properties name) وغيرها.

إليك مثالًا عن خاصية `user.name` كتبنا تنفيذها عبر ضابِط وجالِب:

```
class User {

  constructor(name) {
    // يشغّل الضابط
    this.name = name;
  }

  // هنا
  get name() {
    return this._name;
  }

  // وهنا
  set name(value) {
    if (value.length < 4) {
      alert("Name is too short."); // الاسم قصير جدًا
```

```
      return;
    }
    this._name = value;
  }

}

let user = new User("Ahmad");
alert(user.name); // Ahmad

user = new User(""); // الاسم قصير جدًا
```

ينشئ التعريف عن الصنف جوالِب وضوابِط في كائن `User.prototype` هكذا:

```
Object.defineProperties(User.prototype, {
  name: {
    get() {
      return this._name
    },
    set(name) {
      // ...
    }
  }
});
```

وهذا مثال عن استخدام أسماءً محسوبةً للخاصيّة computed property name داخل أقواس `[...]`:

```
class User {

  // هنا
  ['say' + 'Hi']() {
    alert("Hello");
  }

}

new User().sayHi();
```

9.1.6 خاصيات الأصناف

> الخاصيات على مستوى الأصناف هي إضافة حديثة على اللغة، لذا قد يكون عليك تعويض النقص في المتصفّحات القديمة.

نرى في المثال أعلاه بأنّ للصنف `User` توابِع فقط. هيا نُضف الخاصية `name` للصنف `class User`:

```
class User {
  name = "Anonymous"; // هكذا

  sayHi() {
    alert(`Hello, ${this.name}!`);
  }
}

new User().sayHi();

alert(User.prototype.sayHi); // User.prototype موجود في كائن
alert(User.prototype.name); // undefined، غير موجودة في كائن User.prototype
```

الأمر الجدير بالذكر أن خاصيات الصنف تعيّن على أنها كائنات فردية وليست ضمن النموذج الأولي للصنف `User.prototype`. من الناحية العملية تُعالجُ هذه الخاصيات بعد أن يُنهي الباني عمله.

ا. إنشاء توابع مرتبطة بخاصيات الصنف

كما تحدثنا في فصلٍ سابق من هذا الكتاب ربط الدوالّ، إن دلالات الكلمة المفتاحية `this` في الدوالّ ديناميكية إذ تعتمد إعتمادً أساسيًا على سياق الاستدعاء.

لذا فإن مررها تابِعٌ من كائنٍ معين للكلمة المفتاحية `this`، واستدعيت مرة أخرى في مكان آخر بغير السياق السابق، فلن تشير إلى نفس الكائن السابق بسبب تغير سياق الاستدعاء.

على سبيل المثال الشيفرة البرمجية التالية ستظهر `undefined`.

```
class Button {
  constructor(value) {
    this.value = value;
  }

  click() {
```

```
    alert(this.value);
  }
}

let button = new Button("hello");

setTimeout(button.click, 1000); // undefined
```

تدعى هذه المشكلة "ضياع قيمة this". وهنالك طريقتين لإصلاح هذه المشكلة كما ذكرنا في فصل "ربط الدوال" هما:

1. تمرير دالّة مغلّفة (مثل: `setTimeout(() => button.click(), 1000)`).
2. ربط الدالّة بصنف كما في الباني التالي:

```
class Button {
  constructor(value) {
    this.value = value;
    this.click = this.click.bind(this);
  }

  click() {
    alert(this.value);
  }
}

let button = new Button("hello");

setTimeout(button.click, 1000); // hello
```

تزودنا خاصيات الصنف بصياغة مناسبة للحل الّذي سنستخدمه:

```
class Button {
  constructor(value) {
    this.value = value;
  }
  click = () => {
    alert(this.value);
```

```
  }
}

let button = new Button("hello");

setTimeout(button.click, 1000); // hello
```

تنشئ الخاصية `click= () => {...}` دالّة مستقلة لكل كائن من `Button`، والكلمة المفتاحية `this` تشير إلى الكائن نفسه. وبإمكاننا بعدها تمرير `button.click` في أي مكان ومع احتفاظها بالقيمة الصحيحة للكلمة المفتاحية `this`.

إن استخدام هذه الطريقة مفيد جدًا في بيئة المتصفحات وخصيصًا عند احتياجنا لإعداد دالّة مستمع الحدث (event listener).

9.1.7 الخلاصة

الصياغة الأساسية للأصناف هي كالآتي:

```
class MyClass {
  prop = value; // خاصية

  constructor(...) { // الباني
    // ...
  }

  method(...) {} // تابع

  get something(...) {} // تابع جلب
  set something(...) {} // تابع ضبط

  [Symbol.iterator]() {} // تابع اسمه محسوب (نضع رمزًا هنا)
  // ...
}
```

تقنيًا، فالصنف `MyClass` ما هو إلّا دالّة (تلك الّتي نكتبها لتكون الباني `constructor`)، بينما التوابع والضوابِط والجوالِب تُكتب في كائن `MyClass.prototype`.

ستتعرّف أكثر في الفصول اللاحقة عن الأصناف والوراثة والميزات الأخرى.

9.1.8 تمارين

1. أعِد كتابتها لتكون صنفًا

الأهمية: ★★★★★

كُتب صنف الساعة `Clock` وكأنّه دالّة. أعِد كتابته ليكون بصياغة الأصناف.

ملاحظة: تدقّ عقارب الساعة في الطرفية، افتحها واحترس من العقارب اللادغة.

اطلع على تجربة حية للتمرين.

الحل:

```
class Clock {
  constructor({ template }) {
    this.template = template;
  }

  render() {
    let date = new Date();

    let hours = date.getHours();
    if (hours < 10) hours = '0' + hours;

    let mins = date.getMinutes();
    if (mins < 10) mins = '0' + mins;

    let secs = date.getSeconds();
    if (secs < 10) secs = '0' + secs;

    let output = this.template
      .replace('h', hours)
      .replace('m', mins)
      .replace('s', secs);

    console.log(output);
  }

  stop() {
```

```
    clearInterval(this.timer);
  }

  start() {
    this.render();
    this.timer = setInterval(() => this.render(), 1000);
  }
}


let clock = new Clock({template: 'h:m:s'});
clock.start();
```

ويمكن الاطلاع على تجربة حية للحل.

9.2 وراثة الأصناف Class inheritance

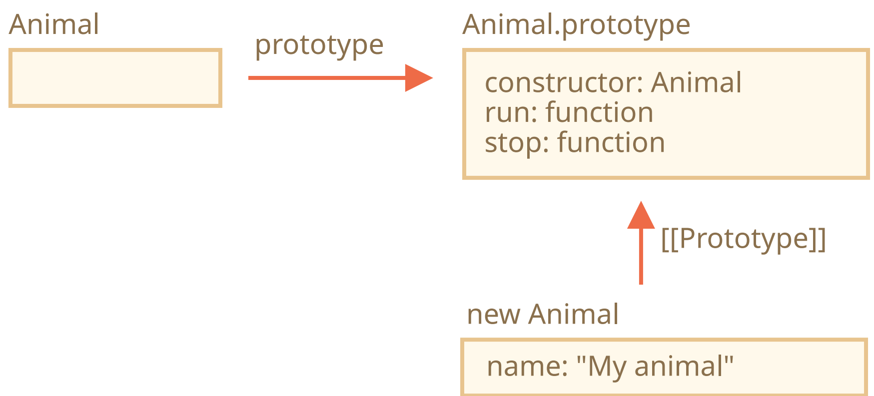
تُعدّ وراثة الأصناف واحدةً من الطرائق لتوسعة أحد الأصناف لديك، أي أن نقدّم وظائف جديدة لأحد الأصناف علاوةً على ما لديه.

9.2.1 عبارة التوسعة extends

لنقل بأنّ لدينا صنف الحيوان `Animal`:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`); // يركض حيوان كذا بالسرعة كذا
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} stands still.`); // يقف حيوان كذا في مكانه
  }
}

let animal = new Animal("My animal");
```

هكذا نصف كائن الحيوان `animal` وصنف الحيوان `Animal` في صورة:



ولنقل بأنّنا نريد إضافة صنف آخر… ليكن أرنبًا `class Rabbit`.

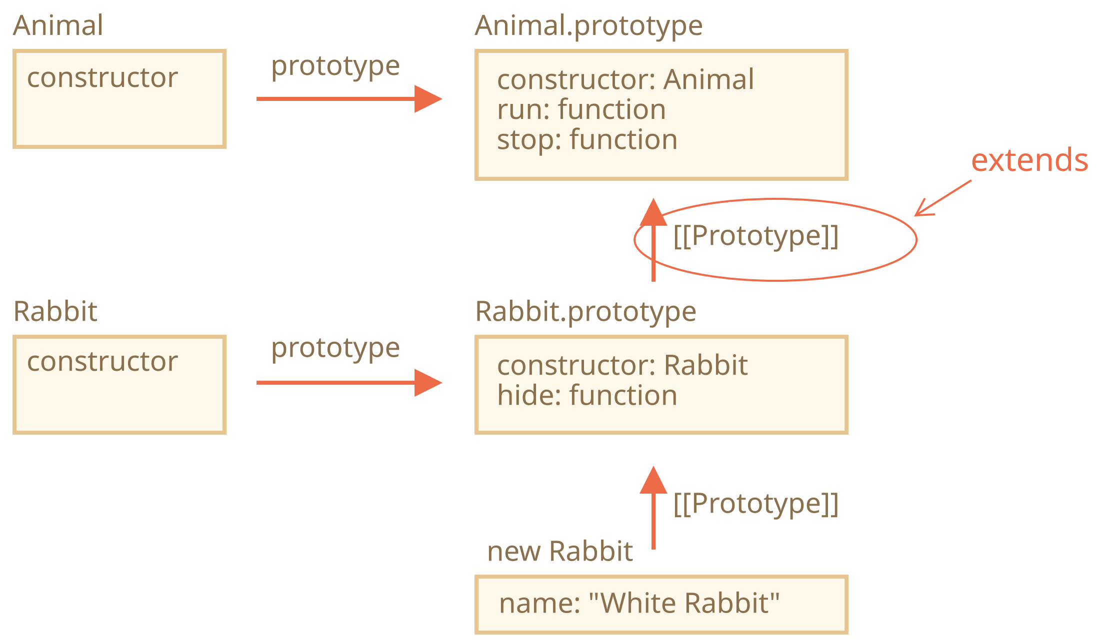
وطبعًا، فالأرانب حيوانات أيضًا، وعلى صنف الأرانب `Rabbit` أن يكون أساسه الحيوان `Animal` ليصل إلى توابِع الحيوانات فتقوم الأرانب بما تقوم به الحيوانات "العادية" (generic).

صياغة توسعة الصنف إلى صنف آخر هي: `class Child extends Parent` (صنف الابن توسعة من صنف الأبّ). لنصنع صنف الأرنب `class Rabbit` ليرث الحيوان `Animal`:

```
class Rabbit extends Animal { // لاحظ
  hide() {
    alert(`${this.name} hides!`); // اختفى هكذا!
  }
}

let rabbit = new Rabbit("White Rabbit"); // الأرنب الأبيض
rabbit.run(5); // يركض حيوان الأرنب بسرعة 5
rabbit.hide(); // اختفى الأرنب الأبيض!
```

يمكن لكائن `Rabbit` الوصول إلى توابِع `Rabbit` (مثل `rabbit.hide()`) كما توابِع `Animal` (مثل `rabbit.run()`).

داخليًا فعبارة `extends` تعمل كما تعمل ميكانيكية كائنات prototype المعهودة، فتضبط `Rabbit.prototype.[[Prototype]]` ليكون `Animal.prototype`. بهذا لو لم يوجد التابع في كائن `Rabbit.prototype`، يأخذه المحرّك من `Animal.prototype`.



فمثلًا لنجد التابع `rabbit.run` يبحث المحرّك (من أسفل إلى أعلى، كما الصورة):

1. كائن الأرنب `rabbit` (ليس فيه `run`).
2. كائن prototype له، أي `Rabbit.prototype` (فيه `hide` وليس فيه `run`).
3. كائن prototype له، أي (بسبب `extends`) `Animal.prototype`، بهذا يجد تابِع `run` أخيرًا.

كما نذكر من فصل "الوراثة النموجية"، فمحرّك جافاسكربت نفسه يستعمل التوارث عبر `prototype` لكائناته المضمّنة في اللغة. فمثلًا كائن `Date.prototype.[[Prototype]]` هو الكائن `Object.prototype`. لهذا يمكن للتواريخ الوصول إلى توابع الكائنات العادية.

لاحظ، تسمح صياغة الصنف ليس بتحديد الصنف فقط وإنما إضافة أي تعبير بعد عبارة `extends`. فمثلًا استدعاء التابع سيبني صنف الأب.

```
function f(phrase) {
  return class {
    sayHi() { alert(phrase) }
  }
}

class User extends f("Hello") {}
new User().sayHi(); // Hello
```

سيرثُ الصنف `class User` من النتيجة للصنف `f("Hello")`.

وهذه الميزة مفيدة لكتابة الأنماط البرمجية المتقدمة عندما نستخدم تابع لينشئ صنف صنف اعتمادًا على عدة شروط والتي يمكن أن ترثها.

9.2.2 إعادة تعريف دالة

الآن نتحرّك ونعيد تعريف أحد التوابع. مبدئيًا فكلّ التوابع غير المحدّدة في صنف الأرانب تُؤخذ مباشرةً "كما هي" من صنف الحيوانات. ولكن لو حدّدنا تابِعًا معيّنًا في في `Rabbit` (وليكن `stop()`) فسيُستعمل بدله:

```
class Rabbit extends Animal {
  stop() {
    // rabbit.stop() الآن سنستخدمها من أجل
    // بدلًا من استخدام stop() من الصنف مباشرة
  }
}
```

عادةً لا نرغب باستبدال كامل ما في التابع الأب، بل البناء فوقه أو تعديله أو توسعة وظائفه، أي ننفّذ شيئًا في التابع ثمّ نستدعي التابع الأبّ قبل أو بعد ذلك.

ولحسن الحظ فالأصناف تقدّم لنا عبارة `"super"`.

- نستعمل `super.method(...)` لنستدعي تابِعًا أبًا.
- ونستدعي `super(...)` لنستدعي الباني الأب (هذا فقط لو كنّا في باني هذه الدالة).

فمثلًا، ليختبئ هذا الأرنب النبه ما إن يتوقّف:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} stands still.`);
  }
}
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
  // هنا
  stop() {
    super.stop(); // نستدعي stop في الأب
    this.hide(); // ثمّ نستدعي hide
  }
}
let rabbit = new Rabbit("White Rabbit");
rabbit.run(5); // سيركض "White Rabbit" بسرعة 5
rabbit.stop(); // توقف الأرنب. وهو الآن مختبئ
```

الآن صار داخل الأرنب التابع Rabbit الذي يستدعي التابع stop من أباه، كما شرحنا في فصل "الدوال السهمية" لا يمكننا استخدام الكلمة المفتاحية super على الدوالّ السهمية .

ولو استطعنا الوصول إليها من خلال الكلمة المفتاحية super فستكون مأخوذة من الدالّة الخارجية. هكذا:

```
class Rabbit extends Animal {
  stop() {
    setTimeout(() => super.stop(), 1000); // استدعاء الأب سيتوقف بعد ثانية واحدة
  }
}
```

إن عمل الدالّة stop() مشابه تمامًا لعمل الكلمة المفتاحية super مع الدوال السهمية. لذا فإنها تعمل مثلما نريد. ولو حُددت كدالّة عادية فسيظهر لدينا خطأ:

```
// Unexpected super
setTimeout(function() { super.stop() }, 1000);
```

9.2.3 إعادة تعريف الباني

لم يكن لصنف الأرنب Rabbit (حتّى اللحظة) أيّ بانٍ له. حسبما تقول المواصفة فلو وسّع أحد الأصناف صنفًا آخر ليس له بانيًا، فسيُولّد المحرّك هذا الباني "الفارغ":

```
class Rabbit extends Animal {
  // يُولّد للأصناف التي تُوسّع أخرى وليس فيها بانيات
  constructor(...args) {
    super(...args);
  }
}
```

كما نرى فهي تستدعي الباني constructor الأبّ بتمرير كلّ الوُسطاء إليه، فقط. لا يحدث هذا إلّا لو لم نكتب بانيًا في الصنف الابن.

لنُضف الآن بانيًا من عندنا إلى الأرنب Rabbit. سيضبط هذا الباني الخاصية earLength علاوةً على name:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
```

```
    }
    // ...
  }
  class Rabbit extends Animal {
    // هنا  constructor(name, earLength) {
      this.speed = 0;
      this.name = name;
      this.earLength = earLength;
    }
    // ...
  }

  // لا تعمل!
  let rabbit = new Rabbit("White Rabbit", 10); // Error: this is not defined.
```

ماذا؟! هناك خطأ! انتهى الأمر، "صناعة الأرانب" لم تعد ممكنة بعد الآن. تراه ما المشكلة؟

باختصار: على البانيات في الأصناف الموروثة استدعاء `super(...)` هذا أولًا، وثانيًا استدعاءه قبل استعمال `this`. ولكن لحظة... لماذا؟ ما الذي يجري؟ هذا المطلب غريب حقًا.

بالطبع لا شيء بدون توضيح وشرح، لذا فلنتعمّق داخل التفاصيل ونفهم ما يجري "حبّة حبّة". تضع لغة جافاسكربت خطًا فاصلًا بين الدالة البانية للصنف الموروث (أي "الباني المشتقّ") وغيرها من دوال. لهذا الباني خاصية داخلية فريدة اسمها `[[ConstructorKind]]:"derived"`، وهي علامة يضعها المحرّك عليه داخليًا خلف الكواليس. تؤثّر هذه "العلامة" على سلوك الباني حين نستعمله مع `new`.

- حين نُنفّذ الدوال العادية باستعمال `new`، تُنشِئ لنا كائنًا فارغًا وتضبطه ليكون `this`.
- ولكن حين يعمل الباني المشتق، فلا يفعل ذلك، بل يتوقّع من الباني الأبّ القيام بهذه المهمة الصعبة.

لذا على الباني المشتق استدعاء `super` ليُنفّذ باني أباه (غير المشتق) وإلّا فلن يُنشأ أيّ كائن يكون `this`، بهذا تكون الشيفرة خطأ.

على باني الصنف `Rabbit` استدعاء `super()` قبل `this` ليعمل، هكذا تمامًا:

```
  class Animal {

    constructor(name) {
      this.speed = 0;
```

```
    this.name = name;
  }

  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
    super(name); // هنا
    this.earLength = earLength;
  }

  // ...
}

// الآن كلّ شيء كما يجب أن يكون
let rabbit = new Rabbit("White Rabbit", 10);
alert(rabbit.name); // White Rabbit
alert(rabbit.earLength); // 10
```

9.2.4 عبارة super: أمور داخلية و [[HomeObject]]

> سترى معلومات متقدّمة، فلو كنت تقرأ هذا الفصل أوّل مرّة فيمكنك تخطّي هذا الجزء، إذ يتكلّم عن الميكانيكا الداخلية التي يستعملاها التوارث وعبارة super.

هيّا ننزل إلى الأعماق ونرى ما خلف كواليس super. في هذه الرحلة سنرى أمور جميلة أيضًا "إكسترا سوبر". لنوضّحها من البداية: لو أخذت كلّ ما تعلّمناه حتّى الحظة، فما من طريقة لتعمل فيها عبارة super في أيّ حال من الأحوال!

تمامًا، كما فكّرت الآن، لنطرح السؤال: كيف تعمل هذه العبارة تقنيًا أساسًا؟ متى ما عمل أحد توابِع الكائنات، جلب الكائن الحالي على أنّه this. فلو استدعينا super.method()‎ فكلّ ما على المحرّك فعله هو جلب التابع method من كائن prototype للكائن الحالي، صحيح؟ أجل ولكن كيف ذلك؟

ربّما ترى المهمة سهلة ولكنّها ليست كذلك البتة. يمكن القول أنّ المحرّك يعلم بالكائن الحالي this، فيمكنه أن يأخذ تابِع method في الأب باستعمال this.__proto__.method. ولكن للأسف فهذا الحلّ "البسيط" لن يعمل أبدًا.

لنوضّح المشكلة أولًا، باستعاضة الأصناف لتكون كائنات عادية لتسهيل الفهم.

> لو لم تريد معرفة التفاصيل فتخطّى هذا القسم وانتقل إلى الجزء `[[HomeObject]]`، لا مشكلة. أو واصِل معنا في هذه الرحلة الموحشة في أعماق غابة لغة جافاسكربت.

في المثـال أسـفله، نـرى `rabbit.__proto__ = animal`. لنجـرّب الآن هـذا الأمـر: داخـل `rabbit.eat()` نستدعي `animal.eat()` باستعمال `this.__proto__`:

```
let animal = {
  name: "Animal",
  eat() {
    alert(`${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Rabbit",
  eat() {
    // هذه إحدى طرق عمل super.eat()
    this.__proto__.eat.call(this); // (*)
  }
};

rabbit.eat(); // Rabbit eats.
```

أخذنا في السطر `(*)` التابعَ `eat` من كائن prototype ‏(`animal`) واستدعيناه على أنّ السياق هـو الكـائن الحالي. لاحظ أهمية `.call(this)` إذ لو كتبنا `this.__proto__.eat()` فقـط فسـيُنفّذ التـابع `eat` الأب داخل سياق كائنَ prototype، وليس في الكائن الحالي.

وفي الجزء الأول من الشيفرة نرى كلّ شيء يعمل: عمل التابع `alert` كمـا ينبغي عليـه. حـان وقت إضـافة كائن آخر إلى السلسلة، وكسر هذه السلسلة إربًا:

```
let animal = {
  name: "Animal",
  eat() {
    alert(`${this.name} eats.`);
  }
```

```
};

let rabbit = {
  __proto__: animal,
  eat() {
    // ...هنا يأكل الأرنب كما تأكل الأرانب، بعدها نستدعي التابع الأبّ (animal)
    this.__proto__.eat.call(this); // (*)
  }
};
let longEar = {
  __proto__: rabbit,
  eat() {
    // ...الأرنب ذو الأذن الطويلة يلهو ويلعب، ثمّ نستدعي التابع الأبّ (rabbit)
    this.__proto__.eat.call(this); // (**)
  }
};

// هنا!longEar.eat(); // Error: Maximum call stack size exceeded
```

لم تعد الشيفرة تعمل الآن! نرى خطأً عند استدعاء `longEar.eat()`.

قد لا يبدو الأمر جليًا من أوّل نظرة، ولكن لو تعقّبنا استدعاء `longEar.eat()` فسنرى الأمر بوضوح، ففي السطرين (*) و (**) تكون قيمة `this` هي الكائن الحالي (`longEar`). هذا ضمن الأساسيات، فعلى توابِع الكائنات جلب الكائن الحالي فهو `this`، وليس كائنَ prototype أو ما شابهه.

بذلك في السطرين معًا (*) و (**) تكون قيمة `this.__proto__` واحدة: الكائن `rabbit`، وكلاهما يستدعيان التابع `rabbit.eat` دون أن يرتقيا بالسلسلة، فيدوران في حلقة لا نهاية لها.

إليك عمّا يحدث في صورة:

```
let rabbit = {
  __proto__: animal,
  eat() {
    this.__proto__.eat.call(this); (*)
  }  rabbit                 longEar
};

let longEar = {
  __proto__: rabbit,
  eat() {
    this.__proto__.eat.call(this); (**)
  }  rabbit                 longEar
};
```

نرى في التابع `longEar.eat()` عند السطر `(**)` استدعاء `rabbit.eat` بتمرير `this=longEar`.

```
// this = longEar قيمة longEar.eat() نرى داخل
this.__proto__.eat.call(this) // (**)
// يصير
longEar.__proto__.eat.call(this)
// وهو فعليًا
rabbit.eat.call(this);
```

وبعدها في السطر `(*)` داخل `rabbit.eat`، نحاول تمرير الاستدعاء إلى مستوًى أعلى داخل السلسلة، ولكن قيمة `this=longEar`، بهذا تصير قيمة `this.__proto__.eat` هي `rabbit.eat` ثانيةً.

```
// this = longEar قيمة rabbit.eat() نرى داخل
this.__proto__.eat.call(this) // (*)
// becomes
longEar.__proto__.eat.call(this)
// or (again)
rabbit.eat.call(this);
```

بهذا... يستدعي `rabbit.eat` نفسه في حلقة لانهاية لها لأنّها يعجز عن الارتقاء في السلسلة. ما من طريقة لحلّ هذه المشكلة باستعمال `this` فقط.

أ. [[HomeObject]]

أضافت لغة جافاسكربت -لحلّ هذه المعضلة- خاصية داخلية (أخرى) للدوال، وهي "الكائن المنزل" `[[HomeObject]]`. متى ما حُدّدت الدالة لتكون صنفًا أو تابعًا لكائن، أصبحت خاصية `[[HomeObject]]` للدالة ذلك الصنف أو الكائن.

تستعمل super هذا الكائن لحل كائن prototype الأبّ هو وتوابِعه. لنرى كيف يعمل هذا الشيء، بالكائنات العادية أولًا:

```
let animal = {
  name: "Animal",
  eat() {         // animal.eat.[[HomeObject]] == animal
    alert(`${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Rabbit",
  eat() {         // rabbit.eat.[[HomeObject]] == rabbit
    super.eat();
  }
};

let longEar = {
  __proto__: rabbit,
  name: "Long Ear",
  eat() {         // longEar.eat.[[HomeObject]] == longEar
    super.eat();
  }
};

// يعمل التابع كما نريد
longEar.eat();  // Long Ear eats.
```

عملت الشيفرة كما المفترض ذلك بسبب آلية عمل `[[HomeObject]]`. تعرف التوابع (مثل `longEar.eat`) خاصيةَ `[[HomeObject]]` لها وتأخذ التابع الأبّ من كائن prototype لذاك الكائن، ودون استعمال `this` أبدًا.

ب. التوابع ليست "حرّة"

كما نعلم فالتوابع -بنحوٍ عام- "حرّة" وليست مربوطة بأيّ كـائن. فيمكننـا نسـخها بين الكائنـات واسـتعمالها بتمرير قيمة `this` أخرى. ولكن وجود `[[HomeObject]]` يخلّ بهـذا المبـدأ إذ تتـذكّر التوابـع كائناتهـا الأصـلية هكذا. يبقى هذا الارتباط وثيقًا للأبد إذ لا يمكننا تغيير `[[HomeObject]]`.

ولكن المكان الوحيد الذي نستعمل فيه خاصـية `[[HomeObject]]` (في لغـة جافاسـكربت) هـو `super`. يعني أنّه لو لم يستعمل التـابع `super` فيمكننـا عـدّه حـرًا ونمضـي بنسـخه وتوزيعـه على الكائنـات. ولكن مـتى استعملت `super`، ساءت الأمور.

إليك مثالًا كاملًا عن نتيجة خطأ بعد النسخ بسبب `super`:

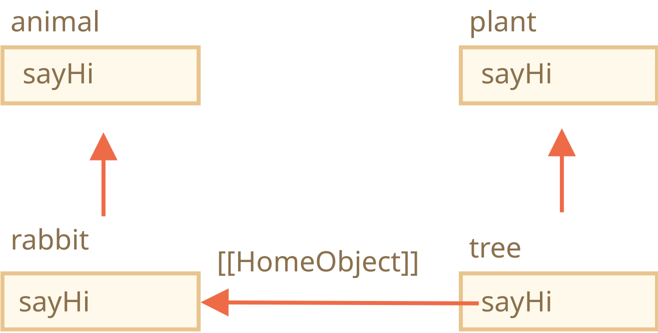
```
let animal = {
  sayHi() {
    console.log(`I'm an animal`); // أنا حيوان
  }
};

// يرث صنف الأرنب صنفَ الحيوان
let rabbit = {
  __proto__: animal,
  sayHi() {
    super.sayHi();
  }
};

let plant = {
  sayHi() {
    console.log("I'm a plant"); // أنا نبات
  }
};

// يرث صنف الشجرة صنفَ النبات
let tree = {
  __proto__: plant,
  sayHi: rabbit.sayHi // (*)
};
```

```
tree.sayHi();  // أنا حيوان (؟!؟)
```

باستدعاء `tree.sayHi()` نرى الشجرة تقول "أنا حيوان". لا، لا! سبب ذلك بسيط جدًا:

- في السطر `(*)` نسخنا التابع `tree.sayHi` من `rabbit`. من يدري، ربّما لنقلّل من تكرار الشيفرات؟
- وخاصية `[[HomeObject]]` لها هي الصنف `rabbit`، إذ صنعنا التابع داخل `rabbit`، وما من طريقة لتغيير `[[HomeObject]]`.
- نرى في شيفرة التابع `tree.sayHi()` الاستدعاءَ `super.sayHi()`، وهو ينتقل إلى أعلى عند `rabbit` ويأخذ التابع من `animal`.

إليك صورة توضّح ما يحدث:



ج. توابِع لا صفات دالية

تُعرّف اللغة عن خاصيات `[[HomeObject]]` للتوابِع في الأصناف وفي الكائنات العادية. ولكن في حالة الكائنات فيجب تعريف التوابع هكذا تمامًا `method()` وليس هكذا `"method: function()"`.

قد لا نرى فرقًا جوهريًا في الطريقتين، لكنّ محرّكات جافاسكربت تراه كذلك. استعملنا في المثال أسفله صياغة ليست بتابِع للموازنة. بهذا لم تُضبط خاصية `[[HomeObject]]` ولن تعمل الوراثة:

```
let animal = {
  eat: function() { // نكتبها هكذا بدل eat() {...} عمدًا
    // ...
  }
};
```

```
let rabbit = {
  __proto__: animal,
  eat: function() {
    super.eat();
  }
};

rabbit.eat();  // Error calling super (إذ [[HomeObject]] غير موجودة)
```

9.2.5 الخلاصة

- نستعمل `class Child extends Parent` لتوسعة الأصناف: أي أنّ خاصية `Child.prototype.__proto__` ستكون `Parent.prototype`، فتصير التوابع موروثة.
- عندما نعيد تعريف الباني: علينا استدعاء الباني الأبّ باستعمال `super()` في الباني "الابن" ذلك قبل استعمال `this`.
- عندما نعيد تعريف أي تابع آخر: يمكننا استعمال `super.method()` في التابع "الابن" لاستدعاء التابع "الأبّ".
- أمور داخلية: تتذكّر التوابع صنفها/كائنها وتحفظه في خاصية `[[HomeObject]]` الداخلية، هكذا يحلّ `super` التوابع الأب، بذلك يكون ليس من الآمن نسخ تابِع لديه `super` من كائن ووضعه في آخر.
- كما وأنّ: ليس للدوال السهمية لا `this` ولا `super`

9.2.6 تمارين

ا. خطأ في اشتقاق صنف

الأهمية: ★★★★★

إليك الشيفرة التي نُوسّع فيها صنف `Rabbit` من صنف `Animal`. للأسف فلا يمكننا صناعة كائنات الأرانب. ما المشكلة؟ أصلِحها في طريقك.

```
class Animal {

  constructor(name) {
    this.name = name;
  }

```

```
}

class Rabbit extends Animal {
  constructor(name) {
    this.name = name;
    this.created = Date.now();
  }
}

// هنا
let rabbit = new Rabbit("White Rabbit"); // Error: this is not defined
alert(rabbit.name);
```

الحل:

هذا لأنّ على الباني الابن استدعاء `super()`. إليك الشيفرة الصحيحة:

```
class Animal {

  constructor(name) {
    this.name = name;
  }

}

class Rabbit extends Animal {
  constructor(name) {
    super(name); // هنا
    this.created = Date.now();
  }
}

let rabbit = new Rabbit("White Rabbit"); // الآن تمام
alert(rabbit.name); // White Rabbit
```

ب. توسعة ساعة

الأهمية: ★★★★★

لدينا صنف ساعة Clock، وهو حاليًا يطبع الوقت في كلّ ثانية.

```
class Clock {
  constructor({ template }) {
    this.template = template;
  }

  render() {
    let date = new Date();
    let hours = date.getHours();
    if (hours < 10) hours = '0' + hours;

    let mins = date.getMinutes();
    if (mins < 10) mins = '0' + mins;

    let secs = date.getSeconds();
    if (secs < 10) secs = '0' + secs;

    let output = this.template
      .replace('h', hours)
      .replace('m', mins)
      .replace('s', secs);
    console.log(output);
  }

  stop() {
    clearInterval(this.timer);
  }
  start() {
    this.render();
    this.timer = setInterval(() => this.render(), 1000);
  }
}
```

أنشِئ الصنف الجديد `ExtendedClock` ليرث من `Clock` وأضِف المُعامل `precision`، وهو عدد الملّي ثانية `ms` بين كلّ "تَكّة". يجب أن يكون مبدئيًا `1000` (أي ثانية كاملة).

- ضع شيفرتك في الملف `extended-clock.js`.
- تعديل ملف `clock.js` الأصلي ممنوع. وسّع الصنف.

يمكن الاعتماد على هذه البيئة التجريبية لحل التمرين.

الحل:

```
class ExtendedClock extends Clock {
  constructor(options) {
    super(options);
    let { precision = 1000 } = options;
    this.precision = precision;
  }

  start() {
    this.render();
    this.timer = setInterval(() => this.render(), this.precision);
  }
};
```

مشاهدة الحل في بيئة تجريبية.

ج. الأصناف تُوسّع الكائنات؟

الأهمية: ★★★★★

كما نعلم فالكائنات كلها ترث `Object.prototype` وتقدر على الوصول إلى توابِع الكائنات "العادية" مثل `hasOwnProperty` وغيرها. مثال سريع:

```
class Rabbit {
  constructor(name) {
    this.name = name;
  }
}

let rabbit = new Rabbit("Rab");
```

```
// التابع hasOwnProperty مأخوذ من Object.prototype
alert( rabbit.hasOwnProperty('name') ); // true
```

ولكن لو كتبنا ذلك جهارةً هكذا `"class Rabbit extends Object"` فالناتج يختلف عن `"class Rabbit"` فقط! غريب. تُراه ما الفرق؟

إليك مثالًا عمّا أقصد (الشيفرة لا تعمل، لماذا؟ أصلِحها!):

```
class Rabbit extends Object {
  constructor(name) {
    this.name = name;
  }
}

let rabbit = new Rabbit("Rab");

alert( rabbit.hasOwnProperty('name') ); // true
```

الحل:

أولًا نرى ما مشكلة الشيفرة تلك. متى شغّلنا الشيفرة بان سبب المشكلة: على باني الأصناف الموروثة استدعاء `super()` وإلّا تكون قيمة `"this"` "غير معرّفة". لذا سنصلحها:

```
class Rabbit extends Object {
  constructor(name) {
    super(); // علينا استدعاء الباني الأب حين نرث الصنف
    this.name = name;
  }
}

let rabbit = new Rabbit("Rab");

alert( rabbit.hasOwnProperty('name') ); // true
```

ولكن... لم ننتهِ بعد.

حتّى مع هذا ... فهناك فرق مهم جوهري بين `"class Rabbit extends Object"` و`class Rabbit`.

كما نعلم فصياغة `extends` تضبط كائنا prototype:

1. بين توابِع الباني لِـ`"prototype"` (بالنسبة للتوابع العادية).
2. بين توابِع الباني نفسها (بالنسبة للتوابع الثابتة).

في حالتنا إن `class Rabbit extends Object` تعني:

```
class Rabbit extends Object {}

alert( Rabbit.prototype.__proto__ === Object.prototype ); // (1) true
alert( Rabbit.__proto__ === Object ); // (2) true
```

لذا يوفر `Rabbit` إمكانية الوصول إلى الدوال الثابتة للكائن `Object`. هكذا:

```
class Rabbit extends Object {}

// Object.getOwnPropertyNames عادةً نستدعي
alert ( Rabbit.getOwnPropertyNames({a: 1, b: 2})); // a,b
```

ولكن إذا لم يكن لدينا `extends Object` فلن تُسند `Rabbit.__proto__` للصنف `Object`.

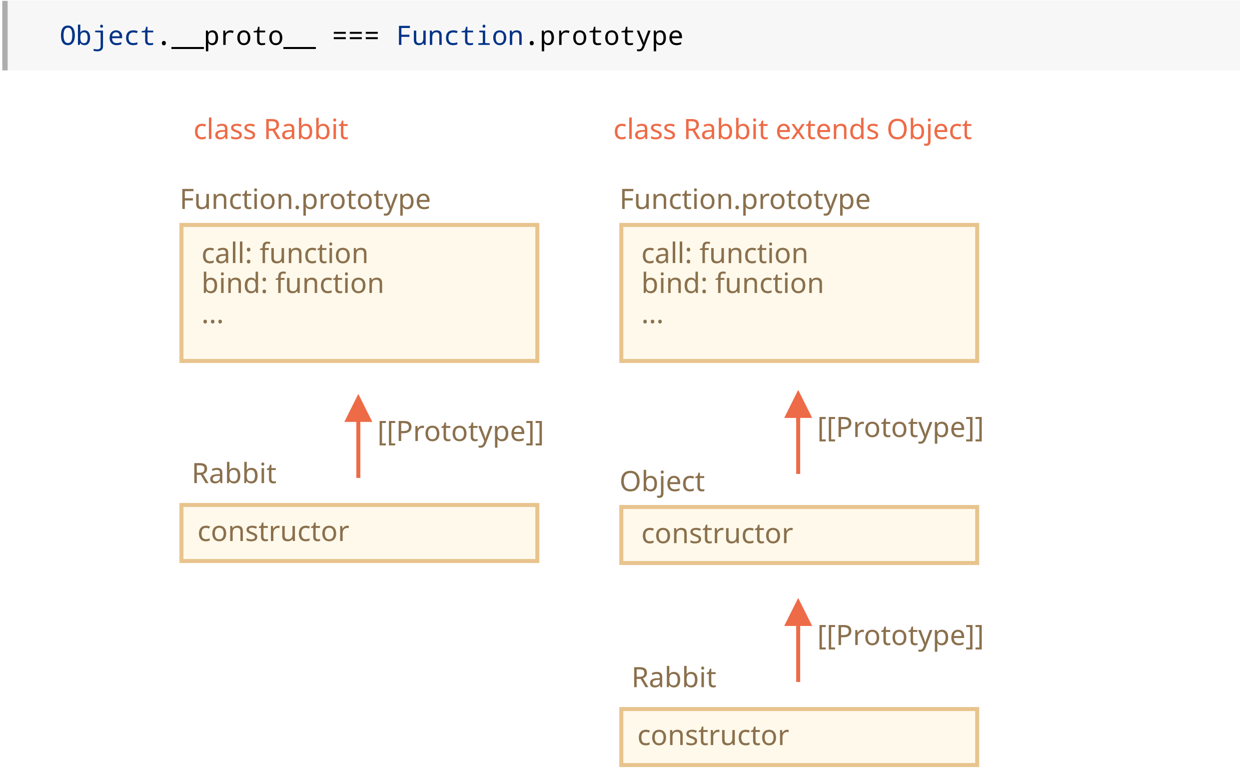
إليك المثال:

```
class Rabbit {}

alert( Rabbit.prototype.__proto__ === Object.prototype ); // (1) true
alert( Rabbit.__proto__ === Object ); // (2) false (!)
alert( Rabbit.__proto__ === Function.prototype ); // as any function
by default

// error, no such function in Rabbit
alert ( Rabbit.getOwnPropertyNames({a: 1, b: 2})); // Error
```

في هذه الحالة إن `Rabbit` لن يزودنا بطريقة للوصول إلى التوابع الثابتة في `Object`.

بالمناسبة يملك النموذج الأولي للتوابع `Function.prototype` دوال مُعمَّمة مثل: `call` و `bind` ...إلخ. وهي متاحة دائمًا في كِلا الحالتين، لأن باني `Object` المضمن في اللغة هو:

```
Object.__proto__ === Function.prototype
```



لذلك وباختصار هناك اختلافان وهما:

class Rabbit	class Rabbit extends Object
--	يحتاج لاستدعاء `super()` في الباني
`Rabbit.__proto__ === Function.prototype`	`Rabbit.__proto__ === Object`

9.3 الخاصيات والتوابع الثابتة

يمكننا أيضًا إسناد التوابع إلى دالة الصنف ذاتها وليس إلى كائن "prototype" لها. نسمّي هذه التوابع بالتوابِع "الثابتة" static methods. في الأصناف نضع بعدها كلمة static هكذا:

```
class User {
  static staticMethod() { // لاحظ
    alert(this === User);
  }
}

User.staticMethod(); // true
```

في الواقع، لا يفرق هذا عن إسنادها على أنّها خاصية بشيء:

```
class User() { }

User.staticMethod = function() {
  alert(this === User);
};

User.staticMethod(); // true
```

وتكون قيمة this في الاستدعاء User.staticMethod() هي باني الصنف User ذاته (تذكّر قاعدة "الكائن قبل النقطة").

عادةً ما نستعمل التوابع الثابتة لكتابة دوال تعود إلى الصنف نفسه وليس إلى أيّ كائن من ذلك الصنف.

مثال للتوضيح: لدينا كائنات فصلات Article ونريد دالة لموازنتها. الحل الطبيعي هو إضافة تابِع Article.compare هكذا:

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }

  // هنا
```

```
  static compare(articleA, articleB) {
    return articleA.date - articleB.date;
  }
}

// الاستعمال
let articles = [
  new Article("HTML", new Date(2019, 1, 1)),
  new Article("CSS", new Date(2019, 0, 1)),
  new Article("JavaScript", new Date(2019, 11, 1))
];

articles.sort(Article.compare); // لاحظ

alert( articles[0].title ); // CSS
```

نرى هنا التابع `Article.compare` "فوق" الفصلات فيتيح لنا طريقة لموازنتها. ليس التابع تابعًا للفصلة ذاتها، بل لصنف الفصلات نفسه.

توابِع "المصانع" (factory) تنفعها أيضًا التوابع الثابتة هذه. لنقل بأنّا نريد إنشاء الفصلات بطرائق عدّة:

1. بتمرير المُعاملات (العنوان `title` والتاريخ `date` وغيرها).
2. إنشاء فصلة فارغة تحمل تاريخ اليوم.
3. أو... كما تريد أنت.

يمكننا تنفيذ الطريقة الأولى باستعمال بانٍ، وللثانية صناعة تابِع ثابت للصنف. مثل التابع `Article.createTodays()` هنا:

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }

  static createTodays() {
    // تذكّر: this = Article
    return new this("Today's digest", new Date()); // موجز أحداث اليوم
```

```
    }
  }

  let article = Article.createTodays();

  alert( article.title ); // موجز أحداث اليوم
```

الآن متى أردنا صناعة موجز عن أحداث اليوم، استدعينا `Article.createTodays()`. أُعيد، ليس هذا تابِعًا للفصلة نفسها بل تابِعًا للصنف كله.

كما نستعمل التوابع الثابتة في أصناف قواعد البيانات للبحث عن المُدخلات وحفظها وإزالتها، هكذا:

```
// بفرض أنّ Article هو صنف مخصّص لإدارة الفصلات
// نستعمل تابِعًا ثابتًا لإزالة الفصلة:
Article.remove({id: 12345});
```

9.3.1 الخاصيات الثابتة

انتبه رجاءً إلى أنه هذه الميزة مضافة حديثًا إلى اللغة، لذا لن تعمل الأمثلة إلا في الإصدارات الحديث من المتصفح كروم.

كما يمكننا أيضًا استعمال الخاصيات الثابتة. ظاهرها فهي خاصيات للأصناف، ولكن نضع قبلها عبارة `static`:

```
class Article {
  static publisher = "Ilya Kantor";
}

alert( Article.publisher ); // Ilya Kantor
```

لا يفرق هذا عن الإسناد المباشر إلى صنف `Article`:

```
Article.publisher = "Ilya Kantor";
```

9.3.2 وراثة الخاصيات والتوابع الثابتة

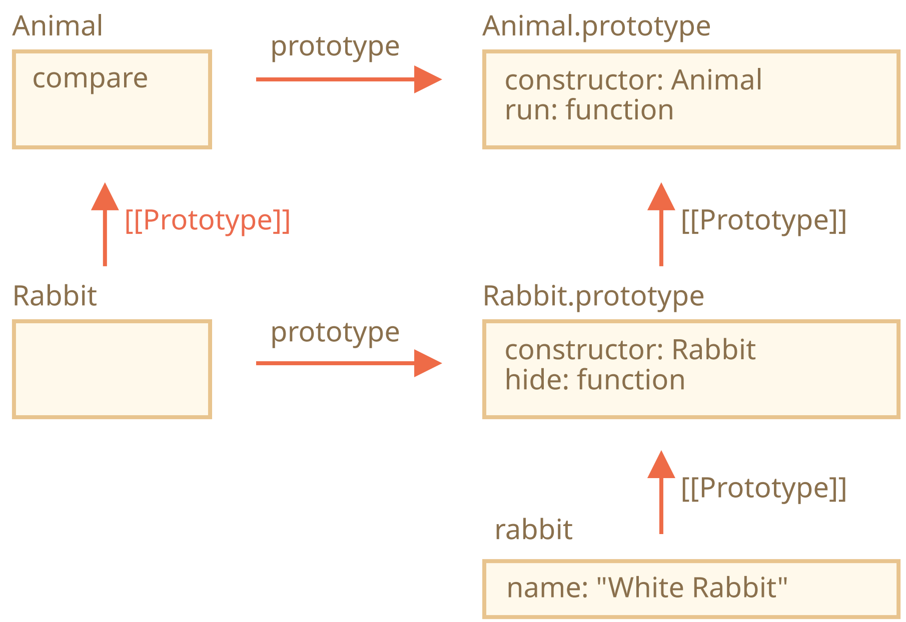
هذه الأنواع من الخاصيات والتوابع

فمثلًا التابع `Animal.compare` والخاصية `Animal.planet` في الشيفرة أسفله موروثين بالأسماء `Rabbit.compare` و `Rabbit.planet`:

```
class Animal {
  static planet = "Earth"; // الأرض

  constructor(name, speed) {
    this.speed = speed;
    this.name = name;
  }

  run(speed = 0) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }

  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }

}

// نرث من الصنف Animal
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
}

let rabbits = [
  new Rabbit("White Rabbit", 10),
  new Rabbit("Black Rabbit", 5)
];

rabbits.sort(Rabbit.compare); // لاحظ

rabbits[0].run(); // Black Rabbit runs with speed 5.

alert(Rabbit.planet); // الأرض
```

الآن متى استدعينا `Rabbit.compare`، استدعى المحرّك التابع `Animal.compare` الموروث.

ولكن كيف يعمل هذا الشـيء؟ كالعـادة، بكائنـات prototype: تُقـدّم `extends` للصـنف `Rabbit` إشـارة `[[Prototype]]` إلى الصنف `Animal`.



لذا فعبارة `Rabbit extends Animal` تصنع إشارتي `[[Prototype]]`:

1. دالة `Rabbit` موروثة عبر prototype من دالة `Animal`.
2. كائن `Rabbit.prototype` موروث عبر prototype من كائن `Animal.prototype`.

بهذا تعمل الوراثة للتوابِع العادية والثابتة معًا. تشكّ؟ انظر للشيفرة:

```
class Animal {}
class Rabbit extends Animal {}

// لكلّ ما هو ثابت
alert(Rabbit.__proto__ === Animal); // true

// للتوابِع العادية
alert(Rabbit.prototype.__proto__ === Animal.prototype); // true
```

9.3.3 الخلاصة

نستعمل التوابع الثابتة لأيّة وظائف تخصّ الصنف "كلّه على بعضه"، ولا يخصّ نسخة معيّنة من الصنف.

مثـــل تـــابع الموازنـــة `Article.compare(article1, article2)` أو تـــابع المصـــنع `Article.createTodays()`. ويصنفون كثوابت (`static`) في تعريف الصنف.

نستعمل الخاصيات الثابتة متى أردنا تخزين البيانات على مستوى الصنف لا على مستوى النُسخ المُشتقَّة.

صياغتها هي:

```
class MyClass {
  static property = ...;

  static method() {
    ...
  }
}
```

تقنيًا فالتصريح الثابت (static declaration) لا يفرق عن الإسناد إلى الصنف مباشرةً:

```
MyClass.property = ...
MyClass.method = ...
```

الخاصيات والدوال الثابتة الموروثة.

من أجل العبارة `class B extends A` إن prototype للصنف `B` يشير إلى:

```
A:B.[[Prototype]] = A
```

لذا إن تعذر العثور على خاصية ما في الصنف `B` فسيستمر البحث في الصنف `A`.

9.4 الخاصيات والتوابِع الخاصّة والمحمية

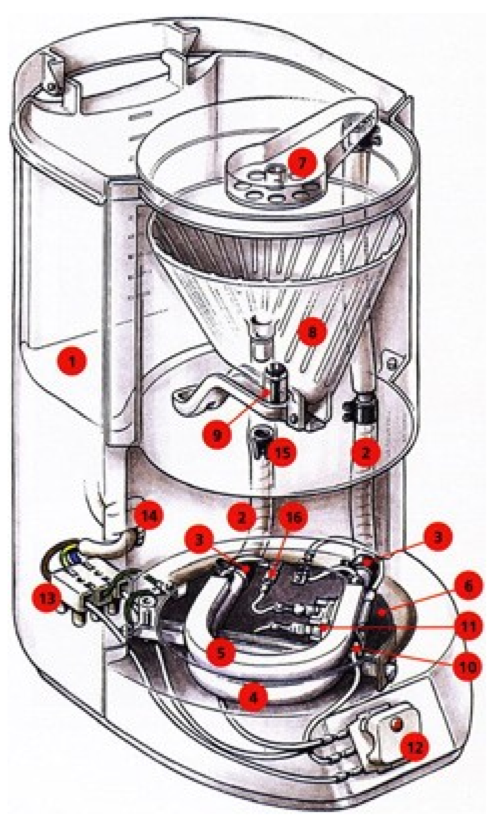
في البرمجة كائنية التوجّه، يُعدّ أكثر المبادئ أهميـة هـو فصـل الواجهـة الداخليـة عن الخارجيـة. هـذا المبـدأ "إلزامي" متى ما طوّرنا تطبيقًا يفوق تطبيقات "أهلًا يا عالم" تعقيدًا. سنضع البرمجة "على جنب" ونـرى الواقـع؛ لنفهم هذا المبدأ.

الأجهزة الّتي نستعملها يوميًا عادةً ما تكون معقّدة، إلّا أنّ فصل واجهتها الداخلية عن تلك الخارجية يُتيح لنـا استعمالها دون مشاكل تُذكر.

9.4.1 مثال من الحياة العملية

لنقل مثلًا آلة صنع القهوة، من الخارج بسيطة: زرّ تشغيل وشاشة عـرض وبعض الفتحـات وطبعًـا لا ننسـى الناتج… القهوة المميّزة الطعم! :)

ولكن من الداخل نرى… (هذه الصورة من كتيّب التصليح):



نرى تفاصيل وتفاصيل ولا شيء إلّا التفاصيل، ولكنّنا نستعملها دون أيّ فكرة عن تلك التفاصيل. يمكننا أن نثق بجودة آلات صنع القهوة، ألا توافقني الرأي؟ فنحن نستعملها لسنوات وسنوات وإن ظهر أيّ عُطل، أخذناها للتصليح.

يكمن سرّ هذه الثقة وهذه البساطة لآلات صنع القهوة في أنّ التفاصيل كلّها مضبوطة كما يجب ومخفية داخلها. إن أزلنا غطاء الحماية من الآلة وحاولنا استعمالها لكان الأمر أكثر تعقيدًا (أيّ زرّ نضغط؟) وخطرًا (الصدمات الكهربية).

كما سنرى أسفله، فالكائنات في البرمجة تشبه تمامًا آلات صنع القهوة. ولكن لنُخفي التفاصيل الداخلية لن نستعمل غطاء حماية بل صياغة خاصّة في اللغة، كما وما هو متّفق عليه بين الناس.

9.4.2 الواجهتان الداخلية والخارجية

في البرمجة كائنية التوجّه، تنقسم الخاصيات والتوابِع إلى قسمين:

- واجهة داخلية (Internal interface): فيها التوابع والخاصيات الّتي يمكن أن تصل إليها توابِع هذا الصنف، ولا يمكن أن يصل إليها ما خارج هذا الصنف.
- واجهة خارجية (External interface): فيها التوابع والخاصيات الّتي يمكن أن يصل إليها ما خارج هذا الصنف أيضًا.

لو واصلنا الشرح على آلة القهوة تلك، فهذا بعض ممّا هو مخفي: أنبوب غلي، عنصر التسخين، وغيرهما من العناصر الداخلية التي تشكل الواجهة الداخلية الخلفية الخفية حيث نستعمل الواجهة الداخلية ليعمل هذا الشيء (أيًا كان)، إذ ترى تفاصيله هذه تلك. ولكن من الخارج نجد أن آلة القهوة لهـا غلاف محكم الإغلاق فلا يمكن لأحـد أن يصل إلى تلك التفاصيل، فتكون مخفيّة ولا يمكن استعمال ميزات الآلة إلا عبر الواجهة الخارجية.

إذًا، ما نريده لنستعمل الكائنات هو معرفة واجهته الخارجيـة. أمّـا معرفـة طريقـة عملـه بالتفاصـيل الدقيقـة ليس أمرًا مهمًا أبدًا، وهذا في الواقع تسهيل عظيم.

شرحنا الآن الفكرة بشكلٍ عام. أمّا في جافاسكربت فهناك نوعين من حقول الكائنات (الخاصيات والتوابِع):

- عامّة: يمكن أن نصـل إليهـا من أيّ مكـان، وهي تصـنع تلـك الواجهـة الخارجيـة. حتّى اللحظـة في هـذا الكتاب كنّا نستعمل الخاصيات والتوابِع العموميّة فقط.
- خاصّة: يمكن أن نصل إليها من داخل الصنف فقط، وهي تصنع الواجهة الداخلية.

كما هناك في اللغات الأخرى حقول "محميّة": أي نصل إليها من داخل الصنف ومن أيّ صنف آخـر يوسّـعه (تعمل مثل الخاصّة، علاوةً على الوصول إليها من الأصناف الموروثة). هذه الحقول مفيدة جدًا للواجهة الداخليـة، وهي منطقيًا أكثر استعمالاً من الخاصّة إذ حين نرث صنفًا نريدُ أن نصل إلى تلك الحقول الّتي فيه عادةً.

ليس هناك تعريف لتنفيذ هذه الحقول في جافاسـكربت على مسـتوى اللغـة، ولكنّهـا عمليًـا تُسـهّل الأمـور كثيرًا، بذلك نحاكي عملها.

الآن حان وقت صناعة آلة قهوة بجافاسكربت مسـتعملين تلـك الأنـواع من الخاصـيات. طبعًـا لآلـة القهـوة تفاصيل عديدة ولن نضعها كلها في صنفنا لتسهيل الأمور (ولكن ما من مانع لتفعل أنت ذلك).

9.4.3 حماية "مقدار الماء"

لنصنع أولًا صنفًا بسيطًا لآلة قهوة:

```
class CoffeeMachine {
  waterAmount = 0; // مقدار الماء في الآلة

  constructor(power) {
    this.power = power;
    alert( `Created a coffee-machine, power: ${power}` ); // صنعنا آلة قهوة بقوّة كذا
  }

}
```

```
// نصنع آلة قهوة واحدة
let coffeeMachine = new CoffeeMachine(100);

// نُضيف إليها الماء
coffeeMachine.waterAmount = 200;
```

حاليًا فخاصيتي مقدار الماء `waterAmount` والقوّة `power` عامّتين، ويمكننا بسهولة جلبهما أو ضبطهما من خارج الصنف لأيّ قيمة نريد.

لنغيّر خاصية `waterAmount` فتصير محميّة، بذلك يكون في يدنا التحكّم. لنقل مثلًا بأنّه يُمنع ضبط القيمة إلى ما دون الصفر.

عادةً ما نضع شَرطة سفلية _ قبل أسماء الخاصيات المحميّة. لا تفرض اللغة هذا الأمر ولكنّه اتّفاق بين معشر المطوّرين بأنّه ممنوع الوصول إلى هذه الخاصيات والتوابِع من خارج الأصناف.

إذًا، ستكون الخاصية باسم `_waterAmount`:

```
class CoffeeMachine {
  _waterAmount = 0;

  set waterAmount(value) {
    if (value < 0) throw new Error("Negative water"); // قيمة الماء بالسالب
    this._waterAmount = value;
  }

  get waterAmount() {
    return this._waterAmount;
  }

  constructor(power) {
    this._power = power;
  }

}

// نصنع آلة القهوة المميّزة
let coffeeMachine = new CoffeeMachine(100);
```

```
// نُضيف الماء
coffeeMachine.waterAmount = -10; // خطأ: قيمة الماء بالسالب
```

الآن صرنا نتحكّم بالوصول إلى الخاصية، ومحاولة ضبط قيمة الماء إلى ما دون الصفر ستفشل.

9.4.4 "القوّة" للقراءة فقط

أمّا عن خاصية القوّة power فسنجعلها للقراءة فقط. نواجه أحيانًا مواقف حيث تُضبط الخاصية أثناء إنشاء الصنف فقط، ويُمنع تعديلها قطعيًا بعد ذلك.

هذه هي حالة آلة القهوة هنا، إذ قوّة الآلة لا تتغيّر أبدًا. لذلك سنصنع جالبًا فقط دون ضابِط:

```
class CoffeeMachine {
  // ...

  constructor(power) {
    this._power = power;
  }

  get power() {
    return this._power;
  }

}

// نصنع آلة القهوة
let coffeeMachine = new CoffeeMachine(100);

alert(`Power is: ${coffeeMachine.power}W`); // القوّة هي: 100 واط

coffeeMachine.power = 25; // خطأ (لا ضابِط)
```

لاحظ، استخدامنا هنا صياغة الجوالِب/الضوابِط (Getter/setter) الافتراضية ولكن دوالّ get.../set... مفضلة في معظم الأحيان. هكذا:

```
class CoffeeMachine {
  _waterAmount = 0;

```

```
    setWaterAmount(value) {
      if (value < 0) throw new Error("Negative water");
      this._waterAmount = value;
    }

    getWaterAmount() {
      return this._waterAmount;
    }
  }

  new CoffeeMachine().setWaterAmount(100);
```

تبدو هذه الصياغة أطول قليلًا ولكن الدوال هنا أكثر مرونة. إذ يمكننا تمرير عدة وسطاء لهم (حتى ولو لم نحتاج لهذا الأمر في هذا المثال).

من ناحية أخرى إن صياغة الافتراضية أقصر لذا لا توجد قاعدة صارمة للأمر، ولك مطلق الحرية في الاختيار.

الخاصيات المحميّة موروثة

إذا ورثنا `class MegaMachine extends CoffeeMachine`، فلا شيء سيمنعنا من الوصول إلى `this._waterAmount` أو `this._power` من توابع الصنف الجديد، لذا فالوضع الطبيعي للخاصيات المحمية أن تكون قابلة للتوريث. على عكس الخاصيات الخاصة والتّي سنراها أدناه.

9.4.5 "#حد الماء" خاصة

انتبه رجاءً إلى أن هذه الميزة أضيفت حديثًا إلى اللغة، لذا هي ليست مدعومة على محركات جافاسكربت بعد ولا حتى جزئيًا وتحتاج إلى ترقيع يسد نقص الدعم هذا (polyfilling).

هناك مُقترح للغة جافاسكربت (كاد أن يصل إلى المعيار) يُقدّم لنا دعمًا على مستوى اللغة للخاصيات والتوابِع الخاصة.

تبدأ الخاصيات والتوابِع بعلامة `#`، ولا يمكن الوصول إليها إلّا من داخل الصنف. فمثلًا إليك خاصية `#waterLimit` الخاصّة وتابِع فحص مستوى الماء `#checkWater` الخاصّ:

```
  class CoffeeMachine {
    #waterLimit = 200; // هنا

    // هنا
    #checkWater(value) {
```

```
    if (value < 0) throw new Error("Negative water"); // قيمة الماء بالسالب
    if (value > this.#waterLimit) throw new Error("Too much water");
// قيمة الماء فوق اللزوم
  }

}

let coffeeMachine = new CoffeeMachine();

// محاولة الوصول إلى الخاصيات والتوابع الخاصّة من خارج الصنف
coffeeMachine.#checkWater(); // Error
coffeeMachine.#waterLimit = 1000; // Error
```

علامة # على مستوى اللغة هي علامة خاصّة تقول بأنّ هذا الحقل حقلٌ خاصّ، وتمنع أيّ شيء من الوصول إليه من الخارج أو من الأصناف الموروثة.

كما وأنّ الحقول الخاصّة لا تتضارب مع تلك العامّة: يمكن أن نضع حقلين خاصًا ‎#waterAmount‎ وعامًا ‎waterAmount‎ في آنٍ واحد. فمثلًا ليكن ‎waterAmount‎ خاصية وصول للخاصية ‎#waterAmount‎:

```
class CoffeeMachine {

  #waterAmount = 0;

  get waterAmount() {
    return this.#waterAmount;
  }

  set waterAmount(value) {
    if (value < 0) throw new Error("Negative water"); // قيمة الماء بالسالب
    this.#waterAmount = value;
  }
}

let machine = new CoffeeMachine();

machine.waterAmount = 100;
alert(machine.#waterAmount); // خطأ
```

وعلى عكس الحقول المحميّة، فتلك الخاصّة تفرضها اللغة نفسها، وهذا أمر طيّب. ولكن لو ورثنا شيئًا من `CoffeeMachine` فلن يمكننا الوصول إلى `#waterAmount` مباشرةً، بل الاعتماد على جالِب وضابِط `waterAmount`:

```
class MegaCoffeeMachine extends CoffeeMachine {
  method() {
    alert( this.#waterAmount ); // خطأ: يمكنك أن تصل إليه من داخل
CoffeeMachine فقط
  }
}
```

هناك حالات عديدة يكون فيها هذا القيد صارم كثيرًا. حين نُوسّع `CoffeeMachine` يكون لدينا أسباب منطقية لنصل إلى خوارزمياته الداخلية. لهذا السبب فالحقول المحميّة أكثر فائدة بكثير حتّى لو لم تدعمها صياغة اللغة.

لا يمكنك الوصول إلى الحقول الخاصّة هكذا: this[name]

إذ عادةً ما يمكننا الوصول إلى الحقول العادية هكذا : this[name]

```
class User {
  ...
  sayHi() {
    let fieldName = "name";
    alert(`Hello, ${this[fieldName]}`);
  }
}
```

ولكن مع الحقول الخاصة من المحال الوصول إليها ['this['#name فإنها لن تعمل. وذلك بسبب القيد الموجود على صياغة استدعاء الحقول الخاصة لضمان خصوصية الحقل.

9.4.6 الخلاصة

لو استعرنا تسميات البرمجة كائنية التوجّه، فعملية فصل الواجهة الداخلية عن الخارجية تُسمّى تغليف Encapsulation، وهي تقدّم لنا فوائد منها:

حماية المستخدمين كي لا يضروا أنفسهم: فتخيّل فريق مطوّرين يستعمل آلةً للقهوة صنعتها شركة "Best CoffeeMachine"، وهي تعمل كما يجب لكنّها دون غطاء حماية، أيّ أنّ الواجهة الداخلية مكشوفة للعيان.

المطوّرون متحضّرون فيستعملون الآلة كما يفترض استعمالها، ولكنّ أحدهم (واسمه Ahmad) قال بأنّه الأذكى بين الجميع وعدّل على ميكانيكا الآلة الداخلية قليلًا. بعد يومين، لم تعد تعمل.

هذه ليست غلطة Ahmad طبعًا، بل الشخص الّذي أزال غطاء الحماية وسـمح لأشـخاص (مثـل Ahmad) أن يقوموا بهذه التعديلات.

الأمر ذاته في البرمجة. فلو غيّر أحد مستخدمي الصنف أشياء لا يُفـترض تغييرهـا من خـارج الكـائن، يكـون التكهّن بعواقب ذلك مستحيلًا.

تحظى بالدعم: تختلف البرمجة عن آلة القهوة الحقيقية إذ الأولى معقّـدة أكـثر لأنّنـا لا نشـتريها مـرّة واحـدة فقط، بل تمرّ الشيفرة في مراحل تطوير وتحسين عديدة لا تنتهي أحيانًا.

إن أجبرنا فصل الواجهة الداخلية عن الخارجية، فيمكن للمطوّر تغيير خاصـيات الأصـناف وتوابِعهـا الداخليـة كما يشاء دون أن يُبلغ مستخدميها حتّى.

لو كنت تطوّر صنفًا مثل هذا، ففكرة أنّ تغيير اسم التوابع الخاصة أو تغيـير مُعاملتهـا أو حتّى إزالتهـا - فكـرة رائعة إذ ليست هناك أيّ شيفرة خارجية تعتمد عليها.

أمّا للمستخدمين، فلو صدرت نسخة جديدة (وقد تكون كتابة كاملةً داخليًا) سـيكون سـهلًا جـدًا الترقية إليها لو بقيت الواجهة الخارجية كما هي.

إخفاء التعقيد: يعشق الناس استعمال ما هو بسيط، على الأقلّ من الخارج. الداخل لا يهمّ. وليس المـبرمج استثناءً. دومًا ما يكون أفضل لو كانت تفاصيل التنفيذ مخفية، وكانت هناك واجهة خارجية بسـيطة موثّقـة كمـا يجب. لنُخفي الواجهات الداخلية، نستعمل إمّا الخاصيات المحميّة أو الخاصّة:

تبدأ الحقول المحميّة بعلامة الشرطة السفلية _. هذا اتّفـاق بين معشـر المـبرمجين وليس فرضًـا من اللغـة نفسها. على المبرمجين الوصول إلى الحقول الّتي تبـدأ بهـذه العلامـة داخـلَ الصـنف والأصـناف الموروثـة فقط لا غير.

تبدأ الحقول الخاصّة بعلامة #. تضمن لنا لغة جافاسكربت بأن لا يقدر أحد على الوصـول إليهـا إلّا من داخـل الصنف. حاليًا فالحقول الخاصّة ليست مدعومة تمامًا في المتصفّحات، ولكن يمكننا تعويض نقص الدعم هذا.

9.5 توسعة الأصناف المضمّنة في اللغة

يمكننا أيضًا توسيع الأصناف المضمّنة مثل المصفوفات والخرائط وغيرها. فمثلًا يرث صنف PowerArray من المصفوفة Array الأصيلة:

```
// نُضيف تابعًا آخر إليها (أو أكثر لو أردنا)
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // false
```

من الجدير بالملاحظة أن التوابع المضمّنة في اللغة مثل: filter و map وغيرهم، تعيد كائنات جديدة من نفس النوع الموروث بالضبط PowerArray. وذلك لأن التطبيق الداخلي للتوابع يستخدم الباني المخصص لتلك الكائنات. في المثال أعلاه:

```
arr.constructor === PowerArray
```

متى استدعينا arr.filter() أنشأ التابع داخليًا مصفوفةً جديدة من النتائج باستعمال arr.constructor ذاتها، وليس المصفوفة العادية Array. هذا أمر رائع جدًا إذ يمكننا استعمال توابِع PowerArray على النتائج أيضًا.

يمكننا حتّى تخصيص هذا السلوك كما نرغب. فنُضيف جالِبًا ثابتًا Symbol.species إلى الصنف. لو كان موجودًا فسيُعيد الباني الذي ستستعمله جافاسكربت داخليًا لإنشاء المدخلات الجديدة في التوابع map و filter وغيرها.

لو أردنا من التوابع المضمّنة مثل map و filter - لو أردنا أن تُعيد المصفوفات الطبيعية، فعلينا إعادة صنف Array في رمز Symbol.species هكذا:

```
class PowerArray extends Array {
  isEmpty() {
```

```
      return this.length === 0;
    }

    // ستستعمل التوابع المضمَنة هذا الصنف ليكون بانيًا
    static get [Symbol.species]() {
      return Array;
    }
  }

  let arr = new PowerArray(1, 2, 5, 10, 50);
  alert(arr.isEmpty()); // false

  // يصنع التابع  filter مصفوفةً جديدة باستعمال arr.constructor[Symbol.species] بانيًا
  let filteredArr = arr.filter(item => item >= 10);

  // نوع  filteredArr ليس PowerArray بل Array
  alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is not a function
```

كما نرى فالآن يُعيد التابع `.filter` مصفوفةً `Array`. بذلك تلك الميزة الموسّعة لن تُمرّر أكثر وأكثر.

> هنالك مجموعات أخرى تعمل بنفس الطريقة مثل: `Map` أو `Set` ويستخدمون أيضًا `Symbol.species`.

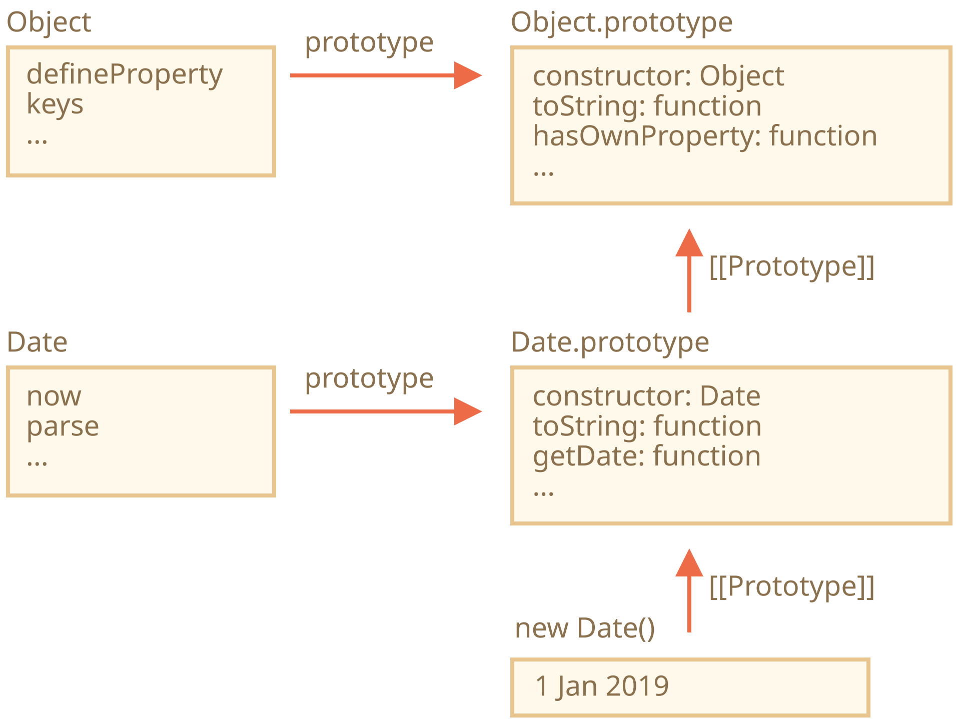
9.5.1 الأنواع المضمّنة لا ترث الثوابت

للكائنات المضمّنة أيضًا توابِع ثابتة مثل `Object.keys` و `Array.isArray` وغيرها. وكما نعلم فالأصناف الأصيلة تُوسّع نفسها. فمثلًا تُوسّع المصفوفات `Array` الكائناتَ `Object`.

وعادةً متى وسّع الصنف صنفًا آخر ورث التوابع الثابتة وغير الثابتة. شرحنا هذا بالتفصيل في الفصل "الخاصيات والتوابع الثابتة".

ولكن الأصناف المضمّنة في اللغة استثناء لهذا، ولا ترث الحقول الثابتة من بعضها البعض. مثال: ترث المصفوفات والتواريخ الكائناتَ، بذلك نرى لنُسخها المُشتقة توابِع أتت من `Object.prototype`. ولكنّ كائن `Array.[[Prototype]]` لا يُشير إلى `Array`، بذلك لا نرى توابِع ثابتة مثل `Array.keys()` أو `Array.keys()` مثلًا.

تغني الصورة عن ألف كلمة، إليك واحدة توضّح بنية التواريخ `Date` والكائنات `Object`:



كما ترى فليس هناك رابط بين التواريخ والكائنات، فهي مستقلة بذاتها. كائن `Date.prototype` فقط هو من يرث `Object.prototype`.

هذا فرق مهمّ للوراثة بين الكائنات المضمّنة في اللغة وتلك التي نحصل عليها باستعمال `extends`.

9.6 فحص الأصناف عبر instanceof

يُتيح لنا المُعامل `instanceof` (أهو نسخة أو مُشتَق من) فحص هل الكائن ينتمي إلى الصنف الفلاني؟ كما يأخذ الوراثة في الحسبان عند الفحص.

توجد حالات عديدة يكون فيها هذا الفحص ضروريًا. سنستعمله هنا لصناعة دالة ، أي دالة تغيّر تعاملها حسب نوع الوسطاء الممرّرة لها.

9.6.1 معامل instanceof

صياغته هي:

```
obj instanceof Class
```

يُعيد المُعامل قيمة `true` لو كان الكائن `obj` ينتمي إلى الصنف `Class` أو أيّ صنف يرثه.

مثال:

```
class Rabbit {}
let rabbit = new Rabbit();

// هل هو كائن من الصنف Rabbit؟
alert( rabbit instanceof Rabbit ); // true نعم
```

كما يعمل مع الدوال البانية:

```
// بدل استعمال class
function Rabbit() {}

alert( new Rabbit() instanceof Rabbit ); // true نعم
```

كما والأصناف المضمّنة مثل المصفوفات `Array`:

```
let arr = [1, 2, 3];
alert( arr instanceof Array ); // true نعم
alert( arr instanceof Object ); // true نعم
```

لاحظ كيف أنّ الكائن `arr` ينتمي إلى صنف الكائنات `Object` أيضًا، إذ ترث المصفوفات -عبر prototype- الكائنات.

عادةً ما يتحقّق المُعامل `instanceof` من سلسلة prototype عند الفحص. كما يمكننا وضع المنطق الذي نريد لكلّ صنف في التابع الثابت `Symbol.hasInstance`.

تعمل خوارزمية `obj instanceof Class` بهذه الطريقة تقريبًا:

أولًا، لو وجدت التابع الثابت `Symbol.hasInstance` تستدعيه وينتهي الأمر `Class[Symbol.hasInstance](obj)`. يُعيد التابع إمّا `true` وإمّا `false`. هكذا نخصّص سلوك المُعامل `instanceof`. مثال:

```
// ضبط instanceOf للتحقق من الافتراض القائل
// بأن كل شيء يملك الخاصية canEat هو حيوان
class Animal {
  static [Symbol.hasInstance](obj) {
    if (obj.canEat) return true;
  }
}

let obj = { canEat: true };

alert(obj instanceof Animal); // true
// تستدعى Animal[Symbol.hasInstance](obj)
```

ثانيًا، ليس لأغلب الأصناف التابع `Symbol.hasInstance`. في هذه الحالة تستعمل المنطق العادي: يفحص `obj instanceOf Class` لو كان كائن `Class.prototype` مساويًا لأحد كائنات prototype في سلسلة كائنات prototype للكائن `obj`.

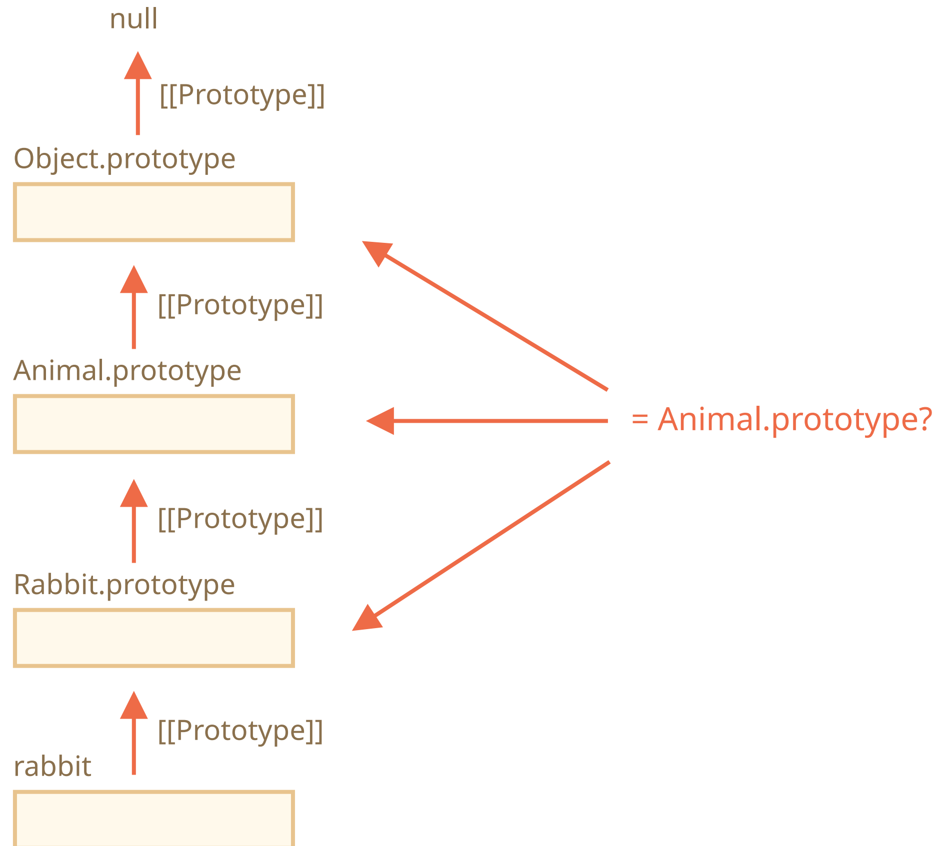
وبعبارة أخرى ، وازن بينهم واحدًا تلو الآخر:

```
obj.__proto__ === Class.prototype?
obj.__proto__.__proto__ === Class.prototype?
obj.__proto__.__proto__.__proto__ === Class.prototype?
...
// لو كانت إجابة أيًا منها true، فتُعيد true
// وإلّا متى وصلت نهاية السلسلة أعادت false
```

في المثال أعلاه نرى `rabbit.__proto__ === Rabbit.prototype`، بذلك تُعطينا الجواب مباشرةً. أمّا لو كنّا في حالة وراثة، فستتوقّف عملية المطابقة عند الخطوة الثانية:

```
class Animal {}
class Rabbit extends Animal {}

let rabbit = new Rabbit();
alert(rabbit instanceof Animal); // نعم (هنا)

// rabbit.__proto__ === Rabbit.prototype
// rabbit.__proto__.__proto__ === Animal.prototype (تطابق!)
```

إليك صورة توضّح طريقة موازنة `rabbit instanceof Animal` مع `Animal.prototype`:



كما وهناك أيضًا التابع objA.isPrototypeOf(objB)، وهو يُعيد `true` لو كان الكائن `objA` في مكان داخل سلسلة prototype للكائن `objB`. يعني أنّنا نستطيع كتابة الفحص هذا `obj instanceof Class` هكذا: `Class.prototype.isPrototypeOf(obj)`.

الأمر مضحك إذ أنّ باني الصنف `Class` نفسه ليس لديه أيّ كلمة عن هذا الفحص! المهم هو سلسلة prototype وكائن `Class.prototype` فقط.

يمكن أن يؤدّي هذا إلى عواقب مثيرة متى تغيّرت خاصية `prototype` للكائن بعد إنشائه. طالع:

```
function Rabbit() {}
let rabbit = new Rabbit();

// غيّرنا كائن prototype
Rabbit.prototype = {};

// ...لم يعد أرنبًا بعد الآن!
alert( rabbit instanceof Rabbit ); // false لا
```

9.6.2 التابع Object.prototype.toString للأنواع

نعلم بأنّ الكائنات العادية حين تتحوّل إلى سلاسل نصية تصير `[object Object]`:

```
let obj = {};

alert(obj); // [object Object]
alert(obj.toString()); // كما أعلاه
```

يعتمد هذا على طريقة توفيرهم لتنفيذ التابع `toString`. ولكن هناك ميزة مخفيّة تجعل هذا التابع أكثر فائدة بكثير ممّا هو عليه، أن نستعمله على أنّه مُعامل `typeof` موسّع المزايا، أو بديلًا عن التابع `toString`.

تبدو غريبة؟ أليس كذلك؟ لنُزل الغموض.

حسب المواصفة، فيمكننا استخراج التابع `toString` المضمّن من الكائن وتنفيذه في سياق أيّ قيمة أخرى نريد، وسيكون ناتجه حسب تلك القيمة.

- لو كان عددًا، فسيكون `[object Number]`
- لو كان قيمة منطقية، فسيكون `[object Boolean]`
- لو كان `null`: `[object Null]`
- لو كان `undefined`: `[object Undefined]`
- لو كانت مصفوفة: `[object Array]`
- …إلى آخره (ويمكننا تخصيص ذلك).

هيا نوضّح:

```
// نسخ التابع toString إلى متغير ليسهل عملنا
let objectToString = Object.prototype.toString;
```

```
// ما هذا النوع؟
let arr = [];

alert( objectToString.call(arr) ); // [object Array] مصفوفة!
```

استعملنا هنا call كما وضّحنا في فصل "المزخرفات والتمرير" لتنفيذ الدالة objectToString بسياق this=arr. تفحص خوارزمية toString داخليًا قيمة this وتُعيد الناتج الموافق لها. إليك أمثلة أخرى:

```
let s = Object.prototype.toString;

alert( s.call(123) ); // [object Number]
alert( s.call(null) ); // [object Null]
alert( s.call(alert) ); // [object Function]
```

ا. Symbol.toStringTag

يمكننا تخصيص سلوك التابع toString للكائنات باستعمال خاصية الكائنات Symbol.toStringTag الفريدة. مثال:

```
let user = {
  [Symbol.toStringTag]: "User" // مستخدم
};

alert( {}.toString.call(user) ); // [object User] // لاحظ
```

لأغلب الكائنات الخاصّة بالبيئات خاصية مثل هذه. إليك بعض الأمثلة من المتصفّحات مثلًا:

```
// تابع toStringTag للكائنات والأصناف الخاصّة بالمتصفّحات:
alert( window[Symbol.toStringTag]); // window
alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest

alert( {}.toString.call(window) ); // [object Window]
alert( {}.toString.call(new XMLHttpRequest()) ); // [object XMLHttpRequest]
```

كما نرى فالناتج هو تمامًا ما يقوله Symbol.toStringTag (لو وُجد) بغلاف [object ...].

في النهاية نجد أن لدينا "نوع من المنشطات" لا تعمل فقط على الأنواع الأولية للبيانات بل وحتى الكائنات المضمنة في اللغة ويمكننا تخصيصها أيضًا.

يمكننا استخدام `{}.toString.call` بدلًا من `instanceof` للكائنات المضمنة في اللغة عندما نريد الحصول على نوع البيانات كسلسلة بدلًا من التحقق منها.

9.6.3 الخلاصة

لنلخّص ما نعرف عن التوابع التي تفحص الأنواع:

	يعمل على	يُعيد
`typeof`	الأنواع الأولية	سلسلة نصية
`{}.toString`	الأنواع الأولية والكائنات المضمّنة والكائنات التي لها `Symbol.toStringTag`	سلسلة نصية
`instanceof`	الكائنات	true/false

كما نرى فعبارة `{}.toString` هي تقنيًا "متقدّمة أكثر" ولكن `typeof`. بل إن التابع `instanceof` يؤدي دور مميّز عندما نتعامل مع تسلسل هرمي للأصناف ونريد التحقق من صنفٍ ما مع مراعاة الوراثة.

9.6.4 تمارين

9.6.5 instnaceof غريب عجيب

الأهمية: ★★★★★

في الشيفرة أسفله، لماذا يُعيد `instanceof` القيمة `true`. يتّضح جليًا بأنّ `B()` لم يُنشِئ `a`.

```
function A() {}
function B() {}

A.prototype = B.prototype = {};
let a = new A();

// هنا
alert( a instanceof B ); // true
```

الحل:

أجل، غريب عجيب حقًا. ولكن كما نعرف فلا يكترث المُعامل `instanceof` بالدالة، بل بكائن prototype لها حيث تُطابقه مع غيره في سلسلة prototype.

وهنا نجد `a.__proto__ == B.prototype`، بذلك يُعيد `instanceof` القيمة `true`.

إذًا فحسب منطق `instanceof`، كائن `prototype` هو الذي يُعرّف النوع وليس الدالة البانية.

9.7 المخاليط Mixins

لا يمكننا في جافاسكربت إلّا أن نرث كائنًا واحدًا فقط. وليس هناك إلّا كائن `[[Prototype]]` واحد لأيّ كائن. ولا يمكن للصنف توسعة أكثر من صنف واحد.

وأحيانًا يكون ذلك مُقيّدًا نوعًا ما. فنقل بأنّ لدينا صنف "كنّاسة شوارع" `StreetSweeper` وصنف "دراجة هوائية" `Bicycle`، وأردنا صنع شيء يجمعهما: "كنّاسة شوارع بدراجة هوائية" `StreetSweepingBicycle`. أو ربّما لدينا صنف المستخدم `User` وصنف "مُطلق الأحداث" `EventEmitter` حيث فيه خاصية توليد الأحداث، ونريد إضافة ميزة `EventEmitter` إلى `User` كي يُطلق المستخدمون الأحداث أيضًا.

هناك مفهوم في اللغة يساعد في حلّ هذه وهو "المخاليط" (Mixins). كما تقول ويكيبيديا، المخلوط هو صنف يحتوي على توابِع يمكن للأصناف الأخرى استعمالها دون أن ترث ذاك الصنف.

أي بعبارة أخرى، يُقدّم المخلوط توابِع فيها وظائف وسلوك معيّن، ولكنّا لا نستعمله لوحده بل نُضيف تلك الوظيفة أو ذاك السلوك إلى الأصناف الأخرى.

9.7.1 مثال عن المخاليط

أبسط طريقة لكتابة تنفيذ مخلوط في جافاسكربت هو صناعة كائن فيه توابِع تُفيدنا في أمور معيّنة كي ندمجها بسهولة داخل كائن prototype لأيّ صنف كان.

لنقل مثلًا لدينا المخلوط `sayHiMixin` ليُضيف بعض الكلام للمستخدم أو الصنف `User`:

```
// مخلوط
let sayHiMixin = {
  sayHi() {
    alert(`Hello ${this.name}`); // مرحبًا يا فلان
  },
  sayBye() {
    alert(`Bye ${this.name}`); // وداعًا يا فلان
  }
};

// الاستعمال:
class User {
  constructor(name) {
    this.name = name;
```

```
    }
  }

  // نسخ التوابع
  Object.assign(User.prototype, sayHiMixin);

  // الآن يمكن أن يرحّب المستخدم بغيره
  new User("Dude").sayHi(); // Hello Dude! مرحبًا يا Dude!
```

ما من وراثة هنا، بل نسخ بسيط للتوابِع. هكذا يمكن أن يرث صنف المستخدم من أيّ صنف آخر وأن يُضيف هذا المخلوط ليدمج فيه توابِع أخرى، هكذا:

```
class User extends Person {
  // ...
}

Object.assign(User.prototype, sayHiMixin);
```

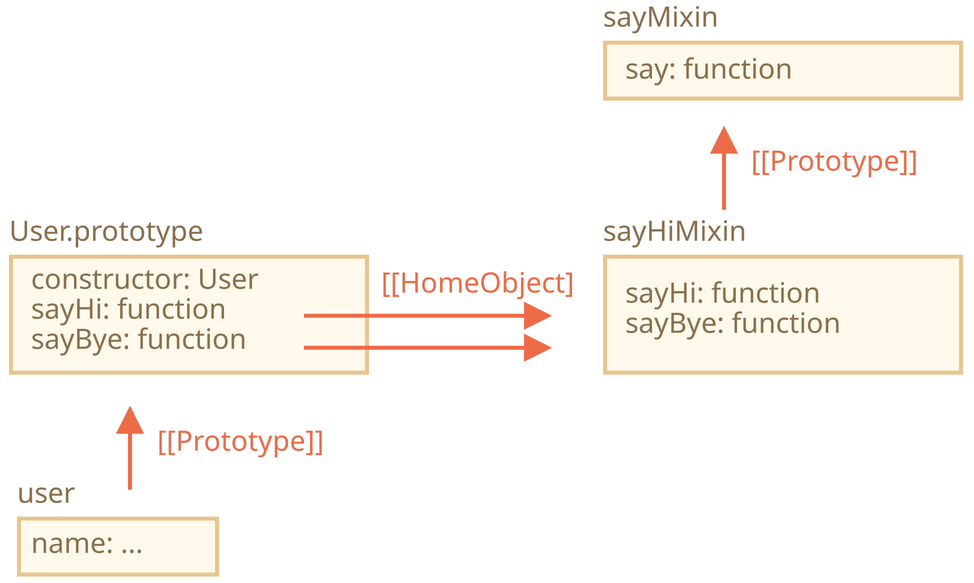
يمكن أن تستفيد المخاليط من الوراثة بينها ذاتها المخاليط. فمثلًا، نرى هنا كيف يرث `sayHiMixin` المخلوطَ `sayMixin`:

```
let sayMixin = {
  say(phrase) {
    alert(phrase);
  }
};

let sayHiMixin = {
  __proto__: sayMixin, // (أو نستعمل Object.create لضبط كائن prototype هنا)

  sayHi() {
    // نستدعي التابع الأب
    super.say(`Hello ${this.name}`); // (*)
  },
  sayBye() {
    super.say(`Bye ${this.name}`); // (*)
  }
```

```
};

class User {
  constructor(name) {
    this.name = name;
  }
}

// نسخ التوابع
Object.assign(User.prototype, sayHiMixin);

// الآن يمكن أن يرحّب المستخدم بالناس
new User("Dude").sayHi(); // مرحبًا يا !Dude
```

لاحظ كيف يبحث استدعاء التابع الأب `super.say()` في `sayHiMixin` (في الأسطر عند `(*)`) - كيف يبحث عن التابع داخل كائن prototype المخلوطَ وليس داخل الصنف.

إليك صورة (طالع الجانب الأيمن فيها):



يعزو ذلك إلى أنّ التابعين `sayHi` و`sayBye` أُنشآ في البداية داخل `sayHiMixin`. فحتّى حين ننسخهما تُشير خاصية `[[HomeObject]]` الداخلية فيهما إلى `sayHiMixin` كما نرى في الصورة.

وطالما يبحث `super` عن التوابع الأب في `[[HomeObject]].[[Prototype]]` يكون البحث داخل `sayHiMixin.[[Prototype]]` وليس داخل `User.[[Prototype]]`.

9.7.2 مخلوط الأحداث EventMixin

الآن حان وقت استعمال المخاليط في الحياة العملية.

يُعدّ توليد الأحداث ميزة مهمّة جدًا للكائنات بشكل عـام، وكائنـات المتصـفّحات بشـكل خـاص. وتُعـدّ هـذه الأحداث الطريقة المُثلى "لنشر المعلومات" لمن يريدها. لذا هيا نصنع مخلوطًا يُتيح لنا إضافة دوال أحــداث إلى أيّ صنف أو كائن.

- سيقدّم المخلوط تابِعًا باسم `.trigger(name, [...data])` "يُولّد حدثًا" متى حدث شـيء مهم. وسيط الاسم `name` هو... اسم الحدث، وبعده تأتي بيانات الحدث إن احتجناها.
- كما والتابع `.on(name, handler)` الذي سيُضيف دالة "معاملة" `handler` لتستمع إلى الأحـداث حسب الاسم الذي مرّرناه. ستُستدعى الدالة متى تفعّل الحدث بالاسم `name` وسيُمرّر لهـا الوسـطاء في استدعاء `.trigger`.
- وأيضًا... التابع `.off(name, handler)` الذي سيُزيل المستمع `handler`.

بعدما نُضيف المخلوط سيقدر كائن المستخدم `user` على توليد حدث ولوج `"login"` مـتى ولج الزائـر إلى الموقع. ويمكن لكائن آخر (لنقل التقويم `calendar`) الاستماع إلى هذا الحدث للقيـام بمهمـة مـا (مثلًا تحميـل تقويم المستخدم الذي ولج).

أو يمكن أن تُولّـد القائمـة `menu` حـدثَ الاختيـار `"select"` مـتى اختـار المسـتخدم عنصـرًا منهـا، ويمكن للكائنات الأخرى إسناد ما تحتاج من معالجات للاستماع إلى ذلك الحدث. وهكذا.

إليك الشيفرة:

```
let eventMixin = {
  /**
   *طريقة الانضمام إلى الحدث
   *  menu.on('select', function(item) { ... }
  */
  on(eventName, handler) {
    if (!this._eventHandlers) this._eventHandlers = {};
    if (!this._eventHandlers[eventName]) {
      this._eventHandlers[eventName] = [];
    }
    this._eventHandlers[eventName].push(handler);
  },
```

```
  /**
   * Cancel the subscription, usage:
   * طريقة إلغاء الانضمام إلى الحدث
   *  menu.off('select', handler)
   */
  off(eventName, handler) {
    let handlers = this._eventHandlers &&
this._eventHandlers[eventName];
    if (!handlers) return;
    for (let i = 0; i < handlers.length; i++) {
      if (handlers[i] === handler) {
        handlers.splice(i--, 1);
      }
    }
  },

  /**
   * توليد حدث من خلال الاسم والبيانات المعطاة
   *  this.trigger('select', data1, data2);
   */
  trigger(eventName, ...args) {
    if (!this._eventHandlers || !this._eventHandlers[eventName]) {
      return; // no handlers for that event name
    }

    // call the handlers
    this._eventHandlers[eventName].forEach(handler =>
handler.apply(this, args));
  }
};
```

شرح الشيفرة:

- `.on(eventName, handler)`: يضبط الدالة `handler` لتُشغّل متى ما حدث الحدث بهذا الاسم. تقنيًا تُخزّن خاصية `_eventHandlerslrm` مصفوفة من دوال المعاملة لكلّ حدث، وتُضيف الدالة إلى القائمة.
- `.off(eventName, handler)`: يحذف الدالّة `handler` من القائمة.

- `.trigger(eventName, ...args)`: يولّد حدث: كلّ المعاملات تُستدعى من `_eventHandlers[eventName]` مع قائمة الوسطاء `...args`.

الاستعمال:

```
// أنشئ صنف
class Menu {
  choose(value) {
    this.trigger("select", value);
  }
}
// أضف مخلوط مع التوابع المرتبط به
Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();
// أضف معالج حدث من أجل أن يستدعى عند الإختيار
menu.on("select", value => alert(`Value selected: ${value}`));

// ينطلق الحدث=> يُشغَل المعالج مباشرةً ويعرُض
// القيمة المختارة:123
menu.choose("123");
```

يمكننا الآن أن نحوّل أيّ شيفرة لتتفاعل متى ما اخترنا شيئًا من القائمة بالاستماع إلى الحدث عبر `menu.on(...)`. كما ويُسهّل المخلوط `eventMixin` من إضافة هذا السلوك (أو ما يشابهه) إلى أيّ صنف نريد دون أن نتدخّل في عمل سلسلة الوراثة.

9.7.3 الخلاصة

مصطلح المخلوط في البرمجة كائنية التوجّه: صنف يحتوي على توابِع نستعملها في أصناف أخرى.

بعض اللغات تتيح لنا الوراثة من أكثر من صنف، أمّا جافاسكربت فلا تتيح ذلك، ولكن يمكننا التحايل على الأمر وذلك بنسخ التوابع إلى النموذج الأولي prototype.

يمكننا استعمال المخاليط بطريقة لتوسيع الصنف وذلك بإضافة سلوكيات متعددة إليه، مثل: معالج الأحداث كما رأينا في المثال السابق.

من الممكن أن تصبح المخاليط نقطة تضارب إذا قاموا بدون قصد بإعادة تعريف توابع الأصناف. لذا عمومًا يجب التفكير مليًا بأسماء المخاليط لتقليل احتمالية حدوث ذلك.

دورة إدارة تطوير المنتجات

تعلم تحويل أفكارك لمنتجات ومشاريع حقيقية بدءًا من دراسة السوق وتحليل المنافسين وحتى إطلاق منتج مميز وناجح

التحق بالدورة الآن

10. التعامل مع الأخطاء

يتضمن هذا الفصل الأقسام التالية:

1. التعامل مع الأخطاء: جرب... التقط try..catch
2. الأخطاء المخصصة وتوسعة صنف Error

10.1 التعامل مع الأخطاء: "جرب... التقط" try..catch

مهما كنّا عباقرة نحن معشر المبرمجين، فلا بدّ أن تكون في السكربتات مشكلة ما. تحدث هذه المشاكل إمّا بسببنا، أو بسبب شيء أدخله المستخدم لم نتوقّعه، أو بسبب ردّ فيه خطأ من الخادم، أو بمليار سبب آخر.

في العادة "يموت" السكربت (أي يتوقّف مباشرة) لو حدث خطأ، ويطبع ذلك في الطرفية. ولكنّ الصياغة `try..catch` تتيح لنا "التقاط" هذه الأخطاء والقيام بما هو مفيد بدل أن يموت السكربت.

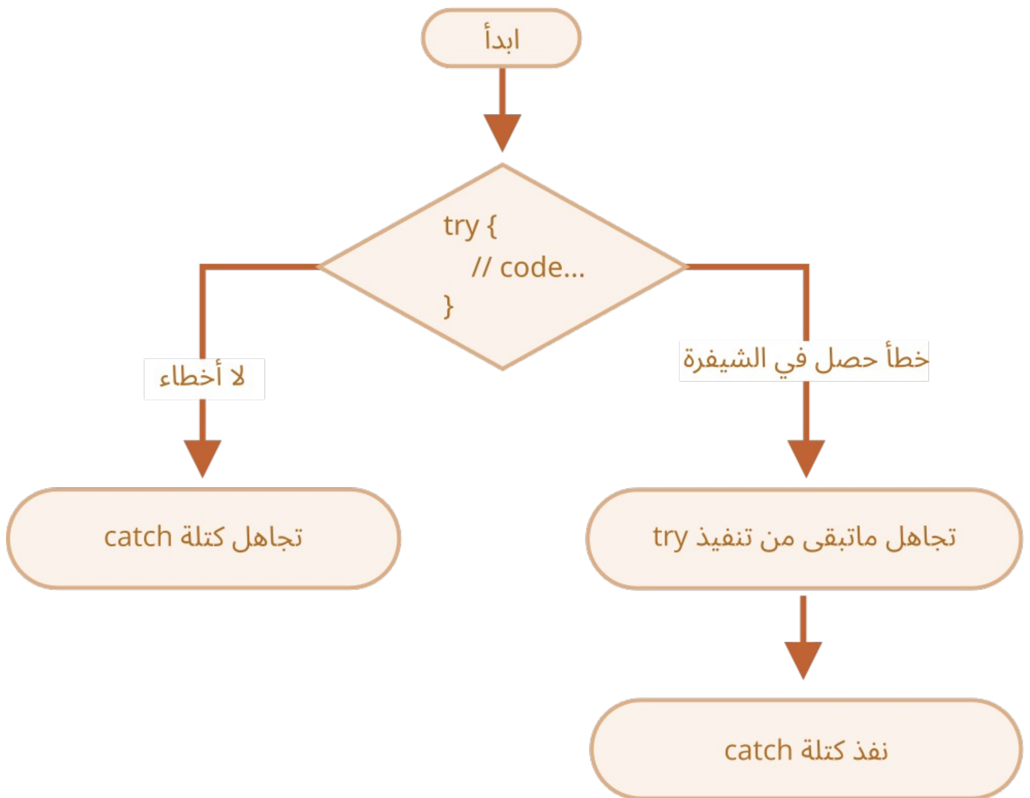
10.1.1 صياغة try..catch

للتعبير `try..catch` كتلتين برمجيتين أساسيتين: التجربة `try` والالتقاط `catch` بعدها:

```
try {

  // الشيفرة...

} catch (err) {

  // التعامل مع الأخطاء

}
```

يعمل التعبير هكذا:

1. أولًا، يجري تنفيذ الشيفرة في `try {...}`.
2. لم نجح التنفيذ دون أيّ أخطاء، تُهمل الكتلة `catch(err)` وتصل عملية التنفيذ إلى نهاية `try` وتواصل عملها بعد تجاهل `catch`.
3. إن حدث أيّ خطأ يتوقّف التنفيذ داخل كتلة `try` وينتقل سير العملية إلى بداية الكتلة `catch(err)`. سيحتوي المتغير `err` (يمكننا استبداله بأيّ اسم آخر) كائن خطأ فيه تفاصيل عمّا حدث.



بهذا لا تقتل الأخطاء داخل كتلة `{…} try` السكربت، فما زال ممكنًا أن نتعامل معها في `catch`.

وقت الأمثلة.

- إليك مثالًا ليس فيه أخطاء: يعرض `alert` السطرين `(1)` و `(2)`:

```
try {

  alert('Start of try runs');  // (1) <-- بداية التشغيلات

  // ...ما من أخطاء هنا

  alert('End of try runs');   // (2) <-- نهاية التشغيلات

} catch(err) {
   // تُهمل catch إذ ليست هناك أخطاء(3)
  alert('Catch is ignored, because there are no errors');

}
```

- مثال فيه خطأ: يعرض السطرين `(1)` و `(3)`:

```
try {

  alert('Start of try runs');  // (1) <-- بداية التشغيلات

  lalala; // error, variable is not defined! خطأ: المتغير غير معرّف!

  alert('End of try (never reached)');  // (2) نهاية كتلة try (لا نصل إليها أبدًا)

} catch(err) {
  alert(`Error has occurred!`); // (3) <-- حدث خطأ!
}
```

انتبه، لا يعمل تعبير `try..catch` إلّا مع الأخطاء أثناء التشغيل، فيجب أن تكون الشيفرة البرمجية صياغتها صحيحة لكي تعمل `try..catch` بعبارة أخرى، يجب أن تكون الشيفرة البرمجية خالية من أخطاء الصياغة. لن تعمل في هذه الحالة لأن هنالك أقواسًا مفتوحة بدون غُلاقاتها.

```
try {
  {{{{{{{{{{{{
} catch(e) {
  alert("The engine can't understand this code, it's invalid");
}
```

أولًا يقرأ محرك جافاسكربت الشيفرة البرمجية، ومن ثمّ يشغّلها. تدعى الأخطاء التي تحدث في مرحلة القراءة أخطاء التحليل (parse-time) ولا يمكن إصلاحها (من داخل الشيفرة البرمجية نفسها). وذلك لأن محرك لغة جافاسكربت لا يستطيع فهم الشيفرة البرمجية من الأساس.

انتبه أيضًا،تعمل الصياغة `try..catch` بشكل متزامن. إذا حدث خطأ ما في شيفرة برمجية مجدولة مثلما في `setTimeout` فلن تستطيع الصياغة `try..catch` أن تلتقطه:

```
try {
  setTimeout(function() {
    noSuchVariable; // السكربت سيموت هنا
  }, 1000);
} catch (e) {
  alert( "won't work" );
}
```

وذلك لأن التابع سيُنفذ لاحقًا، بينما يكون محرك جافاسكربت غادر باني `try..catch`.

للتقاط خطأ بداخل تابِع مُجدوَل يجب أن تكون الصياغة `try..catch` في داخل هذا التابع.

```
setTimeout(function() {
  try {
// صياغة try..catch تُعالج الخطأ
    noSuchVariable; // try..catch handles the error!
  } catch {
    alert( "error is caught here!" );
  }
}, 1000);
```

10.1.2 كائن الخطأ Error

تولّد جافاسكربت -متى حدث الخطأ- كائنًا يحوي تفاصيل الخطأ كاملةً، بعدها تُمرّرها وسيطًا إلى `catch`:

```
try {
  // ...
} catch(err) { // err هذا "كائن الخطأ"، ويمكننا استعمال أيّ اسم نريد بدل
  // ...
}
```

لكائن الخطأ (في حالة الأخطاء المضمّنة في اللغة) خاصيتين اثنتين:

- `name`: اسم الخطأ. فمثلًا لو كان المتغير غير معرّفًا فسيكون الاسم `"ReferenceError"`.
- `message`: الرسالة النصية بتفاصيل الخطأ.

كما هناك خاصيات أخرى غير قياسية في أغلب بيئات جافاسكربت. أكثرها استعمالاً ودعمًا هي:

- `stack`: مَكدس الاستدعاء الحالي: سلسلة نصية فيها معلومات عن سلسلة الاستدعاءات المتداخلة التي تسبّبت بالخطأ. نستعمل الخاصية للتنقيح فقط.

مثال:

```
try {
  lalala; // خطأ المتغيّر غير معرَف
} catch(err) {
  alert(err.name); // ReferenceError(خطأ في الإشارة)
  alert(err.message); // المتغيّر lalala غير معرف
```

```
  alert(err.stack); // ReferenceError:(خطأ في الإشارة) المتغيّر lalala غير معرّف
(call stack...) في

  // يمكننا أيضًا عرض الخطأ بالكامل
  // تحول الخطأ إلى سلسلة نصية هكذا "name: message"

  alert(err); // ReferenceError:(خطأ في الإشارة) المتغيّر lalala غير معرّف
}
```

10.1.3 إسناد "catch" الاختياري

من الممكن أن تحتاج لترقيع الخطأ في المتصفحات القديمة لأن هذه الميّزة جديدة.

لو لم تريد تفاصيل الخطأ فيمكنك إزالتها من `catch`:

```
try {
  // ...
} catch { // <-- بدون المتغير (err)
  // ...
}
```

10.1.4 استعمال try..catch

هيا نرى معًا مثالًا واقعيًا عن `try..catch`:

كما نعلم فجافاسكربت تدعم التابع JSON.parse(str) ليقرأ القيم المرمّزة بِـ JSON. عادةً ما نستعمله لفكّ ترميز البيانات المستلمة من الشبكة، كانت آتية من خادم أو من أيّ مصدر غيره.

نستلم البيانات ونستدعي `JSON.parse` هكذا:

```
let json = '{"name":"Ahmad", "age": 30}'; // بيانات من الخادم

// لاحظ
let user = JSON.parse(json); // نحوّل التمثيل النصي إلى كائن جافاسكربت

// الآن صار user كائنًا فيه خاصيات أتت من السلسلة النصية
alert( user.name ); // Ahmad
alert( user.age );  // 30
```

يمكنك أن ترى تفاصيل أكثر عن JSON في فصل صيغة JSON وتوابعها.

لو كانت سلسلة json معطوبة، فسيُولّد JSON.parse خطأً و"ويموت" السكربت. هل نقبل بالأمر الواقع المرير؟ لا وألف لا!

في هذا الحال لن يعرف الزائر ما يحدث لو حدث للبيانات أمر (ما لم يفتح طرفية المطوّرين). وعادةً ما لا يحب الناس ما "يموت" أمامهم دون أن يُعطيهم رسالة خطأ. فلنستعمل try..catch للتعامل مع هذه المعضلة:

```
let json = "{ bad json }";

try {

  let user = JSON.parse(json); // <-- لو حدث خطأ...
  alert( user.name ); // فلن يعمل هذا

} catch (e) {
  // تنتقل عملية التنفيذ إلى هنا...
  alert( "Our apologies, the data has errors, we'll try to request it one more time." ); // المعذرة، في البيانات أخطاء. سنحاول طلبها مرة ثانية.
  alert( e.name );
  alert( e.message );
}
```

استعملنا هنا الكتلة catch لعرض رسالة لا أكثر، ولكن يمكن أن نستغلّها أفضل من هذا: مثل إرسال طلب إلى الشبكة، أو اقتراح عملية بديلة على الزائر أو إرسال معلومات الخطأ إلى ... تسجيل، وغيرها... هذا كلّه أفضل من الموت والموت فقط.

10.1.5 رمي أخطائنا نحن

ماذا لو كان كائن json صحيح صياغيًا إلّا أنّ الخاصية المطلوبة name ليست فيه؟ هكذا مثلًا:

```
let json = '{ "age": 30 }'; // البيانات ناقصة

try {

  let user = JSON.parse(json); // <-- لا أخطاء
  alert( user.name ); // ما من اسم!

} catch (e) {
```

```
    alert( "doesn't execute" ); // لا يتنفّذ السطر
  }
```

هنا سيعمل التابع `JSON.parse` كما يجب، لكن عدم وجود الخاصية `name` هي المشكلة والخطأ في هذه الحالة. لتوحيد طريقة معالجة الأخطاء، سنستخدم مُعامل `throw`.

ا. مُعامل "الرمي" throw

يُولّد لنا مُعامل الرمي `throw` خطأً. صياغته هي:

```
throw <error object>
```

يمكننا تقنيًا استعمال ما نريد ليكون كائن خطأ. مثل الأنواع الأولية كالأعداد والسلاسل النصية. ولكن من الأفضل استعمال الكائنات ومن الأفضل أكثر أن تملك خاصيتي الاسم `name` والرسالة `message` (لتكون متوافقة إلى حدٍّ ما مع الأخطاء المضمّنة).

في جافاسكربت مختلف البانيات المضمّنة للأخطاء القياسية: الخطأ `Error` والخطأ الصياغي `SyntaxError` والخطأ في الإشارة `ReferenceError` والخطأ في النوع `TypeError` وغيرها. يمكننا استعمال هذه البانيات أيضًا لإنشاء كائنات الخطأ.

صياغتها هي:

```
let error = new Error(message);
// أو
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

في الأخطاء المضمّنة في اللغة (ولا أعني الكائنات أيًا كانت، الأخطاء بعينها) تكون قيمة الخاصية `name` هي ذاتها اسم الباني، وتُؤخذ الرسالة `message` من الوسيط.

مثال:

```
let error = new Error("Things happen o_O"); // غرائب وعجائب ة_ه

alert(error.name); // خطأ
alert(error.message); // غرائب وعجائب ة_ه
```

لنرى نوع الخطأ الذي يُولّده التابع `JSON.parse`:

```
try {
  JSON.parse("{ bad json o_O }"); // جيسون شقي ة_ه
} catch(e) {
  alert(e.name); // SyntaxError خطأ صياغي
  alert(e.message); // Unexpected token o in JSON at position 2
}
```

كما رأينا فهو خطأ صياغي SyntaxError.

وفي حالتنا نحن فعدم وجود الخاصية name هي خطأ، إذ لا بدّ أن يكون للمستخدم اسمًا. هيا نرمِ الخطأ:

```
let json = '{ "age": 30 }'; // البيانات ناقصة

try {

  let user = JSON.parse(json); // <-- لا أخطاء

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name"); // (*)  البيانات
ناقصة: ليس هنالك اسم
  }

  alert( user.name );

} catch(e) {

alert( "JSON Error: " + e.message ); //  خطأ JSON: البيانات ناقصة: ليس هنالك
اسم
}
```

يُولّد مُعامل throw (في السطر (*)) خطأً صياغيًا SyntaxError له الرسالة الممـرّرة message، تمامًـا كما تُولّد جافاسكربت نفسها الخطأ. بهذا يتوقّف التنفيذ داخل try وينتقل سير العملية إلى catch.

الآن صارت الكتلة catch هي المكان الذي نتعامل فيه مع الأخطاء فحسب، أكانت أخطـاء JSON.parse أم أخطاء أخرى.

10.1.6 إعادة الرمي

يمكننا في المثال أعلاه استعمال try..catch للتعامل مع البيانات الخاطئة. ولكن هـل يمكن أن يحـدث خطأ آخر غير متوقّع في كتلة try {...}؟ مثلًا لو كان خطأ المبرمج (لم يعرّف المتغـير) أو شـيئًا آخـر ليس أنّ "البيانات خاطئة"؟

مثال:

```
let json = '{ "age": 30 }'; // البيانات ناقصة

try {
  user = JSON.parse(json); // نسينا "let" قبل user

  // ...
} catch(err) {

  alert("JSON Error: " + err);   // خطأ JSON: خطأ في الإشارة: user غير معرَف
(في الواقع، ليس خطأ JSON)

}
```

كـلّ شـيء ممكن كمـا تعلم! فـالمبرمجون يخطـؤون. حتّى في الأدوات مفتوحـة المصـدر والـتي يسـتعملها ملايين البشر لسنين تمضي، لو اكتُشفت علة فجأةً ستؤدّي إلى اختراقات مأساوية.

في حالتنا هذه على try..catch التقاط أخطـاء "البيانـات الخاطئـة" فقـط. ولكنّ طبيعـة catch هي أن تلتقـط كـلّ الأخطـاء من try. فهنـا مثلًا التقطت خطـأً غـير متوقـع ولكنهـا مـا زالت تصـرّ على رسـالة "JSON Error". هذا ليس صحيحًا ويصعّب من تنقيح الشيفرة كثيرًا.

لحسن الحظ فيمكننا معرفة الخطأ الذي التقطناه، من اسمه name مثلًا:

```
try {
  user = { /*...*/ };
} catch(e) {
  alert(e.name); // "خطأ في الإشارة" محاولة الوصول لمتغيّر غير معرّف
}
```

القاعدة ليست بالمعقّدة:

على catch التعامل مع الأخطاء التي تعرفها فقط، و"إعادة رمي" ما دونها.

يمكننا توضيح فكرة "إعادة الرمي" بهذه الخطوات:

1. تلتقط `catch` كلّ الأخطاء.
2. نحلّل كائن الخطأ `err` في كتلة `catch(err) {...}`.
3. لو لم نعرف طريقة التعامل معه، رميناه `throw err`.

في الشيفرة أسفله استعملنا إعادة الرمي لتتعامل الكتلة `catch` مع أخطاء الصياغة فقط `SyntaxError`:

```
let json = '{ "age": 30 }'; // البيانات ناقصة
try {
  let user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name"); // البيانات ناقصة: لا اسم
  }

  blabla(); // خطأ غير متوقع

  alert( user.name );

} catch(e) {

  if (e.name == "SyntaxError") {
    alert( "JSON Error: " + e.message ); // كذا كذا :JSON خطأ
  } else {
    throw e; // نُعيد رميه (*)
  }
}
```

الخطأ المرمي في السطر `(*)` داخل كتلة `catch` "يسقط خارج أرض" `try..catch` ويمكننا إمّا التقاطه باستعمال تعبير `try..catch` خارجي (لو كتبناه) أو سيقتل الخطأ السكربت.

هكذا لا تتعامل الكتلة `catch` إلا مع ما تعرف مع أخطاء وتتخطّى (إن صحّ القول) الباقي.

يوضّح المثال أسفله كيفية التقاط هذه الأخطاء بمستويات أخرى من `try..catch`:

```
function readData() {
  let json = '{ "age": 30 }';
  try {
    // ...
    blabla(); // خطأ!
  } catch (e) {
    // ...
    if (e.name != 'SyntaxError') {
      throw e; // نُعيد رميه (لا ندري طريقة التعامل معه)
    }
  }
}
try {
  readData();
} catch (e) {
  alert( "External catch got: " + e ); // التقطناه أخيرًا!
}
```

هكذا لا تعرف الدالة readData إلّا طريقة التعامل مع أخطاء الصياغة SyntaxError، بينما تتصرّف تعابير try..catch الخارجية بغيرها من أخطاء (إن عرفتها).

10.1.7 try..catch..finally

لحظة... لم ننتهي بعد. يمكن أيضًا أن يحتوي تعبير try..catch على مُنغلقة أخرى: finally.

لو كتبناها فستعمل في كلّ الحالات الممكنة:

- بعد try لو لم تحدث أخطاء
- وبعد catch لو حدثت أخطاء.

هكذا هي الصياغة "الموسّعة":

```
try {
   ... نحاول تنفيذ الشيفرة ...
} catch(e) {
   ... نتعامل مع الأخطاء ...
} finally {
   ... ننفّذه مهما كان الحال ...
```

```
}
```

جرّب تشغيل هذه الشيفرة:

```
try {
  alert( 'try' );
  if (confirm('Make an error?')) BAD_CODE(); // أتريد ارتكاب خطأ؟
} catch (e) {
  alert( 'catch' ); // أمسكناه
} finally {
  alert( 'finally' ); // بعد ذلك
}
```

يمكن أن تعمل الشيفرة بطريقتين اثنتين:

1. لو كانت الإجابة على "أتريد ارتكاب خطأ؟" "نعم"، فستعمل هكذا `try -> catch -> finally`.
2. لو رفضت ذلك، فستعمل هكذا `try -> finally`.

نستعمل عادةً مُنغلقة `finally` متى ما شرعنا في شيء وأردنا إنهائه مهما كانت نتيجة الشيفرة.

فمثلًا نريد قياس الوقت الذي تأخذه دالة أعداد فيبوناتشي `fib(n)`. الطبيعي أن نبدأ القياس قبل التشغيل ونُنهيه بعده، ولكن ماذا لو حدث خطأ أثناء استدعاء الدالة؟ مثلًا خُذ شيفرة الدالة `fib(n)` أسفله وسترى أنّها تُعيد خطأً لو كانت الأعداد سالبة أو لم تكن أعدادًا صحيحة (not integer).

مُنغلقة `finally` هي المكان الأمثل لإنهاء هذا القياس مهما كانت النتيجة، فتضمن لنا بأنّ الوقت سيُقاس كما ينبغي في الحالتين معًا، نجح تنفيذ أم لم ينجح وأدّى لخطأ:

```
let num = +prompt("Enter a positive integer number?", 35)
let diff, result;

function fib(n) {
  if (n < 0 || Math.trunc(n) != n) {
    throw new Error("Must not be negative, and also an integer.");
  }
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}
let start = Date.now();
try {
```

```
    result = fib(num);
  } catch (e) {
    result = 0;

  } finally {
    diff = Date.now() - start;
  }
  alert(result || "error occurred");
  alert( `execution took ${diff}ms` );
```

يمكنك التأكّد بتشغيل الشيفرة وإدخال 35 في prompt، وستعمل كما يجب، أي تكون finally بعد try. وبعد ذلك جرّب إدخال ‎-1 وسترى خطأً مباشرةً وسيأخذ تنفيذ الشيفرة مدّة 0ms، وكِلا الزمنين المقيسين صحيحين. أي أنّه يمكن للدالة أن بعبارة return أو throw، لا مشكلة إذ ستُنفّذ مُنغلقة finally في أيّ من الحالتين.

ا. المتغيرات في صياغة try..catch..finally محلية

لاحظ أن المتغيرات result و diff في الشيفرة البرمجية أعلاه معرّفة قبل الصياغة try..catch. وبذلك إذا عرفنا let في الصياغة try فستبقى مرئية بداخلها فقط.

ب. بخصوص finally و return

أما بخصوص finally و return فإن مُنغلقة finally تعمل مع أي خروج من صياغة try..catch والتي تتضمن خروج واضح من خلال تعليمة return.

في المثال أدناه هنالك تعليمة return في try في هذه الحالة ستنفذّ finally قبل عودة عنصر التحكم إلى الشيفرة البرمجية الخارجية مباشرة.

```
function func() {
  try {
    return 1;
  } catch (e) {
    /* ... */
  } finally {
    alert( 'finally' );
  }
}
alert( func() ); // أولًا ستعمل تعليمة alert من المنغلقة finally ومن ثمَ هذه
```

ج. بخصوص باني try..finally

إن باني `try..finally` بدون منغلقة `catch` مفيد أيضًا إذ يمكننا استخدامه عندما لا نريد معالجة الأخطاء وإنما نريد التأكد من أن العمليات التي بدأناها انتهت فعلًا.

```
function func() {
// نبدأ بشيء يحتاج لإكمال (مثل عملية القياس)

  try {
    // ...
  } finally {
    // نكمله هنا حتى وإن كلّ السكربت مات
  }
}
```

في الشيفرة البرمجية أعلاه، هنالك خطأ دائمًا يحدث في `try` لأنه لا يوجد `catch`. ومع ذلك ستعمل `finally` قبل الخروج من التابع.

10.1.8 الالتقاط العمومي

إن المعلومات في هذه الفقرة ليست جزءًا من لغة جافاسكربت فهي مخصص لبيئة العمل

لنقل مثلًا أنْ حصل خطأ فادح خارج تعبير `try..catch`، ومات السكربت (مثلًا كان خطأ المبرمج أو أيّ شيء آخر مريب).

أهناك طريقة يمكننا التصرّف فيها في هكذا حالات؟ لربّما أردنا تسجيل الخطأ أو عرض شيء معيّن على المستخدم (عادةً لا يرى المستخدمون رسائل الأخطاء) أو غيرها.

لا نجد في المواصفة أيّ شيء عن هذا، إلّا أنّ بيئات اللغة تقدّم هذه الميزة لأهميتها البالغة. فمثلًا تقدّم Node.js الحدث process.on("uncaughtException"). ولو كنّا نُطوّر للمتصفّح فيمكننا إسناد دالة إلى خاصية window.onerror الفريدة (إذ تعمل متى حدث خطأ لم يُلتقط).

الصياغة هي:

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

- `message`: رسالة الخطأ.

- `url`: عنوان URL للسكربت أين حدث الخطأ.
- `line` و`col`: رقم السطر والعمود أين حدث الخطأ.
- `error`: كائن الخطأ.

مثال:

```
<script>
  window.onerror = function(message, url, line, col, error) {
    alert(`${message}\n At ${line}:${col} of ${url}`);
  };
  function readData() {
    badFunc(); // Whoops, something went wrong!
  }
  readData();
</script>
```

عادةً، لا يكون الهدف من المعالج العمومي `window.onerror` استعادة تنفيذ السكربت (إذ هو أمر مستحيل لو كانت أخطاء من المبرمج) بل يكون لإرسال هذه الرسائل إلى المبرمج.

كما أنّ هناك خدمات على الوِب تقدّم مزايا لتسجيل الأخطاء في مثل هذه الحالات، أمثال errorception.com وmuscula.com و sentry.io.

وهكذا تعمل:

1. نُسجّل في الخدمة ونأخذ منهم شيفرة جافاسكربت صغيرة (أو عنوانًا للسكربت) لنضعها في صفحاتنا.
2. يضبط هذا السكربت دالة `window.onerror` مخصّصة.
3. ومتى حدث خطأ أرسلَ طلب شبكة بالخطأ إلى الخدمة.
4. ويمكننا متى أردنا الولوج إلى واجهة الوِب للخدمة ومطالعة تلك الأخطاء.

10.1.9 الخلاصة

يتيح لنا تعبير `try..catch` التعامل مع الأخطاء أثناء تنفيذ الشيفرات. كما يدلّ اسمها فهي تسمح "بتجربة" تنفيذ الشيفرة و"والتقاط" الأخطاء التي قد تحدث فيها.

صياغتها هي:

```
try {
  // شغّل الشيفرة
} catch(err) {
  // انتقل إلى هنا لو حدث خطأ
  // هو كائن الخطأ err
} finally {
  // مهما كان الذي حدث، نفّذ هذا
}
```

يمكننا إزالة قسم catch أو قسم finally، واستعمال الصيغ الأقصر try..catch و try..finally.

لكائنات الأخطاء الخاصيات الآتية:

- message: رسالة الخطأ التي نفهمها نحن البشر.
- name: السلسلة النصية التي فيها اسم الخطأ (اسم باني الخطأ).
- stack (ليست قياسية ولكنّها مدعومة دعمًا ممتازًا): المَكدس في لحظة إنشاء الخطأ.

لو لم تُرد كائن الخطأ فيمكنك إزالتها باستعمال catch { بدل catch(err) {.

يمكننا أيضًا توليد الأخطاء التي نريد باستعمال مُعامل throw. تقنيًا يمكن أن يكون وسيط throw ما تريد ولكن من الأفضل لو كان كائن خطأ يرث صنف الأخطاء Error المضمّن في اللغة. سنعرف المزيد عن توسعة الأخطاء في القسم التالي.

كما يُعدّ نمط "إعادة الرمي" البرمجي نمطًا مهمًا عند التعامل مع الأخطاء، فلا تتوقّع كتلة catch إلّا أنواع الأخطاء التي تفهمها وتعرف طريقة التعامل معها، والباقي عليها إعادة رميه لو لم تفهمه.

حتّى لو لم نستعمل تعبير try..catch فأغلب البيئات تتيح لنا إعداد ... أخطاء "عمومي" لالتقاط الأخطاء التي "تسقط أرضًا". هذا ... في المتصفّحات اسمه window.onerror.

ا. تمارين

10.1.10 أخيرًا أم الشيفرة؟

الأهمية: ★★★★★

وازِن بين الشيفرتين هاتين.

1. تستعمل الأولى finally لتنفيذ الشيفرة بعد try..catch:

```
try {
  // عمل عمل عمل
} catch (e) {
  // نتعامل مع الأخطاء
} finally {
  // ننظّف مكان العمل هنا
}
```

2. الثانية تضع عملية التنظيف بعد `try..catch`:

```
try {
  // عمل عمل عمل
} catch (e) {
  // نتعامل مع الأخطاء
}
// ننظّف مكان العمل هنا
```

سنحتاج بكل بتأكيد للتنظيف بعد العمل. بغض النظر إن وجدَنا خطأً ما أم لا.

هل هنالك ميزة إضافية لاستخدام `finally`؟ أم أن الشيفرتين السابقتين متساويتين؟ إن كان هنالك ميزّة اكتب مثلًا يوضحها.

الحل:

عندما ننظر إلى الشيفرة الموجودة بداخل التابع يصبح الفرق جليًا. يختلف السلوك إذا كان هناك "قفزة" من "try..catch". على سبيل المثال، عندما يكون هناك تعليمة `return` بداخل صياغة `try..catch`. تعمل منغلقة `finally` في حالة وجود أي خروج من صياغة `try..catch`، حتى عبر تعليمة `return`: مباشرة بعد الانتهاء من`try..catch` ، ولكن قبل أن تحصل شيفرة الاستدعاء على التحكم.

```
function f() {
  try {
    alert('start');
    return "result";
  } catch (e) {
    /// ...
  } finally {
    alert('cleanup!');
  }
```

```
}
f(); // cleanup!
```

... أو عندما يكون هناك throw، مثل هنا:

```
function f() {
  try {
    alert('start');
    throw new Error("an error");
  } catch (e) {
    // ...
    if("can't handle the error") {
      throw e;
    }

  } finally {
    alert('cleanup!')
  }
}

f(); // cleanup!
```

تضمنُ finally عملية التنظيف هنا. إذا وضعنا الشيفرة البرمجية في نهاية f ، فلن تشغّل في هذا السيناريو.

10.2 الأخطاء المخصصة وتوسعة صنف Error

متى نكون نطوّر البرامج نحتاج إلى أصناف أخطاء خاصة بنا لتوضّح تمامًا ما قد يحدث خطأً في المهام الـتي نقوم بها. فمثلًا لأخطاء الشبكة نستعمل `HttpError`، ولعمليات قواعد البيانـات `DbError` ولعمليـات البحث `NotFoundError` وهكذا.

وعلى هذه الأخطاء أن تدعم الخاصيات الأساسية مثل الرسالة `message` والاسـم `name` والمَكـدس `stack` (يفضّل ذلك)، ولكن يمكن أن تحتـوي على خصـائص أخـرى خاصـة بهـا مثـل خاصيـة `statusCode` لكائنـات `HttpError` وتحمل قيمة من قيم رموز الحالة 404 أو 403 أو 500.

تتيح لنا جافاسكربت استعمال `throw` بتمرير أيّ وسـيط، لـذا فأصـناف الخطـأ الخاصـة بنـا يمكن ألّا تـرث (تقنيًا) من كائن الخطأ `Error`، ولكن لو ورثنا منهـا فيمكنن للجميـع اسـتعمال `obj instanceof Error` لاحقًا لتتعرّف على كائنات الخطأ. بذلك يكون أفضل لو ورثناها.

وكلّما كبر التطبيق شكّلت الأخطاء التي خصّصناها شـجرة، فمثلًا سـيرث الصـنف `HttpTimeoutError` الصنفَ `HttpError`، وهكذا دواليك.

10.2.1 توسعة Error

لنأخـذ مثـالًا دالـةَ `readUser(json)` تقـرأ كـائن JSON في بيانـات المسـتخدم وهـذا مثـال عن كـائن `json` صالح:

```
let json = `{ "name": "Ahmad", "age": 30 }`;
```

سنستعمل في الشيفرة التابع `JSON.parse`، وإن استلم كائن `json` معطوب رمى خطأ `SyntaxError`. ولكن، حتّى لو كان الكائن صحيحًا صياغيًا، فلا يعني هذا أنّ المستخدم صالحًا أيضًا، أم لا؟ لربّمـا لا يحتـوي بعض البيانات مثل خاصيتي الاسم `json` والعمر `name` الضروريتين للمستخدمين.

بهذا لن تقرأ الدالة `readUser(json)` كائن JSON فحسب، بل ستفحص ("تتحقّـق من") البيانـات، فلـو لم تكن الحقول المطلوبة موجودة، أو كان تنسيق الكائن مغلوطًا، فهنا نكون أمام خطأ... وهـذا الخطـأ ليس خطـأً صـياغيًا `SyntaxError` إذ أنّ البيانـات صـحيحة صـياغيًا، بـل نـوع آخـر من الأخطـاء. سنسـمّي هـذا النـوع `ValidationError` ونصـنع صـنف لـه. على هـذا النـوع من الأخطـاء احتـواء مـا يلـزم من معلومـات تخصّ الحقل المخالف.

يفترض علينا وراثـة الصـنف المضـمّن في اللغـة `Error` لصـنفنا `ValidationError`. إليـك شـيء عن شيفرة الصنف المضمّن لنعرف ما نحاول توسعته:

```
// شيفرة مبسّطة لصنف الخطأ Error المضمّن في لغة جافاسكربت نفسها
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (تختلف الأسماء باختلاف أصناف الأخطاء المضمّنة)
    this.stack = <call stack>; // ليست قياسية، إلّا أنّ أغلب البيئات تدعمها
  }
}
```

الآن صار وقت أن يرث الصنف `ValidationError` منها:

```
class ValidationError extends Error {
  constructor(message) {
    super(message); // (1)
    this.name = "ValidationError"; // (2) خطأ في التحقّق
  }
}

function test() {
  throw new ValidationError("Whoops!"); // آخ
}

try {
  test();
} catch(err) {
  alert(err.message); // آخ!
  alert(err.name); // خطأ في التحقّق
  alert(err.stack); // قائمة من الاستدعاءات المتداخلة في كلّ منها رقم السطر
}
```

لاحظ كيف أنّنا في السطر `(1)` استدعينا الباني الأب، إذ تطلب منا لغة جافاسكربت استدعاء `super` في البانيات الابنة، أي أنّه أمر إلزامي. يضبط الباني الأب خاصية الرسالة `message`. كما يضبط أيضًا خاصية الاسم `name` لتكون `"Error"`، ولذلك نُعدّلها إلى القيمة الصحيحة في السطر `(2)`.

فلنحاول الآن استعماله في `readUser(json)`:

```
class ValidationError extends Error {
  constructor(message) {
```

```
    super(message);
    this.name = "ValidationError"; // خطأ في التحقّق
  }
}

// الاستعمال
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new ValidationError("No field: age"); // حقل غير موجود: العمر
  }
  if (!user.name) {
    throw new ValidationError("No field: name"); // حقل غير موجود: الاسم
  }

  return user;
}

// هنا نجرّب باستعمال try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // البيانات غير صالحة: حقل غير موجود: الاسم
  } else if (err instanceof SyntaxError) { // (*)
    alert("JSON Syntax Error: " + err.message); // خطأ صياغي لكائن JSON
  } else {
    throw err; // خطأ لا نعرفه، علينا إعادة رميه (**)
  }
}
```

تتعامل كتلة `try..catch` في الشيفرة أعلاه النوعين من الأخطاء: `ValidationError` والخطأ المضمّن `SyntaxError` الذي يرميه التابع `JSON.parse`.

لاحِظ أيضًا كيف استعملنا `instanceof` لفحص نوع الخطأ في السطر `(*)`.

يمكننا أيضًا مطالعة err.name هكذا:

```
// ...
// (err instanceof SyntaxError) بدل
} else if (err.name == "SyntaxError") { // (*)
// ...
```

ولكنّ استعمال instanceof أفضل بكثير إذ يحدث ونوسّع مستقبلًا الصنف ValidationError بأصناف فرعية منه مثل PropertyRequiredError، والفحص عبر instanceof سيظلّ يعمل للأصناف الموروثة منه.

كما من المهمّ أن تُعيد كتلة catch رمي الأخطاء التي لا تفهمها، كما في السطر (**). ليس على هذه الكتلة إلّا التعامل مع أخطاء التحقّق والصياغة، أمّا باقي الأخطاء (والتي قد تحدث بسبب الأخطاء المطبعية في الشيفرة أو غيرها من أمور غريبة عجيبة) فيجب أن تسقط أرضًا.

10.2.2 تعميق الوراثة

صنف الخطأ ValidationError عامٌ جدًا جدًا، إذ يمكن أن تحصل أمور كثيرة خطأً في خطأ. لربّما كانت الخاصية غير موجودة أو كان نسقها خطأ (مثل تقديم سلسلة نصية قيمةً للعمر age). لنصنع الصنف PropertyRequiredError ونستعمله فقط للخاصيات غير الموجودة، وسيحتوي على أيّة معلومات إضافية عن الخاصية الناقصة.

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError"; // خطأ في التحقق
  }
}

// هذا
class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property); // خاصية غير موجودة
    this.name = "PropertyRequiredError"; // خطأ إذ الخاصية مطلوبة
    this.property = property;
  }
}
```

```
// الاستعمال
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new PropertyRequiredError("age"); // العمر
  }
  if (!user.name) {
    throw new PropertyRequiredError("name"); // الاسم
  }

  return user;
}

// try..catch هنا نجرّب باستعمال

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // البيانات غير صالحة: خاصية غير موجودة: الاسم
    alert(err.name); // PropertyRequiredError
    alert(err.property); // name
  } else if (err instanceof SyntaxError) {
    alert("JSON Syntax Error: " + err.message); // JSON خطأ صياغي لكائن
  } else {
    throw err; // خطأ لا نعرفه، علينا إعادة رميه
  }
}
```

يسهُل علينا استعمال الصنف الجديد `PropertyRequiredError`، فكلّ ما علينا تمريره هو اسم الخاصية: `new PropertyRequiredError(property)`، وسيُولّد الباني الرسالةَ `message` التي نفهمها نحن البشر.

لاحظ كيف أنّنا أسندنا الخاصية this.name في باني PropertyRequiredError يدويًا، مرّة ثانية. قد ترى هذا الأمر مُتعبًا حيث ستُسند قيمة <class name> = this.name في كلّ صنف خطأ تصنعه ولكن يمكن تجنّب هذا العناء وإسناد قيمة مناسبة this.name = this.constructor.name لصنف "الخطأ الأساس" وبعدها نرث من هذا الصنف كلّ أصناف الأخطاء المخصّصة.

لنسمّه مثلًا MyError. إليك شيفرة MyError وغيرها من أصناف أخطاء مخصّصة، ولكن مبسّطة:

```
class MyError extends Error {
  constructor(message) {
    super(message);
    this.name = this.constructor.name; // هنا
  }
}

class ValidationError extends MyError { }

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property); // خاصية غير موجودة
    this.property = property;
  }
}

// الاسم صحيح
alert( new PropertyRequiredError("field").name ); //
PropertyRequiredError
```

الآن صارت شيفرات الأخطاء المخصّصة أقصر بكثير (خاصّة ValidationError) إذ حذفنا السطر "... = this.name" في الباني.

10.2.3 تغليف الاستثناءات

هدف الدالة readUser في الشيفرة أعلاه هو "قراءة بيانات المستخدم"، ويمكن أن تحدث مختلف الأخطاء أثناء تنفيذ ذلك. حاليًا نرى SyntaxError وValidationError فقط، ولكن في المستقبل العاجل ستصير الدالة readUser أكبر وأكبر وقد تُولّد لنا أنواع أخرى من الأخطاء.

ومَن يتعامل مع هذه الأخطاء؟ الشيفرة التي تستدعي readUser! حاليًا لا تستعمل إلا بضعة تعابير شرطية if في كُتل catch (تفحص الصنف وتتعامل مع الأخطاء وتُعيد رمي ما لا تفهم)، وسيكون المخطط هكذا:

```
try {
  ...
  readUser()  // الخطأ الأساسي هنا
  ...
} catch (err) {
  if (err instanceof ValidationError) {
    // ValidationError معالجة أخطاء
  } else if (err instanceof SyntaxError) {
    // SyntaxError معالجة الأخطاء الصياغية
  } else {
    throw err; // خطأ مجهول فلنُعد رميه من جديد
  }
}
```

نرى في الشيفرة البرمجية أعلاه نوعين من الأخطاء ولكن ممكن أن يكون أكثر من ذلك.

ولكن متى ولّدت الدالة readUser أنواع أخرى من الأخطاء، فعلينا طرح السؤال: هل علينا حقًا فحص كلّ نوع من أنواع الأخطاء واحدًا واحدًا في كلّ شيفرة تستدعي readUser؟

عادةً ما يكون الجواب هو "لا"، فالشيفرة الخارجية تفضّل أن تكون "على مستوًى أعلى من ذلك المستوى"، أي أن تستلم ما يشبه "خطأ في قراءة البيانات"، أمّا عن سبب حدوثه فلا علاقة لها به (طالما رسالة الخطأ تصف نفسها). أو ربّما (وهذا أفضل) تكون هناك طريقة لتحصل فيها على بيانات الخطأ، لو أرادت الشيفرة ذلك.

تدعى الطريقة التي وصفناها بتغليف الاستثناءات.

1. لنصنع الآن صنف "خطأ في القراءة" ReadError لنمثّل هذه الأخطاء.
2. متى حدثت أخطاء داخل الدالة readUser، سنلتقطها فيها ونُولّد خطأ ReadError، بدلًا من ValidationError و SyntaxError.
3. الكائن ReadError سيُبقي أيضًا إشارة إلى الخطأ الأصلي في خاصية السبب cause.

لذا و هكذا لن يكون للشيفرة الخارجية (التي ستستدعي readUser) إلّا فحص ReadError. وليس كلّ نوع من أخطاء قراءة البيانات. وإن كان يحتاج لمزيد من التفاصيل عن الخطأ يمكنه العودة إلى الخاصية cause والتحقق منها.

إليك الشيفرة التي نُعرّف فيها الخطأ `ReadError` ونمثّل طريقة استعماله في الدالة `readUser` وفي `try..catch`:

```
class ReadError extends Error {
  constructor(message, cause) {
    super(message);
    this.cause = cause;
    this.name = 'ReadError'; // خطأ في القراءة
  }
}

class ValidationError extends Error { /*...*/ }
class PropertyRequiredError extends ValidationError { /* ... */ }

function validateUser(user) {
  if (!user.age) {
    throw new PropertyRequiredError("age"); // العمر
  }

  if (!user.name) {
    throw new PropertyRequiredError("name"); // الاسم
  }
}

function readUser(json) {
  let user;

  try {
    user = JSON.parse(json);
  } catch (err) {
  // هنا
    if (err instanceof SyntaxError) {
      throw new ReadError("Syntax Error", err); // خطأ صياغي
    } else {
      throw err;
    }
  }
```

```
    try {
      validateUser(user);
    } catch (err) {
    // وهنا
      if (err instanceof ValidationError) {
        throw new ReadError("Validation Error", err); // خطأ في التحقّق
      } else {
        throw err;
      }
    }

  }

  try {
    readUser('{bad json}');
  } catch (e) {
    if (e instanceof ReadError) {
    // هنا    alert(e);
      // Original error: SyntaxError: Unexpected token b in JSON at
  position 1
      alert("Original error: " + e.cause); // الخطأ الأصل
    } else {
      throw e;
    }
  }
```

في الشيفرة أعلاه تعمل الدالة readUser تمامًا كم وصفها، تلتقط أخطاء الصياغة والتحقّق وترمي أخطاء قراءة ReadError بدلها (كما وتُعيد رمي الأخطاء المجهولة أيضًا).

هكذا ليس على الشيفرة الخارجية إلّا فحص ReadError instanceof فقط، لا داعٍ للتحقّق من كلّ خطأ يمكن أن يحصل في هذه المجرّة.

يُسمّى هذا النهج "بتغليف الاستثناءات" حيث نستلم نحن "الاستثناءات في المستوى المنخفض من البرمجة" (low level) و"نُغلّفها" داخل خطأ ReadError يكون أكثر بساطة وأسهل استعمالاً للشيفرات التي تنادي على الدوال. هذا النهج مستعمل بكثرة في البرمجة كائنية التوجّه.

10.2.4 الخلاصة

- يمكننا وراثة صنف الخطأ `Error` وغيرها من أخطاء مضمّنة كما الوراثة العادية. المهم أن ننتبه من خاصية الاسم `name` ولا ننسى استدعاء `super`.
- يمكننا استعمال `instanceof` لفحص ما نريد من أخطاء بدقّة، كما ويعمل المُعامل مع الوراثة. أحيانًا نستلم كائن خطأ من مكتبة طرف ثالث وما من طريقة سهلة لنعرف اسم صنفها. هنا يمكن استعمال خاصية الاسم `name` لإجراء هذا الفحص.
- أسلوب تغليف الاستثناءات هو أسلوب منتشر الاستعمال، فيه تتعامل الدالة مع الاستثناءات في المستوى المنخفض من البرمجة، وتصنع أخطاء مستواها عالٍ بدل تلك المنخفضة. وأحيانًا تصير الاستثناءات المنخفضة خصائص لكائن الخطأ (تمامًا مثل `err.cause` في الأمثلة أعلاه)، ولكنّ هذا ليس إلزاميًا أبدًا.

10.2.5 تمارين

ا. الوراثة من SyntaxError

الأهمية: ★★★★★

اصنع الصنف `FormatError` ليرث من الصنف المضمّن `SyntaxError`. يجب أن يدعم الصنف خصائص الاسم `name` والرسالة `message` والمَكدس `stack`.

طريقة الاستعمال:

```
let err = new FormatError("formatting error"); // خطأ في التنسيق

alert( err.message ); // خطأ في التنسيق
alert( err.name ); // FormatError
alert( err.stack ); // المَكدس

alert( err instanceof FormatError ); // true
alert( err instanceof SyntaxError ); // true (إذ يرث الصنف الصنفَ SyntaxError)
```

الحل:

```
class FormatError extends SyntaxError {
  constructor(message) {
    super(message);
    this.name = this.constructor.name;
  }
}

let err = new FormatError("formatting error");

alert( err.message ); // خطأ في التنسيق
alert( err.name ); // خطأ في التنسيق
alert( err.stack ); // stack(المَكدس)

alert( err instanceof SyntaxError ); // true
```

أكبر سوق عربي لبيع وشراء الخدمات المصغرة
اعرض خدماتك أو احصل على ما تريد بأسعار تبدأ من 5$ فقط

تصفح الخدمات

11. الوعود واللاتزامن/الانتظار

يتضمن هذا الفصل الأقسام التالية:

1. مقدمة إلى ردود النداء callbacks
2. الوعود Promise
3. سَلسلة الوعود Promises chaining
4. التعامل مع الأخطاء: then وcatch
5. واجهة الوعود البرمجية
6. الدوال الواعدة: تحويل الدوال إلى وعود
7. المهام السريعة مقابل الوعد لتنفيذ المهام لاحقًا
8. اللاتزامن والانتظار async/await

11.1 مقدمة إلى ردود النداء callbacks في جافاسكربت

> لتوضيح طريقة استخدام ردود النداء والوعود والمفاهيم المجردة سنستخدم بعض توابع المتصفح، تحديدًا سكربتات التحميل وأدوات التلاعب بالمستندات البسيطة، فإن لم تكُ على دراية بهذه الطرق، وكان استخدامها في الأمثلة مربكًا نوعًا ما، أو حتى إن رغبتَ فقط في فهمها فهمًا أفضل، فيمكنك الرجوع إلى الجزء التالي من الكتاب. وبالرغم من ذلك سنحاول توضيح الأمور بكل الأحوال ولن يكون هنالك شيء معقد في المتصفح.

أغلب الإجراءات في جافاسكربت هي إجراءات غير متزامنة، أي أنّنا نشغّلها الآن ولكنّها تنتهي في وقت لاحق، مثال على ذلك هو حين نُجدول تلك الإجراءات باستعمال `setTimeout`.

يُعدّ تحميل السكربتات والوحدات (سنشرحها في الفصول القادمة) أمثلة فعلية عن الإجراءات غير المتزامنة. لاحظ مثلًا الدالة `loadScript(src)` أسفله، إذ تُحمّل سكربت من العنوان الممرّر `src`:

```
function loadScript(src) {
  // أنشئ وسم <script> وأضفه للصفحة
  // فسيؤدي ذلك إلى بدء تحميل السكربت ذي الخاصية src ثم تنفيذه عند الاكتمال
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

تُضيف الدالة إلى المستند الوسم `<script src="…">` الجديد الذي وُلّد ديناميكيًا. حين يعمل المتصفّح ينفّذ الدالة. يمكننا استعمال هذه الدالة هكذا:

```
// حمّل ونفّذ السكربت في هذا المكان
loadScript('/my/script.js');
```

يُنفّذ هذا السكربت "بلا تزامن" إذ تحميله يبدأ الآن أمّا تشغيله فيكون لاحقًا متى انتهى الدالة. ولو كانت هناك شيفرة أسفل الدالة `loadScript(…)` فلن ننتظر انتهاء تحميل السكربت.

```
loadScript('/my/script.js');
// الشيفرة أسفل loadScript
// لا تنتظر انتهاء تحميل السكربت
// ...
```

فنقل بأنّنا نريد استعمال السكربت متى انتهى تحميله (فهو مثلًا يُصرّح عن دوال جديدة ونريد تشغيلها). ولكن لو حدث ذلك مباشرةً بعد استدعاء `loadScript(…)`، فلن تعمل الشيفرة:

```
loadScript('/my/script.js'); // newFunction في السكربت الدالة
newFunction(); // ما من دالة بهذا الاسم!
```

الطبيعي هو أنّ ليس على المتصفّح انتظار مدّة من الزمن حتّى ينتهي تحميل السكربت. كما نـرى فحاليًـا لا تقدّم لنا الدالة `loadScript` أيّ طريقة لنعرف وقت اكتمال التحميل، بل يُحمّل السكربت ويُشغّل متى اكتمل، وفقط. ولكن ما نريده هو معرفة وقت ذلك كي نستعمل الدوال والمتغيرات الجديدة فيه.

لنُضيـف دالـة ردّ نـداء `callback` لتكـون الوسـيط الثـاني لدالـة `loadScript`، كي تُنفّـذ مـتى انتهى تحميل السكربت:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}
```

الآن متى أردنا استدعاء الدوال الجديدة في السكربت، نكتبها في دالة ردّ النداء تلك:

```
loadScript('/my/script.js', function() { // يعمل ردّ النداء بعد تحميل السكربت
  newFunction(); // الآن تعمل
  ...
});
```

هذه الفكرة تمامًا: يُعدّ الوسيط الثاني دالةً (عادةً ما تكون مجهولة) تعمل متى اكتمـل الإجـراء. وإليـك مثـالًا حقيقيًا قابل للتشغيل:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}
loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/
lodash.js', script => {
  alert(`Cool, the script ${script.src} is loaded`);
  alert( _ ); // التابع معرف في السكربت المُحمّل
});
```

يُسمّى هذا الأسلوب في البرمجة غير المتزامنة "الأسلوب المبني على ردود النـداء" (callback-based)، إذ على الدوال التي تنفّذ أمور غير متزامنة تقديمَ وسيط ردّ نداء `callback` نضع فيـه الدالـة الـتي ستُشـغّل مـتى اكتملت تلك الأمور.

ولقد فعلناها في `loadScript` ولكن بكلّ الأحوال هذا نهج عام.

11.1.1 رد النداء داخل رد النداء

وكيف نُحمّل سكربتين واحدًا تلو الآخر: يعمل الأول وبعدها حين ينتهي يعمل الثاني؟

الحلّ الطبيعي الذي سيفكّر به الجميع هو وضع استدعاء `loadScript` الثاني داخل ردّ النداء، هكذا:

```
loadScript('/my/script.js', function(script) {

  // جميل، اكتمل تحميل كذا، هيّا نُحمّل الآخر
  alert(`Cool, the ${script.src} is loaded, let's load one more`);

  loadScript('/my/script2.js', function(script) {
    // جميل، اكتمل تحميل السكربت الثاني أيضًا
    alert(`Cool, the second script is loaded`);
  });
});
```

وبعدما تكتمل الدالة الخارجية `loadScript`، يُشغّل ردّ النداء الدالة الداخليـة. ولكن مـاذا لـو أردنـا سـكربتًا آخر، أيضًا...؟

```
loadScript('/my/script.js', function(script) {
  loadScript('/my/script2.js', function(script) {
    loadScript('/my/script3.js', function(script) {
      // نواصل متى اكتملت كلّ السكربتات...
    });
  })
});
```

هكذا نضع كلّ إجراء جديد داخل ردّ نداء. لو كانت الإجراءات قليلة فليست مشكلة، ولكن لو زادت فستصـير كارثية. سنرى ذلك قريبًا.

11.1.2 التعامل مع الأخطاء

لم نضع الأخطاء في الحسبان في هذه الأمثلة. ماذا لو فشل تحميل السكربت؟ يفترض أن تتفاعل دالة ردّ النداء بناءً على الخطأ.

إليك نسخة محسّنة من الدالة `loadScript` تتعقّب أخطاء التحميل:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  // هنا
  script.onload = () => callback(null, script);
  // خطأ في تحميل السكربت كذا
  script.onerror = () => callback(new Error(`Script load error for $
{src}`));
  document.head.append(script);
}
```

هنا نستدعي `callback(null, script)` لو اكتمل التحميل، ونستدعي `callback(error)` لو لم يكتمل. طريقة الاستعمال:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // نتعامل مع الخطأ
  } else {
    // تحمّل السكربت بنجاح
  }
});
```

نُعيد، ما استعملناه هنا للدالة `loadScript` هي طريقة مشهورة جدًا، تُدعى بأسلوب "ردّ نداء الأخطاء أولًا". إليك ما اصطُلح عليه:

1. يُحجز الوسيط الأوّل من دالة `callback` للخطأ إن حدث، ويُستدعى `callback(err)`.
2. يكون الوسيط الثاني (وغيرها إن دعت الحاجة) للنتيجة الصحيحة.

هكذا نستعمل الدالة `callback` فقط للأمرين: الإبلاغ عن الأخطاء وتمرير النتيجة.

11.1.3 هرم العذابات Pyramid of Doom

يمكن أن نرى أنّ البرمجة بنمط اللاتزامن مفيد جدًا، وهذا جليّ من النظرة الأولى. فحين نكون أمـام اسـتدعاء أو استدعاءين فقط، فأمورنا ممتازة.

ولكن لو تتابعت الإجراءات غير المتزامنة واحدة تلو الأخرى، سنرى هذا:

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // (*) نُواصل بعد اكتمال تحميل كل السكربتات...
          }
        });
      }
    })
  }
});
```

إليك ما في الشيفرة أعلاه:

1. نُحمّل `1.js` ونرى لو لم تحدث أخطاء.
2. نُحمّل `2.js` ونرى لو لم تحدث أخطاء.
3. نُحمّل `3.js`، ولو لم تحدث أخطاء نفّذنا شيئًا `(*)`.

فكلّما تداخل الاستدعاءات أكثر أصبحت الشيفرة متشعّبة جدًا وأصعب في الإدارة كثيرًا، هذا خصوصًا لو كانت هناك شيفرة فعلية لا `...` (مثل الحلقات والعبارات الشرطية وغيرها).

يُسمّون هذا أحيانًا "بجحيم ردود النداء" (callback hell) أو "هرم العذابات".

```
loadScript('1.js', function(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...
          }
        });
      }
    })
  }
});
```

بزيادة الإجراءات غير المتزامنة واحدًا بعد آخر، يتشعّب "هرم" الاستدعاءات المتداخلة إلى اليمين أكثر فأكثر، ولن يمضي من الوقت الكثير حتى دخلتْ في دوّامة حلزونية محال تنظيمها.

بهذا تُعدّ طريقة البرمجة هذه سيّئة. يمكننا في محاولة يائسة لتقليل وقع المشكلة تحويل كلّ إجراء إلى دالة منفردة، هكذا:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
```

```
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...نواصل بعد اكتمال تحميل كل السكربتات (*)
  }
};
```

رأيت الفكرة؟ مبدأ الشيفرة واحد وليس هناك تداخلات متشعّبة إذ حوّلنا كلّ إجراء إلى دالة

لا مشكلة في الشيفرة إذ هي تعمل، ولكنّها فعليًا مقطّعة إربًا إربًا، ويصعُب على الشخص قراءتها وغالبًا تنتقل بنظرك بين أجزاء الشيفرة وأنت تقرأها. ليس هذا بالأمر الجيد خصوصًا لو لم يكن قارئ الشيفرة يعلم بما تفعله أو إلى أين ينقل نظره.

كما وأنّ الدوال بأسماء الخطوات `step*` هي لاستعمال واحد ولم نصنعها إلّا لتفادي "هرم العذابات". لن يأتي أحد لاحقًا ويُعيد استعمالها في أيّ شيء عدا سلسلة الإجراءات هذه. بهذا "نلوّث" فضاء الأسماء، إن جاز التعبير.

نريد بالطبع ما هو أفضل من هذا الحل. ولحسن حظنا (كالعادة)، فهناك طرق أخرى نتجنّب بها الأهرام هذه، أفضلها هي استعمال "الوعود" وسنشرحها في الفصل القادم.

11.1.4 تمارين

ا. دائرة تتحرّك ولها ردّ نداء

نرى في التمرين Animated circle دائرة يكبُر مقاسها في حركة جميلة.

لنقل بأنّا لا نريد الدائرة فقط، بل أيضًا عرض رسالة فيها. يجب أن تظهر الرسالة بعـدما ينتهي التحريـك (أي تكون الدائرة بمقاسها الكبير الكامل)، وإلّا بدا النص قبيح المظهر.

في حلّ ذاك التمرين، نرى الدالة `showCircle(cx, cy, radius)` ترسم الدائرة ولكن لا تُقدّم لنـا أيّ طريقة نعرف بها انتهاء الرسم.

أضِف وسيط ردّ نداء `showCircle(cx, cy, radius, callback)` يُستدعى متى اكتمل التحريك. على المُعامل `callback` استلام كائن `<div>` للدائرة وسيطًا له.

إليك مثالًا:

```
showCircle(150, 150, 100, div => {
  div.classList.add('message-ball');
  div.append("Hello, world!");
});
```

وإليك المثال الحي.

الحل:

يمكنك معاينة الحل من خلال المثال الحي.

11.2 الوعود Promise

لنقل بأنّك أنت هو عبد الحليم حافظ، ولنفترض بأنّ مُعجبوك من المحيط إلى الخليج يسألونك ليلًا نهارًا عن الأغنية الشاعرية التالية.

وكي تُريح بالك تعدهم بإرسالها إليهم ما إن تُنشر. فتُعطي مُعجبيك قائمة يملؤون فيها عناوين بريدهم. ومتى ما نشرت الأغنية يستلمها كلّ من في تلك القائمة. ولو حصل مكروه (لا سمح الله) مثل أن شبّت النار والتهمت الأستديو ولم تقدر على نشر الأغنية - لو حصل ذلك فسيعلمون به أيضًا.

وعاش الجميع بسعادة وهناء: أنت إذ لا يُزعجك الجميع بالتهديدات والتوعّدات، ومُعجبيك إذ لن تفوتهم أيّة رائعة من روائعك الفنية.

إليك ما يشبه الأمور التي نفعلها في الحياة الواقعية - في الحياة البرمجية:

1. "شيفرة مُنتِجة" تُنفّذ شيئًا وتأخذ الوقت. مثل الشيفرات التي تُحمّل البيانات عبر الشبكة. هذا أنت، "المغنّي".
2. "شيفرة مُستهلِكة" تطلب ناتج "الشيفرة المُنتِجة" ما إن يجهز. وهناك عديد من الدوال تحتاج إلى هذا الناتج. هذه "مُعجبوك".
3. الوعد (Promise) هو كائن فريد في جافاسكربت يربط بين "الشيفرة المُنتِجة" و"الشيفرة المُستهلِكة". في الحياة العملية، الوعد هو "قائمة الاشتراك". يمكن أن تأخذ "الشيفرة المُنتِجة" ما تلزم من وقت لتقدّم لنا النتيجة التي وعدتنا بها، وسيُجهّزها لنا "الوعد" لأيّة شيفرة طلبتها متى جهزت.

إن هذه المقاربة ليست دقيقة جدًا على الرغم من أنها جيدة كبداية ولكن وعود جافاسكربت أكثر تعقيدًا من قائمة اشتراك بسيطة بل لديها ميزات وقيود إضافية.

هذه صياغة الباني لكائنات الوعد:

```
let promise = new Promise(function(resolve, reject) {
  // المُنفِّذ (الشيفرة المُنتجة، مثل "المغنّي")
});
```

تُدعى الدالة الممرّرة إلى `new Promise` "بالمُنفِّذ". متى صُنع الوعد `new Promise` عملت الدالة تلقائيًا. يحتوي هذا المُنفِّذ الشيفرة المُنتِجة، ويمكن أن تقدّم لنا في النهاية ناتجًا. في مثالنا أعلاه، فالمُنفِّذ هذا هو "المغنّي".

تقدّم جافاسكربت الوسيطين `resolve` و `reject` وهما ردود نداء. كما ولا نضع الشيفرة التي نريد تنفيذها إلا داخل المُنفِّذ.

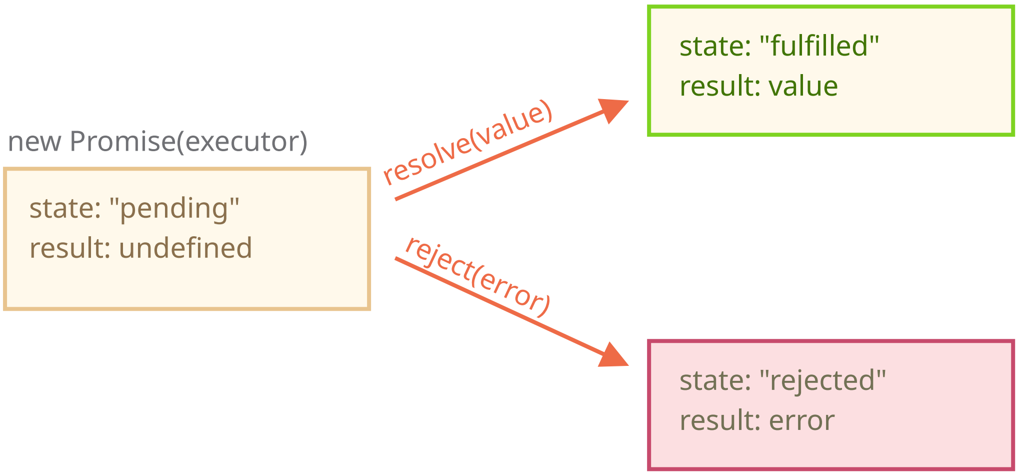
لا يهمّنا متى سيعرف المُنفِّذ الناتجَ (آجلًا كان ذلك أم عاجلًا)، بل أنّ عليه نداء واحدًا من ردود النداء هذه:

- `resolve(value)`: لو اكتملت المهمّة بنجاح. القيمة تسجّل في `value`.
- `reject(error)`: لو حدث خطأ. `error` هو كائن الخطأ.

إذًا نُلخّص: يعمل المُنفِّذ تلقائيًا وعليه مهمّة استدعاء `resolve` أو `reject`. لكائن الوعد `promise` الذي أعاده الباني `new Promise` خاصيتين داخليتين:

- الحالة `state`: تبدأ بالقيمة `"pending"` وبعدها تنتقل إلى `"fulfilled"` متى استُدعت `resolve`، أو إلى `"rejected"` متى استُدعت `reject`.
- الناتج `result`: يبدأ أولًا غير معرّف `undefined`، وبعدها يتغيّر إلى `value` متى استُدعت `resolve(value)` أو يتغيّر إلى `error` متى استُدعت `reject(error)`.

وفي النهاية ينقل المُنفِّذ الوعدَ `promise` ليصير بإحدى الحالات الآتية:



سنرى لاحقًا كيف سيشترك "مُعجبونا" بهذه التغييرات.

إليك مثالًا عن بانيًا للوعود ودالة مُنفِّذ بسيطة فيها "شيفرة مُنتجِة" تأخذ بعض الوقت (باستعمال `setTimeout`):

```
let promise = new Promise(function(resolve, reject) {
  // تُنفَّذ الدالة مباشرةً ما إن يُصنع الوعد
  // وبعد ثانية واحدة نبعث بإشارة بأنّ المهمة انتهت والنتيجة هي "تمت" (done)
  setTimeout(() => resolve("done"), 1000);
});
```

بتشغيل الشيفرة أعلاه، نرى أمرين اثنين:

1. يُستدعى المُنفِّذ تلقائيًا ومباشرةً (عند استعمال `new Promise`).
2. يستلم المُنفِّذ وسيطين: دالة الحلّ `resolve` ودالة الرفض `reject`، وهي دوال معرّفة مسبقًا في محرّك جافاسكربت، ولا داعٍ بأن نصنعها نحن، بل استدعاء واحدة ما إن تجهز النتيجة.

بعد سنة من عملية "المعالجة" يستدعي المُنفِّذ الدالةَ `resolve("done")` لتُنتج الناتج. هكذا تتغيّر حالة كائن `promise`:

كان هذا مثالًا عن مهمّة اكتملت بنجاح، أو "وعد تحقّق".

والآن سنرى مثالًا عن مُنفِّذ يرفض الوعد مُعيدًا خطأً:

```
let promise = new Promise(function(resolve, reject) {
  // بعد ثانية واحدة نبعث بإشارة بأنّ المهمة انتهت ونُعيد خطأ
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```

باستدعاء `reject(...)` ننقل حالة كائن الوعد إلى حالة الرفض `"rejected"`:

ملخّص القول هو أنّ على المُنفِّذ تنفيذ المهمة (أي ما يأخذ بعض الوقت ليكتمل) وثمّ يستدعي واحدةً من الدالتين `resolve` أو `reject` لتغيير حالة كائن الوعد المرتبط بالمُنفِّذ.

يُسمّى الوعد الذي تحقّق أو نُكث الوعد المنُجز، على العكس من الوعد المعلّق.

إما أن تظهر نتيجة واحدة أو خطأ

يجب على المنفّذ أن يستدعي إما `resolve` أو `reject`. أي تغيير في الحالة يعدّ تغييرًا نهائيًا.

وسيُتجاهل جميع الاستدعاءات اللاحقة سواءً أكانت `resolve` أو `reject`:

```
let promise = new Promise(function(resolve, reject) {

  resolve("done");

  reject(new Error("…")); // ستتجاهل
  setTimeout(() => resolve("…")); // ستتجاهل
});
```

الفكرة هنا أن خرج عمل المنفذّ سيعرض إما نتيجة معينة أو خطأ.

وتتوقع التعليمتين `resolve/reject` وسيطًا واحدًا مُررًا (أو بدون وسطاء نهائيًا) وأي وسطاء إضافية ستُتجاهل.

الرفض مع كائن Error

في حال حدوث خطأ ما، يجب على المنفذّ أن يستدعي تعليمة `reject`. ويمكن تمرير أي نوع من الوسطاء (تمامًا مثل: `resolve`). ولكن يوصى باستخدام كائنات `Error` (أو أي كائنات ترث من `Error`). وقريبًا سنعرف بوضوح سبب ذلك.

استدعاء resolve/reject الفوري

عمليًا عادة ينجز المنفذّ عمله بشكل متزامن ويستدعي `resolve/reject` بعد مرور بعض الوقت، ولكن الأمر ليس إلزاميًا، يمكننا استدعاء `resolve` أو `reject` فورًا، هكذا:

```
let promise = new Promise(function(resolve, reject) {
  // يمكننا القيام بالمهمة مباشرة
  resolve(123); // 123 :أظهر مباشرة النتيجة
});
```

على سبيل المثال من الممكن أن يحدث ذلك في حال البدء بمهمة معينة ولكن تكتشف بأن كلّ شيء أنجز وخزّن في الذاكرة المؤقتة.

هذا جيد، فعندها يجب أن ننجز الوعد فورًا.

الحالة state و النتيجة result الداخليتين

تكون خصائص الحالة state و النتيجة result لكائن الوعد داخلية. ولا يمكننا الوصول إليهم مباشرة. يمكننا استخدام التوابع ‎.then/.catch/.finally لذلك والتي سنشرحُها أدناه.

11.2.1 الاستهلاك: عبارات then وcatch وfinally

كائن الوعد هو كالوصلة بين المُنفِّذ (أي "الشيفرة المُنتِجة" أو "المغنّي") والدوال المُستهلكة (أي "المُعجبون") التي ستسلم الناتج أو الخطأ. يمكن تسجيل دوال الاستهلاك (أو أن تشترك، كما في المثال العملي ذاك) باستعمال التوابع ‎.then و‎.catch و‎.finally.‎.

ا. then

يُعدّ ‎.then أهمّها وعِماد القصة كلها. صياغته هي:

```
promise.then(
  function(result) { /* نتعامل مع الناتج الصحيح */ },
  function(error) { /* نتعامل مع الخطأ */ }
);
```

الوسيط الأوّل من التابع ‎.then يُعدّ دالة تُشغّل إن تحقّق الوعد، ويكون الوسيطُ الناتج. بينما الوسيط الثاني يُعدّ دالةً تُشغّل إن رُفض الوعد، ويكون الوسيطُ الخطأ.

إليك مثال نتعامل فيه مع وعد تحقّق بنجاح:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// .then تُنفَّذ resolve أول دالة في
promise.then(

  result => alert(result), // shows "done!" after 1 second

  error => alert(error) // doesn't run
);
```

هكذا نرى الدالة الأولى هي التي نُفّذت. وإليك المثال في حالة الرفض:

```
let promise = new Promise(function(resolve, reject) {
```

```
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// تُنفِّذ reject ثاني دالة في .then
promise.then(
  result => alert(result), // لا تعمل

  error => alert(error) // إظهار "Error: Whoops!" بعد ثانية

);
```

لو لم نُرِد إلّا حالات الانتهاء الناجحة، فيمكن أن نقدّم دالةً واحدة وسيطًا إلى `.then` فقط:

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
});


promise.then(alert); // إظهار "done!" بعد ثانية
```

ب. catch

لو لم نكن نهتمّ إلّا بالأخطاء، فعلينا استعمال `null` وسيطًا أولًا:

```
.then(null, errorHandlingFunction)
```

أو نستعمل `.catch(errorHandlingFunction)` وهو يؤدّي ذات المبدأ تمامًا ولا فرق إلّا قصر الثانية مقارنة بالأولى:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});


// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // إظهار "Error: Whoops!" بعد ثانية
```

ج. finally

كما يوجد `finally` في عبارات `try {...} catch {...}` العادية، فهناك مثلها في الوعود.

استدعاء `.finally(f)` يشبه استدعاء `.then(f, f)` ووجه الشبه هو أنّ الدالة `f` تعمل دومًا متى استقر الوعد أو أنجز سواءً كان قد تحقّق أو نُكث.

استعمال `finally` مفيد جدًا لتنظيف ما تبقّى من أمور مهمًا كان ناتج الوعد، مثل إيقاف أيقونات التحميل (فلم نعد نحتاج إليها). هكذا مثلًا:

```
new Promise((resolve, reject) => {
  // resolve/reject lre افعل شيئًا يستغرق وقتًا ثم استدعِ
})

  // تُنفذ عندما يُنجز الوعد في كل الأحوال بغض النظر عن حالته
  .finally(() => stop loading indicator)
  // بذلك نضمن إيقاف أيقونة التحميل قبل المضي قدمًا
  .then(result => show result, err => show error)
```

ولكنها ليست متطابقة تمامًا مع `then(f,f)`، فهناك فروقات مهمّة:

أولًا، ليس لدالة المُعالجة `finally` أيّ وسطاء. أي لسنا نعلم في `finally` أكان الوعد تحقّق أو نُكث، وهذه ليست مشكلة إذ ما نريده عادةً هو تنفيذ بعض الأمور "العامّة" لنُنهي ما بدأنا به.

ثانيًا، يمرُّ مُعالج `finally` على النتائج والأخطاء وبعدها إلى المعالج التالي. مثال على ذلك هو الناتج الذي تمرّر من `finally` إلى `then` هنا:

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("result"), 2000)
})
  .finally(() => alert("Promise ready"))
  .then(result => alert(result)); // <-- ستعالج الناتج .then
```

كما ترى القيمة `value` المعادة من الوعد تُمرر عبر `finally` إلى `then`، وهذا السلوك مفيد جدًا إذ لا يفترض بأن تتعامل `finally` مع ناتج الوعد، بل تمرّره إلى من يتعامل معه، وهي مخصصة إلى إجراء عمليات ختامية معينة (مثل التنظيف) بغض النظر عن الناتج.

وهنا واجه الوعد خطأً، وتمرّر من `finally` إلى `catch`:

```
new Promise((resolve, reject) => {
  throw new Error("error");
})
  .finally(() => alert("Promise ready"))
```

```
  .catch(err => alert(err));  // <-- error object الخطأ كائن ستعالج .catch
```

ثالثًا، معالج `finally` لا يجب أن يعيد شيئًا، وإن فعل، فسيجري تجاهل القيمة المعادة، والاستثناء الوحيد هنا هو عندما يرمي معالج `finally` خطأً ويُمرر آنذاك الخطأ إلى المعالج التالي بغض النظر عما سبقه من مخرجات.

الخلاصة، المعالج `finally`:

- لا يأخذ أي مخرجات من المعالج السابق، فليس لديه معاملات، بل تتخطاه المخرجات إلى المعالج التالي المناسب إن وُجد.
- لا يجب أن يعيد شيئًا وإن أعاد فلن يحصل شيء وتُتجاهل القيمة.
- إن رمى خطأً، فسيستلمه أقرب معالج أخطاء.

المُعالجات تعمل مباشرة في الوعود المنجزة

إن كان الوعد مُعلقًا لسببٍ ما، فإن معالجات `.then/catch/finally` ستنتظره. عدا ذلك (إن كان الوعد مُنجزًا) فإن المعالجات ستنفذّ مباشرةً:

```
// يصبح الوعد منجزًا ومتحققًا بعد الإنشاء مباشرةً
let promise = new Promise(resolve => resolve("done!"));

promise.then(alert); // done! (تظهر الآن)
```

الآن لنرى أمثلة عملية على فائدة الوعود في كتابة الشيفرات غير المتزامنة.

11.2.2 تحميل السكربتات: الدالة loadScript

أمامنا من الفصل الماضي الدالة `loadScript` لتحميل السكربتات. إليك الدالة بطريقة ردود النداء، لنتذكّرها لا أكثر ولا أقل:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  // خطأ في تحميل السكربت كذا
  script.onerror = () => callback(new Error(`Script load error for ${src}`));
  document.head.append(script);
}
```

هيا نُعد كتابتها باستعمال الوعود.

لن تطلب دالة `loadScript` الجديدة أيّ ردود نداء، بل ستصنع كائن وعد يتحقّق متى اكتمل التحميل، وتُعيده. يمكن للشيفرة الخارجية إضافة الدوال المُعالجة (أي دوال الاشتراك) إليها باستعمال `.then`:

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Script load error for ${src}`));
    document.head.append(script);
  });
}
```

الاستعمال:

```
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(
  script => alert(`${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);
promise.then(script => alert('Another handler...'));
```

بنظرة خاطفة يمكن أن نرى فوائد هذه الطريقة موازنةً بطريقة ردود النداء:

الوعود	ردود النداء
تتيح لنا الوعود تنفيذ الأمور بترتيبها الطبيعي أولًا نشغّل `loadScript(script)` ومن بعدها `.then` نكتب ما نريد فعله بالنتيجة.	يجب أن يكون تابِع `callback` تحت تصرفنا عند استدعاء `loadScript(script, callback)`. بعبارة أخرى يجب أن نعرف ما سنفعله بالنتيجة قبل استدعاء `loadScript`.
يمكننا استدعاء `then.` في الوعد عدة مرات كما نريد. في كلّ مرة نضيف معجب جديدة "fan"، هنالك تابع سيضيف مشتركين جُدد إلى قائمة المشتركين. سنرى المزيد حول هذا الأمر في الفصل القادم.	يمكن أن يكون هنالك ردّ واحد فقط.

إذًا، فالوعود تقدّم لنا تحكمًا مرنًا بالشيفرة وسير تنفيذها، وما زالت هنالك الكثير من الأمور الرائعة التي ستتعرف عليها الفصل القادم.

11.2.3 تمارين

ا. إعادة ... الوعد؟

ما ناتج الشيفرة أدناه؟

```
let promise = new Promise(function(resolve, reject) {
  resolve(1);

  setTimeout(() => resolve(2), 1000);
});

promise.then(alert);
```

الحل:

الناتج هو: 1. يُهمل استدعاء `resolve` الثاني إذ لا يتهمّ المحرّك إلّا بأول استدعاء من `reject/resolve`، والباقي كلّه يُهمل.

ب. التأخير باستعمال الوعود

تستعمل الدالة المضمّنة في اللغة `setTimeout` ردودَ النداء. اصنع واحدة تستعمل الوعود. على الدالة `delay(ms)` إعادة وعد ويجب أن ... هذا الوعد خلال `ms` مليثانية، ونُضيف تابِع `then.` إليه هكذا:

```
function delay(ms) {
  // شيفرتك هنا
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

الحل:

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

لاحظ أنّنا في هذا التمرين استدعينا `resolve` بلا وسطاء، ولم نُعد أيّ قيمة من `delay` بل ... فقط

ج. صورة دائرة متحركة مع وعد

أعِد كتابة الدالة `showCircle` في حلّ التمرين السابق لتُعيد وعدًا بدل أن تستلم ردّ نداء. ويكون استعمالها الجديد هكذا:

```
showCircle(150, 150, 100).then(div => {
  div.classList.add('message-ball');
  div.append("Hello, world!");
});
```

ليكن الحلّ في التمرين المذكور أساس المسألة الآن.

الحل:

يمكنك مشاهدة الحل عبر المثال الحي.

11.3 سَلسلة الوعود Promises chaining

طرحناها في الفصل "مقدمة إلى ردود النداء callbacks" مشكلةً ألا وهي أنّ لدينا تسلسلًا من المهام غير المتزامنة ويجب أن تُجرى واحدةً بعد الأخرى، مثلًا تحميل السكربتات. كيف نكتب شيفرة جيدة لهذه المشكلة؟

تقدّم لنا الوعود طرائق مختلفة لهذا الغرض. وفي هذا الفصل سنتكلّم عن سَلسلة الوعود فقط. هكذا تكون:

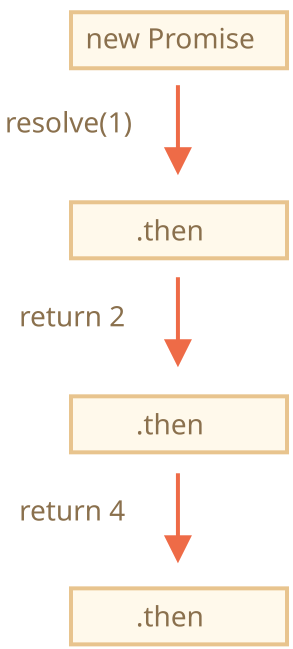
```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

الفكرة وما فيها هي تمرير الناتج في سلسلة توابِع `.then` تابعًا تابعًا.

هكذا تكون:

1. يبدأ الوعد الأوّل ويُنجز خلال ثانية واحدة `(*)`.
2. بعدها يُستدعى معالج `.then` `(**)`.
3. النتيجة التي ستعود ستمرر إلى معالج `.then` التالي `(***)`.
4. وهكذا...

نظـرًا لتمريــر النتيجــة على طــول سلســلة المعالجــات، يمكننــا رؤيــة سلســلة من اســتدعاءات `alert` هكذا: 1 ← 2 ← 4.



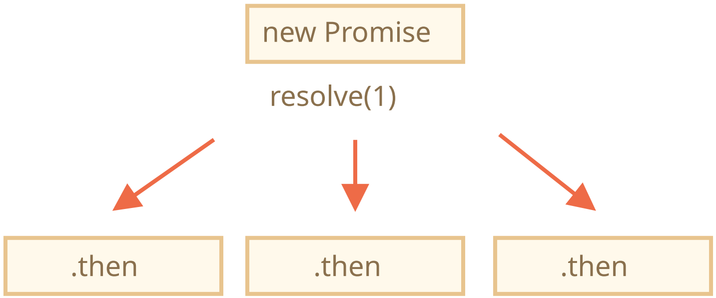
ويعـود سـبب هـذا كلّـه إلى أنّ اسـتدعاء `promise.then` يُعيـد وعـدًا هـو الآخـر، بـذلك يمكننـا اسـتدعاء التابع `.then` التالي على ذلك الوعد، وهكذا.

حين تُعيد دالة المُعاملة قيمةً ما، تصير القيمة ناتج ذلك الوعـد، بـذلك يمكن اسـتدعاء `.then` عليـه. خطـأ شائع بين المبتدئين: تقنيًا يمكننا إضافة أكثر من تابِع `.then` إلى وعد واحد. لا يُعدّ هذا سَلسلة وعود. مثلًا:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

```
promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

هنا كتبنا أكثر من دالة مُعاملة لوعد واحد، وهذه التوابع لا تمرّر القيمة إلى بعضها البعض، بل كلّ تعالجها على حدة. إليك الصورة (ووازِن بينها وبين السلسلة أعلاه):



تتلقّى كلّ توابِع `.then` في نفس الوعد ذات الناتج (أي ناتج الوعد) بذلك تعرض الشيفرة أعلاه نتائج `alert` متطابقة: 1. أمّا عمليًا فنادرًا ما نستعمل أكثر من دالة مُعاملة واحدة لكلّ وعد، على عكس السَلسلة التي يشيع استعمالها.

11.3.1 إعادة الوعود

يمكن لدالة المُعاملة (المستعملة في `.then(handler)`) إنشاء وعد وإعادته. هنا تنتظر دوال المُعاملة الأخرى حتّى يكتمل الوعد وتستلم ناتجه.

مثال على هذا:

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result); // 1

  // لاحِظ
  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });
```

```
}).then(function(result) { // (**)

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {

  alert(result); // 4

});
```

هنا يعرض أوّل تابِع `.then` القيمة 1 ويُعيد `new Promise(…)` في السطر `(*)`. بعد ثانية واحدة، يتحقق الوعد ويُمرّر ناتجه (أي وسيط التابع `resolve`، في حالتنا هو `result * 2`) إلى دالة المُعاملة التالية في تابِع `.then` التالي. نرى كيف أنّ الدالة في السطر `(**)` تعرض 2 وتؤدّي ما أدّته دالة المُعاملة السابقة.

بذلك نحصل على ما حصلنا عليه في المثال السـابق: 1 ثمّ 2 ثمّ 4، الفـرق هـو التـأخير لمـدّة ثانيـة بين كـلّ استدعاء من استدعاءات `alert`.

بإعادة الوعود يمكننا بناء سلسلة من الإجراءات غير المتزامنة.

11.3.2 مثال: loadScript

لنستعمل هذه الميزة مع دالة `loadScript` (التي كتبناها في الفصل السابق) لنُحمّل النصوص البرمجيـة واحدًا تلو الآخر:

```
loadScript("/article/promise-chaining/one.js")
  .then(function(script) {
    return loadScript("/article/promise-chaining/two.js");
  })
  .then(function(script) {
    return loadScript("/article/promise-chaining/three.js");
  })
  .then(function(script) {
    // نستعمل الدوال المعرّف عنها في النصوص البرمجية
```

```
    // ونتأكّد تمامًا بأنّها حُمّلت
    one();
    two();
    three();
  });
```

يمكننا أيضًا تقصير الشيفرة قليلًا باستعمال الدوال السهميّة:

```
loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // اكتمل تحميل النصوص، يمكننا استعمال الدوال فيها الآن
    one();
    two();
    three();
  });
```

نرى هنا أنّ كلّ استدعاء من استدعاءات `loadScript` يُعيد وعدًا، ويعمل تابِع `.then` التالي في السلسلة متى أنجز أو تحقق الوعد. بعدها تبدأ الدالة بتحميل النص البرمجي التالي وهكذا تُحمّل كلّ النصوص واحدًا بعد آخر. ويمكننا إضافة ما نريد من إجراءات غير متزامنة إلى السلسلة، ولن يزيد طول الشيفرة إلى اليمين، بل إلى أسفل، ولن نُقابل وجه هرم ردود النداء القبيح ثانيةً.

يمكننا تقنيًا إضافة تابِع `.then` داخل دوال `loadScript` مباشرةً هكذا:

```
loadScript("/article/promise-chaining/one.js").then(script1 => {
  loadScript("/article/promise-chaining/two.js").then(script2 => {
    loadScript("/article/promise-chaining/three.js").then(script3 => {

      // يمكن أن تصل هذه الدالة إلى المتغيّرات script1 وscript2 وscript3
      one();
      two();
      three();
    });
  });
});
```

وتؤدّي الشيفرة نفس العمل: تُحمّل 3 نصوص برمجية بالترتيب. المشكلة هي أنّ طولها يزيد نحو اليمين وهي نفس مشكلة ردود النداء.

عادةً ما يجهل المبرمجون الجدد الذين يستعملون الوعود ميزة السَلسلة، فيكتبون الشيفرات هكذا. لكنّ سَلسلة الوعود هي الأمثل وغالبًا الأفضل.

ولكنّ استعمال `.then` مباشرةً أحيانًا لا يكون بالمشكلة الكبيرة، إذ يمكن للدوال المتداخلة الوصول إلى المجال الخارجي. في المثال أعلاه مثلًا يمكن لآخر ردّ نداء متداخل الوصول إلى كلّ المتغيّرات `script1` و`script2` و`script3`، إلّا أنّ هذا استثناء عن القاعدة وليس قاعدة بحدّ ذاتها.

‎.I كائنات قابلة للسلسلة Thenables

على وجه الدقة، لا تعيد المعالجات وعودًا وإنما تعيد كائن قابل للسلسلة thenable - وهو كائن عشوائي له التابع `.then` ويتعامل معه بنفس طريقة التعامل مع الوعد.

الفكرة أن مكتبات الخارجية تنفذ كائنات خاصة بها "متوافقة مع الوعد". ويمكن أن يملكوا مجموعة توابع موسّعة. ولكن يجب أن يتوافقوا مع الوعود الأصيلة، لأنهم ينفذون `.then`..

وإليك مثالًا على كائن thenable:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { native code }
    // إنجاز الوعد وتحقيقه مع this.num*2 بعد ثانية
    setTimeout(() => resolve(this.num * 2), 1000); // (**)
  }
}

new Promise(resolve => resolve(1))
  .then(result => {
    return new Thenable(result); // (*)
  })
  .then(alert); // إظهار 2 بعد 1000 ميلي ثانية
```

تتحقق جافاسكربت من الكائن المُعاد من معالج `.then` في السطر `(*)`: إن لديه تابع قابل للاستدعاء يدعى `then` عندها سيستدعي ذاك التابع مزودًا بذلك بالتوابع الأصيلة مثل: `resolve` و `reject` كوسطاء

(مشابه للمنفذ) وينتظر حتى يستدعى واحدًا منهم. في المثال أعلاه تستدعى `resolve(2)` بعد ثانية انظر `(**)`. بعدها تمرر النتيجة إلى أسفل السلسلة.

تتيح لنا هذه المميزات دمج الكائنات المخصصة مع سلاسل الوعود دون الحاجة إلى الوراثة من الوعد `Promise`.

11.3.3 مثال أضخم: fetch

عادةً ما نستعمل الوعود في برمجة الواجهات الرسومية لطلبات الشبكة. لنرى الآن مثالًا أوسع مجالًا قليلًا.

سنستعمل التابع لتحميل بعض المعلومات التي تخصّ المستخدم من الخادم البعيد. لهذا التابع معاملات كثيرة اختيارية كتبنا عنا في فصول مختلفة، إلّا أنّ صياغته الأساسية بسيطة إلى حدّ ما:

```
let promise = fetch(url);
```

هكذا نُرسل طلبًا شبكيًا إلى العنوان `url` ونستلم وعدًا يُحل resolved مع قيمة الكائن `response` ما إن يردّ الخادم البعيد بترويسات الطلب، ولكن قبل تنزيل الردّ كاملًا.

علينا استدعاء التابع `response.text()` لقراءة الردّ كاملًا، وهو يُعيد وعدًا يُحل متى نُزّل النص الكامل من الخادم البعيد، وناتجه يكون ذلك النص.

تُرسل الشيفرة أسفله طلبًا إلى `user.json` وتحمّل نصّه من الخادم:

```
fetch('/article/promise-chaining/user.json')
  // إن .then تعمل عندما يستجيب الخادم البعيد
  .then(function(response) {
    // إن التابع response.text() يُعيد وعدًا جديدًا والذي يعاد مع كامل نص الاستجابة
    // عندما يُحمّل
    return response.text();
  })
  .then(function(text) {
    // ...وهنا سيكون محتوى الملف البعيد
    alert(text); // {"name": "iliakan", isAdmin: true}
  });
```

كما أنّ هناك التابع `response.json()` والذي يقرأ البيانات المستلمة البعيدة ويحلّلها على أنّها JSON. في حالتنا هذا أفضل وأسهل فهيا نستعمله.

كما وسنستعمل الدوال السهميّة للاختصار قليلًا:

```
// JSON مشابه للمثال أعلاه ولكن التابع response.json() يحلل المحتوى البعيد كملف
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => alert(user.name)); // iliakan, got user name
```

الآن لنصنع شيئًا بهذا المستخدم الذي حمّلناه.

يمكننا مثلًا إجراء طلبات أكثر من غِتهَب وتحميل ملف المستخدم الشخصي وعرض صورته:

```
// user.json أنشئ طلب لِ
fetch('/article/promise-chaining/user.json')
  // json حمله وكأنه ملف
  .then(response => response.json())
  // GitHub أنشئ طلب لِ
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  // json حمّل الرد كملف
  .then(response => response.json())
  // أظهر الصورة الرمزية (avatar) من (githubUser.avatar_url) لمدة 3 ثواني
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => img.remove(), 3000); // (*)
  });
```

الشيفرة تعمل على أكمل وجه (طالِع التعليقات لتعرف التفاصيل). ولكن هناك مشكلة فيه قد تحدث، وهي خطأ شائع يقع فيه من يستعمل الوعود أوّل مرّة.

طالِع السطر (*): كيف يمكن أن نفعل مهمّة معينة متى اكتمل عرض الصورة وأُزيلت؟ فلنقل مثلًا سنعرض استمارة لتحرير ذلك المستخدم أو أيّ شيء آخر. حاليًا... ذلك مستحيل.

لنقدر على مواصلة السلسلة علينا إعادة وعد المُنجز متى اكتمل عرض الصورة. هكذا:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
```

```
  // هنا
  .then(githubUser => new Promise(function(resolve, reject) { // (*)
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser); // (**)
    }, 3000);
  }))
  // يحدث بعد 3 ثوانٍ
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
```

هكذا صارت تُعيد دالة المُعاملة في `.then` عند السطر `(*)` كـائنَ `new Promise` لا ... إلّا بعـد اسـتدعاء `resolve(githubUser)` في `setTimeout` عند `(**)`. وسينتظر تابِع `.then` التالي في السلسـلة اكتمـال ذلك. تُعد إعادة الإجراءات غير المتزامنة للوعود دومًا إحدى الممارسات الصحيحة في البرمجة.

هكذا يسهّل علينا التخطيط للإجراءات التي ستصير بعد هذا، فحتّى لو لم نريـد توسـعة السلسـلة الآن لربّمـا احتجنا إلى ذلك لاحقًا. وأخيرًا، يمكننا أيضًا تقسيم الشيفرة إلى دوال يمكن إعادة استعمالها:

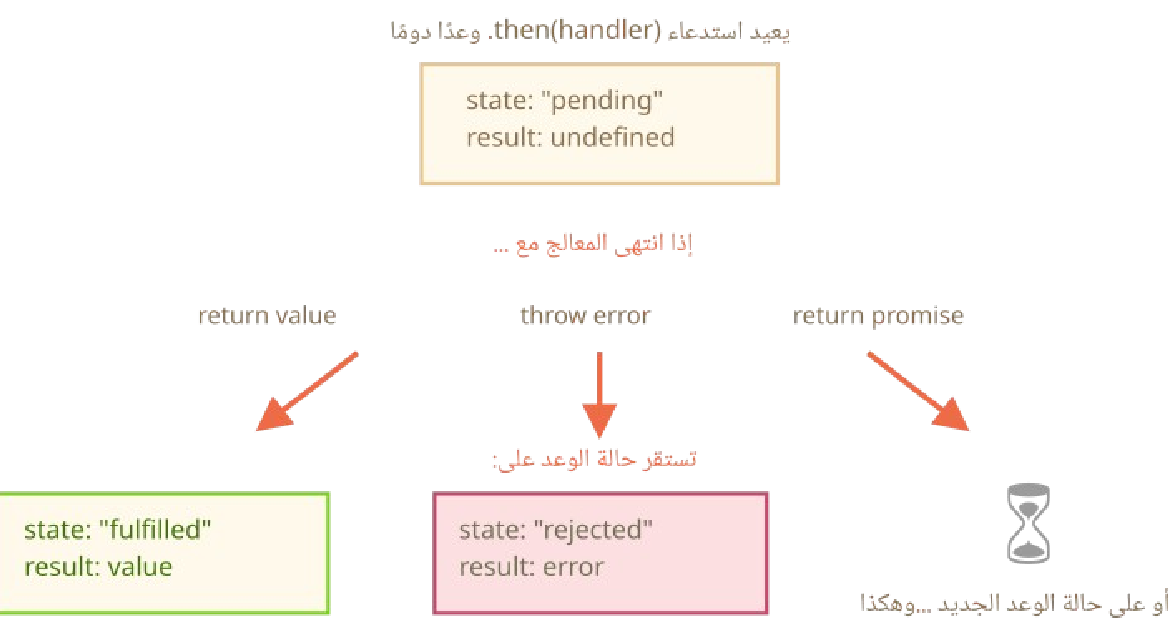
```
function loadJson(url) {
  return fetch(url)
    .then(response => response.json());
}

function loadGithubUser(name) {
  return fetch(`https://api.github.com/users/${name}`)
    .then(response => response.json());
}

function showAvatar(githubUser) {
  return new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
```

```
      document.body.append(img);

      setTimeout(() => {
        img.remove();
        resolve(githubUser);
      }, 3000);
    });
  }
  // نستعملها الآن:
  loadJson('/article/promise-chaining/user.json')
    .then(user => loadGithubUser(user.name))
    .then(showAvatar)
    .then(githubUser => alert(`Finished showing ${githubUser.name}`));
   // اكتمل عرض كذا  // ...
```

11.3.4 الخلاصة

إن أعادت دالة مُعاملة `.then` (أو `catch/finally`، لا يهمّ حقًا) وعدًا، فتنتظر بقية السلسلة حتّى تُنجز متى حدث ذلك يُمرّر الناتج (أو الخطأ) إلى التي بعدها.

إليك الصورة الكاملة:



11.3.5 تمارين

ا. الوعد: catchg then

هل تؤدّي هاتين الشيفرتين نفس الغرض؟ أي هل يتطابق سلوكهما في الحالات المختلفة، وأيّما كانت دوال المُعاملة؟

```
promise.then(f1).catch(f2);
```

مقابل:

```
promise.then(f1, f2);
```

الحل:

الجواب المختصر: لا ليسا متساويين، إذ الفرق أنه إن حدث خطأ في `f1` فستعالجها `.catch`. انظر:

```
promise
  .then(f1)
  .catch(f2);
```

...لكن ليس هنا:

```
promise
  .then(f1, f2);
```

وذلك بسبب تمرير الخطأ لأسفل السلسلة، وفي الجزء الثاني من الشيفرة لا يوجد سلسلة أقل من `f1`.

بمعنى آخر يمرر `.then` النتيجة أو الخطأ إلى `.then/catch` التالية. لذا في المثال الأول يوجد `catch` بينما في المثال الثاني لا يوجد. ولذلك لم يعالج الخطأ.

11.4 التعامل مع الأخطاء then/catch

تُعدّ سلاسل الوعود ممتازة في التعامل مع الأخطاء، فمتى رُفض الوعد ينتقل سير التحكّم إلى أقرب دالة تتعامل مع حالة الرفض، وهذا عمليًا يسهّل الأمور كثيرًا.

فمثلًا نرى في الشيفرة أسفله أنّ العنوان المرّر إلى `fetch` خطأ (ما من موقع بهذا العنوان) ويتعامل التابع `.catch` مع الخطأ:

```
fetch('https://no-such-server.blabla') // هذا الموقع من وحي الخيال العلمي
  .then(response => response.json())
  .catch(err => alert(err)) // TypeError: failed to fetch (يمكن أن يتغيّر النصّ بتغيّر الخطأ)
```

وكما ترى فليس ضروريًا أن يكون التابع `.catch` مباشرةً في البداية، بل يمكن أن يظهر بعد تابِع `.then` واحد أو أكثر حتّى.

أو قد يكون الموقع سليمًا ولكنّ الردّ ليس كائن JSON صالح. الطريقة الأسهل في هذه الحالة لالتقاط كلّ الأخطاء هي بإضافة تابِع `.catch` إلى نهاية السلسلة:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise((resolve, reject) => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  .catch(error => alert(error.message)); // هنا
```

الطبيعي هو ألّا يعمل تابِع `.catch` مطلقًا، ولكن لو رُفض أحد الوعود أعلاه (بسبب مشكلة في الشبكة أو كائن غير صالح أو أيّ شيء آخر)، فسيستلم التابع الخطأ.

11.4.1 صياغة try..catch الضمنية

تُحيط بشيفرتي مُنفّذ الوعود ودوال مُعاملة الوعود عبارة `try..catch` مخفية إن صحّ القول، فإن حدث استثناء يُلتقط ويتعامل معه المحرّك على أنّه حالة رفض. خُذ مثلًا هذه الشيفرة:

```
new Promise((resolve, reject) => {
  throw new Error("Whoops!"); // لاحِظ
}).catch(alert); // Error: Whoops!
```

وهل تعمل تمامًا مثل عمل هذه:

```
new Promise((resolve, reject) => {
  reject(new Error("Whoops!")); // لاحِظ
}).catch(alert); // Error: Whoops!
```

تتلقّى عبارة `try..catch` المخفي المُحيطة بالمُنفّذ الأخطاءَ تلقائيًا وتحوّلها إلى وعود مرفوضة.

ولا يحدث هذا في دالة المُنفّذ فحسب بل أيضًا في دوال المُعاملة. فلو رمينا شيئًا في دالة المُعاملة للتابِع `throw` فيعني ذلك بأنّ الوعد رُفض وينتقل سير التحكّم إلى أقرب دالة تتعامل مع الأخطاء.

إليك مثالًا:

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  throw new Error("Whoops!"); // نرفض الوعد
}).catch(alert); // Error: Whoops!
```

وهذا يحدث مع الأخطاء كافّة وليس فقط لتلك التي رُميت بعبارة `throw`. حتّى أخطاء المبرمجين، انظر:

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  blabla(); // ما من دالة كهذه
}).catch(alert); // ReferenceError: blabla is not defined
```

تابِع `.catch` الأخير لا يستلم حالات الرفض الصريحة فحسب، بل تلك التي تحدث أحيانًا في دوال المُعاملة أعلاه أيضًا.

11.4.2 إعادة الرمي

كما رأينا فوجود تابِع `.catch` نهاية السلسلة شبيه بعبارة `try..catch`. يمكن أن نكتب ما نريد من دوال مُعاملة `.then` وثمّ استعمال تابِع `.catch` واحد في النهاية للتعامل مع أخطائها.

في عبارات `try..catch` العاديّة نحلّل الخطأ ونُعيد رميه لو لم نستطع التعامل معه. ذات الأمر مع الوعود. فإن رمينا شيئًا داخل `.catch`، ينتقل سير التحكّم إلى أقرب دالة تتعامل مع الأخطاء. وإن تعاملنا مع الخطأ كما يجب فيتواصل إلى أقرب دالة `.then`.

في المثال أسفله يتعامل التابع `.catch` مع الخطأ كما ينبغي:

```
// catch -> then :سلسلة التنفيذ
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) {

  alert("The error is handled, continue normally");

}).then(() => alert("Next successful handler runs"));
```

لا تنتهي هنا كتلة `.catch` البرمجية بأيّ أخطاء، بذلك تُستدعى دالة المُعاملة في تابِع `.then` التالي.

نرى في المثال أسفله الحالة الثانية للتابِع `.catch`. تلتقط دالة المُعاملة عند `(*)` الخطأ ولكن لا تعرف التعامل معه (مثلًا لا تعرف إلّا أخطاء `URIError`) فترميه ثانيةً:

```
// catch → catch → then :سلسلة التنفيذ
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) { // (*)

  if (error instanceof URIError) {
    // handle it
  } else {
    alert("Can't handle such error");
```

```
    throw error;  // رمي هذا الخطأ أو أي خطأ آخر سينقُلنا إلى catch التالية
  }

}).then(function() {
  /* لن يعمل هنا */
}).catch(error => { // (**)

  alert(`The unknown error has occurred: ${error}`);
  // لن يعيد أي شيء=>عملية التنفيذ حدثت بطريقة عادية
});
```

ينتقل سير التنفيذ من تابِع `.catch` الأول عند `(*)` إلى التالي عند `(**)` في السلسلة.

11.4.3 حالات رفض

ماذا يحدث لو لم نتعامل مع الخطأ؟ فنقل مثلًا نسينا إضافة تابِع `.catch` في نهاية السلسلة هكذا:

```
new Promise(function() {
  noSuchFunction(); // خطأ (لا يوجد هذا التابع)
})
  .then(() => {
    // معالجات الوعد الناجح سواء واحدة أو أكثر
  });
// بدون .catch في النهاية!
```

لو حدث خطأ يُرفض الوعد وعلى سير التنفيذ الانتقال إلى أقرب دالة تتعامل مع حالات الرفض. ولكن ما من دالة كهذه و"يعلق" الخطأ إن صحّ التعبير إذ ما من شيفرة تتعامل معه. عمليًا يتشابه هذا مع الأخطاء التي لم نتعامل معها في الشيفرات، أي أنّ شيئًا مريعًا قد حدث.

تتذكّر ما يحدث لو حدث خطأ عادي ولم تلتقطه عبارة `try..catch`؟ "يموت" النص البرمجي ويترك رسالةً في الطرفية. ذات الأمر يحدث مع حالات رفض الوعود التي لم يجري التعامل معها.

يتعقّب محرّك جافاسكربت هذه الحالات ويولّد خطأً عموميًا في هذه الحالة. يمكنك أن تراه في الطرفية إن شغلت المثال أعلاه.

يمكننا في المتصفّحات التقاط هذه الأخطاء باستعمال الحدث `unhandledrejection`:

```
window.addEventListener('unhandledrejection', function(event) {
  // لدى كائن الحدث خاصيتين مميزتين:
  alert(event.promise); // [object Promise] - الوعد الذي يولد الخطأ
  alert(event.reason); // Error: Whoops! - كائن الخطأ غير المعالج
});

new Promise(function() {
  throw new Error("Whoops!");
}); // لا يوجد catch لمعالجة الخطأ
```

إنّما الحدث هو جزء من معيار HTML.

لو حدث خطأ ولم يكن هناك تابِع `.catch` فيُشغّل معالج `unhandledrejection` ويحصل على كائن الحدث مع معلومات حول الخطأ حتى نتمكن من فعل شيء ما.

مثل هذه الأخطاء عادةً ما تكون غير قابلة للاسترداد، لذلك أفضل طريقة للخروج هي إعلام المستخدم بالخطأ أو حتى إبلاغ الخادم بالخطأ. ثمّة في البيئات الأخرى غير المتصفّحات (مثل Node.js) طرائقَ أخرى لتعقّب الأخطاء التي لم نتعامل معها.

11.4.4 الخلاصة

- يتعامل `.catch` مع الأخطاء في الوعود أيًا كانت: أكانت من استدعاءات `reject()` أو من رمي الأخطاء في دوال المُعاملة.
- يجب أن نضع `.catch` في الأماكن التي نريد أن نعالج الخطأ فيها ومعرفة كيفية التعامل معها. يجب على المعالج تحليل الأخطاء (الأصناف المخصصة تساعدنا بذلك) وإعادة رمي الأخطاء غير المعروفة (ربما تكون أخطاء برمجية).
- لا بأس بعدم استخدام `.catch` مطلقًا، إن لم يكُ هنالك طريقة للاسترداد من الخطأ.
- على أية حال، يجب أن يكون لدينا معالج الأحداث `unhandledrejection` (للمتصفحات وللبيئات الأخرى) وذلك لتتبع الأخطاء غير المُعالجة وإعلام المستخدم (وربما إعلام الخادم) عنها. حتى لا يموت تطبيقنا مطلقًا.

11.4.5 تمارين

ا. خطأ في تابِع setTimeout

ما رأيك هل ستعمل `.catch`؟ وضح إجابتك.

```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

الجواب: لا لن تعمل:

```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

كما ذكرنا في هذا الفصل لدينا "صياغة `try..catch` ضمنية" حول تابع معيّن. لذلك تُعالج جميع الأخطاء المتزامنة. ولكن هنا الخطأ لا يُنشأ عندما يعمل المُنفذّ وإنما في وقت لاحق. لذا فإن الوعد لن يستطيع معالجته.

11.5 واجهة الوعود البرمجية

ثمّة 5 توابِع ثابتة (static) في صنف الوعود `Promise`. سنشرح الآن عن استعمالاتها سريعًا.

11.5.1 Promise.all

لنقل بأنّك تريد تنفيذ أكثر من وعد واحد في وقت واحد، والانتظار حتّى تجهز جميعها. مثلًا أن تُنزّل أكثر من عنوان URL في آن واحد وتُعالج المحتوى ما إن تُنزّل كلها.

وهذا الغرض من وجود `Promise.all`. إليك صياغته:

```
let promise = Promise.all([...promises...]);
```

يأخذ التابع `Promise.all` مصفوفة من الوعود (تقنيًا يمكن أن تكون أيّ ... ولكنّها في العادة مصفوفة) ويُعيد وعدًا جديدًا.

لا يُحلّ الوعد الجديد إلّا حين تستقرّ الوعود في المصفوفة، وتصير تلك المصفوفة التي تحمل نواتج الوعود - تصير ناتج الوعد الجديد. مثال على ذلك تابِع `Promise.all` أسفله إذ يستقرّ بعد 3 ثوان ويستلم نواتجه في مصفوفة `[1, 2, 3]`:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // 1,2,3 ما إن تجهز الوعود يُعطي كلّ واحد عنصرًا في المصفوفة
```

لاحِظ كيف أنّ ترتيب عناصر المصفوفة الناتجة يتطابق مع ترتيب الوعود، فحتّى لو أخذ الوعد الأوّل وقتًا أطول من غيره حتّى يُحلّ، فسيظلّ العنصر الأول في مصفوفة النواتج.

نستعمل عادةً حيلة صغيرة أن نصنع مصفوفة جديدة بخارطة تأخذ القديمة وتحوّلها إلى وعود، ثمّ نمرّر ذلك كلّه إلى `Promise.all`. فمثلًا لو لدينا مصفوفة من العناوين، يمكننا جلبها كلّها هكذا:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];
// fetch نحوّل كلّ عنوان إلى وعد التابِع
let requests = urls.map(url => fetch(url));
```

```
// ينتظر Promise.all حتّى تُحلّ كلّ المهام
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url}: ${response.status}`)
  ));
```

من أكبر الأمثلة على جلب المعلومات. هي جلب المعلومات لمجموعة من مستخدمي موقع غيتهب GitHub من خلال أسمائهم (يمكننا جلب مجموعة من السلع بحسب رقم المعرف الخاص بها، منطق الحل متشابه في كِلا الحالتين):

```
let names = ['iliakan', 'remy', 'jeresig'];
let requests = names.map(name => fetch(`https://api.github.com/users/$
{name}`));

Promise.all(requests)
  .then(responses => {
    // استدعيت جميع الردود بنجاح
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`); // أظهر 200 من أجل كلّ
عنوان url
    }
    return responses;
  })
  // اربط مصفوفة الردود مع مصفوفة response.json() لقراءة المحتوى
  .then(responses => Promise.all(responses.map(r => r.json())))
  // تحلل جميع الإجابات : وتكوّن مصفوفة "users" منهم
  .then(users => users.forEach(user => alert(user.name)));
```

لو رُفض أيّ وعد من الوعود، سيرفض الوعد الذي يُعيده `Promise.all` مباشرةً بذلك الخطأ. مثال:

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1),
1000)),
  // هنا
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

نرى هنا أنّ الوعد الثاني يُرفض بعد ثانيتين، ويؤدّي ذلك إلى رفض `Promise.all` كاملًا، وهكذا يُنفّذ التابع `.catch` ويصير الخطأ ناتج `Promise.all` كلّه.

> في حالة حدوث خطأ ما، تُتجاهل الوعود الأُخرى، فلو أن وعدًا ما رُفضَ، ستُرفض جميع الوعود من خلال تابع `Promise.all` مباشرةً، متناسيًا بذلك الوعود الأخرى في القائمة. وعندها ستُتجاهل نتائجها أيضًا.
> على سبيل المثال، إن كان العديد من استدعاءات `fetch` كما هو موضح في المثال أعلاه، وفشِل أحدها، فسيستمر تنفيذ الاستدعاءات الأخرى. ولكن لن يشاهدها تابع `Promise.all` بعد الآن. ربما ستُنجز بقية الوعود، ولكن نتائجها ستُتجاهل في نهاية المطاف.
> لن يُلغي التابع `Promise.all` الوعود الأخرى، إذ لا يوجد مثل هذا الفعل في الوعود، سنغطي في فصل آخر طريقة المتبعة في إلغاء الوعود وسيساعدنا التابع `AbortController` في ذلك، ولكنه ليس جزءًا من واجهة الوعود البرمجية.

يقبل التابع Promise.all(iterable) بغير الوعود مثل أي نوع قيم أخرى

يقبل التابع `Promise.all(...)` وعودًا قابلة للتكرار (مصفوفات في معظم الوقت). ولكن لو لم تكُ تلك الكائنات وعودًا. فستُمرّر النتائج كما هي.

فمثلًا، النتائج هنا `[1, 2, 3]`:

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  }),
  2,
  3
]).then(alert); // 1, 2, 3
```

يمكننا تمرير القيم الجاهزة لتابِع `Promise.all` عندما يكون ذلك مناسبًا.

11.5.2 Promise.allSettled

> هذه إضافة حديثة للغة، وقد يحتاج إلى تعويض نقص الدعم في المتصفحات القديمة.

لو رُفض أحد الوعود في `Promise.all` فسيُرفض كلّه مجتمعًا. هذا يفيدنا لو أردنا استراتيجية "إمّا كل شيء أو لا شيء"، أي حين نطلب وجود النتائج كلها كي نواصل:

```
Promise.all([
  fetch('/template.html'),
  fetch('/style.css'),
  fetch('/data.json')
```

```
]).then(render); // تابع التصيير يطلب نواتج كلّ توابع الجلب
```

بينما ينتظر `Promise.allSettled` استقرار كلّ الوعود. للمصفوفة الناتجة هذه العناصر:

- `{status:"fulfilled", value:result}` من أجل الاستجابات الناجحة.
- `{status:"rejected", reason:error}` من أجل الأخطاء.

مثال على ذلك هو لو أردنا جلب معلومات أكثر من مستخدم واحد. فلو فشل أحد الطلبات ما زلنا نريد معرفة معلومات البقية. فلنستعمل `Promise.allSettled`:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*)
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`${urls[num]}: ${result.reason}`);
      }
    });
  });
```

ستكون قيمة `results` في السطر `(*)` أعلاه كالآتي:

```
[
  {status: 'fulfilled', value: ...ردّ...},
  {status: 'fulfilled', value: ...ردّ...},
  {status: 'rejected', reason: ...كائن خطأ...}
]
```

هكذا نستلم لكلّ وعد حالته وقيمته أو الخطأ `value/error`.

ا. تعويض نقص الدعم

لو لم يكن المتصفّح يدعم `Promise.allSettled` فمن السهل تعويض ذلك:

```
if(!Promise.allSettled) {
  Promise.allSettled = function(promises) {
    return Promise.all(promises.map(p => Promise.resolve(p).then(value
=> ({
      state: 'fulfilled',
      value
    }), reason => ({
      state: 'rejected',
      reason
    }))));
  };
}
```

في هذه الشيفرة يأخذ التابع `promises.map` قيم الإدخال ويحوّلها إلى وعود (في حال وصله شيء ليس بالوعد) ذلك باستعمال `p => Promise.resolve(p)`، بعدها يُضيف دالة المُعاملة `.then` لكلّ وعد.

هذه الدالة تحوّل النواتج الناجحة `v` إلى `{state:'fulfilled', value:v}`، والأخطاء `r` إلى `{state:'rejected', reason:r}`. هذا التنسيق الذي يستعمله `Promise.allSettled` بالضبط.

يمكننا لآن استعمال `Promise.allSettled` لجلب نتائج كلّ الوعود الممرّرة حتّى لو رفضت بعضها.

11.5.3 Promise.race

يشبه التابع `Promise.all` إلّا أنّه ينتظر استقرار وعد واحد فقط ويأخذ ناتجه (أو الخطأ). صياغته هي:

```
let promise = Promise.race(iterable);
```

فمثلًا سيكون الناتج هنا 1:

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1),
1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

أوّل الوعود هنا كان أسرعها وهكذا صار هو الناتج. ومتى "فاز أوّل وعد مستقرّ بالسباق"، تُهمل بقية النواتج/الأخطاء.

11.5.4 Promise.resolve/reject

نادرًا ما نستعمل `Promise.resolve` و `Promise.reject` في الشيفرات الحديثة إذ صياغة `async/await` (نتكلم عنها في فصل لاحق) نشرحها هنا ليكون شرحًا كاملًا، وأيضًا لمن لا يستطيع استعمال `async/await` لسبب أو لآخر.

يُنشئ `Promise.resolve(value)` وعدًا مُنجزًا بالناتج `value` ويشبه:

```
let promise = new Promise(resolve => resolve(value));
```

يستخدم هذا التابع للتوافقية، عندما يُتوقع من تابع ما أن يُعيد وعدًا. فمثلًا، يجلب التابع أدناه `loadCached` عنوان URL ويخزن المحتوى في الذاكرة المؤقتة. ومن أجل الاستدعاءات اللاحقة لنفس عنوان URL سيحصل على المحتوى فورًا من الذاكرة المؤقتة، ولكنه يستخدم التابع `Promise.resolve` لتقديم وعدًا بهذا الأمر. لتكون القيمة المرتجعة دائمًا وعدًا:

```
let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {

    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url,text);
      return text;
    });
}
```

يمكننا كتابة `loadCached(url).then(…)` لأننا نضمن أن التابع سيُعيد وعدًا. كما يمكننا دائمًا استخدام `.then` بعد تابِع `loadCached`. وهذا هو الغرض من `Promise.resolve` في السطر `(*)`.

ا. Promise.reject

ينشئ `Promise.reject(error)` وعدًا مرفوضًا مع خطأ ويشبه:

```
let promise = new Promise((resolve, reject) => reject(error));
```

عمليًا لا نستعمل هذا التابع أبدًا.

11.5.5 الخلاصة

لصنف الوعود `Promise` خمس توابِع ثابتة:

1. `Promise.all(promises)`: ينتظر جميع الوعود لتعمل ويعيد مصفوفة بنتائجهم. وإذا رُفض أي وعدٍ منهم، سيرجع `Promise.all` خطأً، وسيتجاهلُ جميع النتائج الأخرى.
2. `Promise.allSettled(promises)`: (التابع مضاف حديثًا) ينتظر جميع الوعود لتُنجز ليُعيد نتائجها كمصفوفة من الكائنات تحتوي على:
   - `state`: الحالة وتكون إما `"fulfilled"` أو `"rejected"`.
   - `value`: القيمة (إذا تحقق الوعد) أو `reason` السبب (إذا رُفض).
3. `Promise.race(promises)`: ينتظر الوعد الأول ليُنجز وتكون نتيجته أو الخطأ الذي سيرميه خرج هذا التابع.
4. `Promise.resolve(value)`: ينشئ وعدًا منجزًا مع القيمة الممرّرة.
5. `Promise.reject(error)`: ينشئ وعدًا مرفوضًا مع الخطأ المُمرّر.

ومن بينها `Promise.all` هي الأكثر استعمالًا عمليًا.

11.6 الدوال الواعدة: تحويل الدوال إلى وعود Promisification

تحويل الدوال إلى وعود (Promisification) هي عملية تغليف الدالة التي تستلم ردّ نداء لتصبح دالة تُعيد وعدًا. وفي الحياة العملية فهذا النوع من التحويل مطلوب جدًا إذ تعتمد العديد من الدوال والمكتبات على ردود النداء. ولكن… الوعود أسهل وأفضل لذا من المنطقي تحويل تلك الدوال.

لنأخذ مثلًا دالة `loadScript(src, callback)` من الفصل مقدمة إلى ردود النداء callback:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for $
{src}`));

  document.head.append(script);
}

// usage:
// loadScript('path/script.js', (err, script) => {...})
```

هيا نحوّلها. على الدالة الجديدة `loadScriptPromise(src)` القيام بنفس ما تقوم به تلك، ولكن لا تقبل إلّا `src` وسيطًا (بدون `callback`) وتُعيد وعدًا.

```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err)
      else resolve(script);
    });
  })
}

// usage:
// loadScriptPromise('path/script.js').then(...)
```

الآن يمكننا استعمال `loadScriptPromise` بسهولة بالغة في الشيفرات التي تعتمد الوعود.

وكما نرى فالدالة تكلّف الدالة الأصلية `loadScript` بكلّ العمل اللازم، ومـا تفعلـه هـو تقـديم ردّ نـداء من عندها تحوّله إلى وعد `resolve/reject`.

نحتاج عمليًا إلى تحويل دوال عديدة وكثيرة لتعتمد الوعود، لذا من المنطقي استعمال دالة مساعِدة. لنسمّها `promisify(f)` وستقبل دالة الأصل `f` اللازم تحويلها، وتُعيد دالة غالِفة.

يؤدّي الغلاف نفس عمل الشيفرة أعلاه: يُعيد وعدًا ويمرّر النداء إلى الدالة الأصلية `f` متتّبعًا الناتج في ردّ نداء يصنعه بنفسه:

```
function promisify(f) {
  return function (...args) { // يعيد التابع المغلّف
    return new Promise((resolve, reject) => {
      function callback(err, result) { // ردّ النداء خاصتنا لِـ f
        if (err) {
          return reject(err);
        } else {
          resolve(result);
        }
      }

      args.push(callback); // نضيف ردّ النداء خاصتنا إلى نهاية وسطاء f

      f.call(this, ...args); // استدعي التابع الأصلي
    });
  };
};

// طريقة الاستخدام:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);
```

نفترض هنا بأنّ الدالة الأصلية تتوقّع استلام ردّ نداء له وسيطين `(err, result)`، وهذا ما نواجهه أغلب الوقت لهذا كتبنا ردّ النداء المخصّص بهذا التنسيق، وتعمل دالة `promisify` على أكمل وجه... لهذه الحالات.

ولكن ماذا لو كانت تتوقّع `f` ردّ نداء له وسطاء أكثر `callback(err, res1, res2, ...)`؟

إليك نسخة ذكية محسّنة من `promisify`: لـو اسـتدعيناها هكـذا `promisify(f, true)` فسـيكون نـاتج الوعد مصفوفة من نواتج ردود النداء `[res1, res2, ...]`:

```
// التابع promisify(f, true) لجب مصفوفة النتائج
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, ...results) { // رد النداء خاصتنا لِ f
        if (err) {
          return reject(err);
        } else {
          // إجلب جميع ردود النداء وإذا حدّد أكثر من وسيط
          resolve(manyArgs ? results : results[0]);
        }
      }

      args.push(callback);

      f.call(this, ...args);
    });
  };
};

// طريقة الاستخدام:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...)
```

أمّا لتنسيقات ردود النداء الشاذّة (مثل التي بدون وسيط `err` أصلًا `callback(result)`) فيمكننا تحويلها يدويًا بدون استعمال الدالة المساعِدة.

كما وهناك وحدات لها دوال تحويل مرنة أكثر في التعامل مثل es6-promisify. وفي Node.js نرى الدالة المضمّنة `util.promisify` لهذا الغرض.

> يعدّ تحويل الدوال إلى وعود نهجًا رائعًا، خاصةً عند استخدام `async/await` (الّتي سنراها في الفصل التالي)، ولكن ليس بديلًا كليًا لردود النداء.
> تذكر أن الوعد له نتيجة واحدة فقط، ولكن تقنيًا ممكن أن تُستدعى ردود النداء عدة مرات. لذا فإن تحويل الدوال إلى وعود مخصصة للدوال التي تستدعي ردود النداء لمرة واحدة. وستُتجاهل جميع الاستدعاءات اللاحقة.

11.7 المهام السريعة Microtasks مقابل الوعد في تنفيذ المهام لاحقًا

دوال المُعاملة للوعود `.then` و `.catch` و`.finally` هي دوال غير متزامنة، دومًا. فحتّى لو سويَ الوعد مباشرةً (أي سواءً أنُجز أو رُفض) فالشيفرة أسفل `.then` و `.catch` و`.finally` ستُنفّذ حتّى قبل دوال المعاملة. لاحِظ:

```
let promise = Promise.resolve();

promise.then(() => alert("promise done!")) // اكتمل الوعد

alert("code finished"); // نرى هذا النص أولًا (انتهت الشيفرة)
```

لو شغّلته فسترى أولًا `code finished` وبعدها ترى `promise done!`. هذا... غريب إذ أن الوعد أنجز قطعًا في البداية. لماذا شُغّلت `.then` بعدئذ؟ ما الّذي يحدث؟

11.7.1 طابور المهام السريعة

تطلب الدوال غير المتزامنة عملية إدارة مضبوطة. ولهذا تحدّد مواصفة اللغة طابورًا داخليًا باسم `PromiseJobs` (غالبًا ما نسمّيه "بطابور المهام السريعة" Microtask Queue حسب مصطلح محرّك V8).

تقول المواصفة:

- الطابور بمبدأ "أوّل من يدخل هو أوّل من يخرج": المهام التي تُضاف أولًا تُنفّذ أولًا.
- لا يبدأ تنفيذ المهمة إلّا لو لم يكن هناك شيء آخر يعمل.

وبعبارة أبسط، فمتى جهز الوعد تُضاف دوال المعاملة `.then/catch/finally` إلى الطابور، وتبقى هناك بلا تنفيذ. متى وجد محرّك جافاسكربت نفسه قد فرغ من الشيفرة الحالية، يأخذ مهمة من الطابور وينفّذها. لهذا السبب نرى "اكتملت الشيفرة" في المثال أعلاه أولًا.

promise . then (handler);
...
alert ("code finished");
انتهى تنفيذالشيفرة
يبدأ معالج طابور المهام
معالجة المهام المنتظرة
في الطابور

دوال معاملة الوعود تمرّ من الطابور الداخلي هذا دومًا.

لو كانت في الشيفرة سلسلة من `then/catch/finally.` فستُنفّذ كلّ واحدة منها تنفيذًا غـير مـتزامن. أي أنّ الأولى تُضاف إلى الطابور وتُنفّذ متى اكتمل تنفيذ الشيفرة الحاليـة وانتهت دوال المُعاملـة الّـتي أُضـيفت إلى الطابور مسبقًا.

ولكن ماذا لو كان الترتيب يهمّنا؟ كيف نشغّل `code finished` بعد `promise done`؟ بسيطة، نضعها في الطابور باستعمال `then.`:

```
Promise.resolve()
  .then(() => alert("promise done!"))
  .then(() => alert("code finished"));
```

هكذا صار الترتيب كما نريد.

11.7.2 الرفض غير المعالج

تذكر حدث `unhandledrejection` من فصل التعامل مع الأخطاء في الوعود.

سنرى الآن كيف تعرف محرّكات جافاسكربت ما إن وُجدت حالة رفض لم يُتعامل معها، أم لا. تحدث "حالـة الرفض لم يُتعامل معها" حين لا يتعامل شيء مع خطأ أنتجه وعد في آخر طابور المهام السريعة.

عادةً لو كنّا نتوقّع حدوث خطأ نُضيف التابع `catch.` إلى سلسلة الوعود للتعامل معه:

```
let promise = Promise.reject(new Error("Promise Failed!"));
promise.catch(err => alert('caught')); // هكذا

// لا يعمل هذا السطر إذ تعاملنا مع الخطأ
window.addEventListener('unhandledrejection', event =>
alert(event.reason));
```

ولكن... لو نسينا وضع `catch.` سيُشغّل المحرّك هذا الحدث متى فرغ طابور المهام السريعة:

```
let promise = Promise.reject(new Error("Promise Failed!"));

// Promise Failed!
window.addEventListener('unhandledrejection', event =>
alert(event.reason));
```

وماذا لو تعاملنا مع الخطأ لاحقًا؟ هكذا مثلًا:

```
let promise = Promise.reject(new Error("Promise Failed!"));
setTimeout(() => promise.catch(err => alert('caught')), 1000); // لاحِظ
```

```
// Error: Promise Failed!
window.addEventListener('unhandledrejection', event =>
alert(event.reason));
```

إذا شغلناه الآن سنرى `Promise Failed!` أولًا ثم `caught`.

لو بقينا نجهل طريقة عمل طابور المهام السريعة فسنتساءل: "لماذا عملت دالة المُعاملة `unhandledrejection`؟ الخطأ والتقطناه!".

أمّا الآن فنعرف أنّه لا يُولّد `unhandledrejection` إلّا حين انتهاء طابور المهام السريعة: فيفحص المحرّك الوعود وإن وجد حالة "رفض" في واحدة، يشغّل الحدث.

في المثال أعلاه، أضيفت `.catch` وشغّلت من قِبل `setTimeout` متأخرةً بعد حدوث `unhandledrejection` لذا فإن ذلك لم يغيّر أي شيء.

11.7.3 الخلاصة

التعامل مع الوعود دومًا يكون غير متزامن، إذ تمرّ إجراءات الوعود في طابور داخلي "لمهام الوعود" أو ما نسمّيه "بطابور المهام السريعة" (مصطلح المحرّك V8).

بهذا لا تُستدعى دوال المُعاملة `.then/catch/finally` إلّا بعد اكتمال الشيفرة الحالية.

ولو أردنا أن نضمن تشغيل هذه الأسطر بعينها بعد `.then/catch/finally` فيمكننا إضافتها إلى استدعاء `.then` في السلسلة.

في معظم محركات جافاسكربت، بما في ذلك المتصفحات و Node.js، يرتبط مفهوم المهام السريعة ارتباطًا وثيقًا بـ "حلقة الأحداث" والمهام الكبيرة "macrotasks". نظرًا لأنها لا تملك علاقة مباشرة بالوعود، لذا فإننا شرحناها في الجزء الثاني من هذا الكتاب.

11.8 اللاتزامن والانتظار async/await

توجد صياغة أخرى مميّزة للعمل مع الوعـود بنحـوٍ أكـثر سـهولة تُـدعى `async/await`. فهمهـا أسـهل من شرب الماء واستعمالها

11.8.1 الدوال غير المتزامنة

فلنبدأ أولًا بكلمة `async` المفتاحية. يمكننا وضعها قبل الدوال هكذا:

```
async function f() {
  return 1;
}
```

وجود الكلمة "async" قبل (اختصار "غير متزامنة" بالإنجليزية) يعني أمرًا واحدًا: تُعيد الدالة وعدًا دومًا.

فمثلًا تُعيد هذه الدالة وعدًا مُنجز فيه ناتج `1`. فلنرى:

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

كما يمكننا أيضًا إعادة وعد صراحةً:

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

هكذا تضمن لنا `async` بأنّ الدالة ستُعيد وعدًا وستُغلّف الأشياء التي ليست بوعـود وعـودًا. بسـيطة صـح؟ ليس هذا فحسب، بل هناك أيضًا الكلمة المفتاحية `await` التي تعمل فقط في الدوال غـير المتزامنـة `async`، وهي الأخرى جميلة.

11.8.2 Await

الصياغة:

```
// لا تعمل إلّا في الدوال غير المتزامنة
let value = await promise;
```

الكلمة المفتاحية تجعل لغة جافاسكربت تنتظر حتى يُنجز الوعد ويعيد نتيجة.

إليك مثالًا لوعد نُفذّ خلال ثانية واحدة:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000) // تم!
  });

  let result = await promise; // ننتظر ... الوعد (*)
  alert(result); // "done!" تم!
}
f();
```

يتوقّف تنفيذ الدالة "قليلًا" عند السطر `(*)` ويتواصل متى ما أُنجز الوعد وصار `result` ناتجه. الشيفرة أعلاه تعرض "تم!" بعد ثانية واحدة.

لنوضح أمرًا مهمًا: تجعل `await` لغة جافاسكربت تنتظر حتى يُنجز الوعد، وبعدها تذهب مع النتيجة. هذا لن يكلفنا أي موارد من المعالج لأن المحرك مُنشغل بمهام أخرى في الوقت نفسه، مثل: تنفيذ سكربتات أُخرى، التعامل مع الأحداث ..إلخ.

هي مجرد صياغة أنيقة أكثر من صياغة `promise.then` للحصول على ناتج الوعد. كما أنها أسهل للقراءة والكتابة.

> لا يمكننا استخدام `await` في الدوال العادية. فإذا حاولنا استخدام الكلمة المفتاحية `await` في الدوال العادية فسيظهر خطأ في في الصياغة:
>
> ```
> function f() {
>   let promise = Promise.resolve(1);
>   let result = await promise; // Syntax error
> }
> ```
>
> سنحصل على هذا الخطأ إذا لم نضع `async` قبل الدالّة. كما قلنا `await` تعمل فقط في الدوال غير المتزامنة.

لنأخذ مثال `showAvatar()` من الفصل سَلسلة الوعود ونُعيد كتابته باستعمال `async/await`:

1. علينا استبدال استدعاءات `.then` ووضع `await`.
2. علينا أيضًا تحويل الدالة لتكون غير متزامنة `async` كي تعمل `await`.

```
async function showAvatar() {
  // JSON اقرأ ملفات
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // github اقرأ مستخدم
  let githubResponse = await fetch(`https://api.github.com/users/$
{user.name}`);
  let githubUser = await githubResponse.json();

  // avatar أظهر الصورة الرمزية
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // انتظر 3 ثواني
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));
  img.remove();
  return githubUser;
}
showAvatar();
```

شيفرة نظيفة وسهلة القراءة! أفضل من السابقة بفراسخ.

ا. لن تعمل await في الشيفرة ذات المستوى الأعلى

يميل المبتدئين إلى نسيان أن `await` لن تعمل في الشيفرة البرمجية ذات هرمية أعلى(ذات المستوى الأعلى). فمثلًا لن يعمل هذا:

```
// خطأ صياغيّ في الشيفرة عالية المستوى
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();
```

يمكننا تغليفها بداخل دالّة متزامنة مجهولة، هكذا:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

ب. إن await تقبل then

تسمح await باستخدام كائنات thenable (تلك التي تستخدم دالة then القابلة للاستدعاء) تمامًا مثل promise.then. الفكرة أنه يمكن ألا يكون الكائن الخارجي وعدًا ولكنه متوافق مع الوعد: إن كان يدعم then.، وهذا يكفي لاستخدامه مع await.

هنا مثال لاستخدام صنف Thenable والكلمة المفتاحية await أدناه ستقبلُ حالاتها:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve);
    // نفذ عملية this.num*2 بعد ثانية واحدة
    setTimeout(() => resolve(this.num * 2), 1000); // (*)
  }
};

async function f() {
  // انتظر لثانية واحدة. ثم النتيجة ستصبح 2
  let result = await new Thenable(1);
  alert(result);
}

f();
```

إذا حصَلت await على كائن ليس وعدًا مع then. فإنه يستدعي الدوالّ المضمنة في اللغة مثل: resolve و reject كوسطاء (كما يفعل المنفذّ للوعد العادي تمامًا). وثم تنتظر await حتى يستدعى أحدهم (في المثال أعلاه في السطر (*)) ثم يواصل مع النتيجة.

ج. توابع صنف غير متزامنة

لتعريف دالّة صنف غير متزامنة، أضفها مع الكلمة async:

```
class Waiter {
  async wait() {
    return await Promise.resolve(1);
  }
}

new Waiter()
  .wait()
  .then(alert); // 1
```

لاحظ أن المعنى هو نفسه: فهو يضمن أن القيمة المرتجعة هي وعد وawait مفعّلة أيضًا.

11.8.3 التعامل مع الأخطاء

لو نُفذّ الوعد بنجاح فسيُعيد promise await الناتج، ولكن لو حصلت حالة رفض فسترمي الخطأ كما لو كانت إفادة throw مكتوبة.

إليك الشيفرة:

```
async function f() {
  await Promise.reject(new Error("Whoops!")); // لاحظ
}
```

و... هي ذاتها هذه:

```
async function f() {
  throw new Error("Whoops!"); // هنا
}
```

في الواقع تأخذ الوعود وقتًا قبل أن تُرفض. في تلك لحالة فستكون هناك مهلة قبل أن يرمي await الخطأ.

يمكننا التقاط ذاك الخطأ باستعمال try..catch كما استعملنا throw:

```
async function f() {
```

```
  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch خطأ في النوع: فشل الجلب
  }
}

f();
```

لو حدث خطأ فينتقل سير التنفيذ إلى كتلة `catch`. يمكننا أيضًا وضع أكثر من سطر واحد:

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // تلتقط أخطاء fetch وresponse.json معًا
    alert(err);
  }
}

f();
```

لو لم نضع `try..catch` فستكون حالة الوعد الذي نفّذه استدعاء الدالة غير المتزامنة `f()` - يكون بحالة رفض. يمكننا هنا استعمال `.catch` للتعامل معه:

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// يصير استدعاء f() وعدًا مرفوضًا
f().catch(alert); // TypeError: failed to fetch خطأ في النوع: فشل الجلب //
(*)
```

لو نسينا هنا إضافة `.catch` فستتلقّى خطأ وعود لم نتعامل معه (يمكن أن نراه من الطرفية). يمكننا أيضًا استلام هذه الأخطاء باستعمال دالة مُعاملة الأحداث العمومية كما وضّحنا في الفصل التعامل مع الأخطاء في الوعود.

ا. async/await و promise.then/catch

عندما نستخدم `async/await` نادرًا ما نحتاج إلى `.then` وذلك لأن `await` تعالج عملية الانتظار. ويمكننا استخدام `try..catch` بدلًا من `.catch`. وهذه عادةً (ليس دائمًا) ما تكون أكثر ملاءمة.

ولكن في الشيفرات ذات هرمية أعلى (مستوى أعلى)، عندما نكون خارج أي دالّة غير متزامنة، يتعذر علينا استخدام `await` لذا من المعتاد إضافة `.then/catch` لمعالجة النتيجة النهائية أو الأخطاء المتساقطة.

كما هو الحال في السطر `(*)` من المثال أعلاه.

ب. تعمل async/await مع Promise.all

عندما نحتاج لانتظار عدة وعود يمكننا أن نغلفها في تابع `Promise.all` وبعده `await`:

```
// انتظر مصفوفة النتائج
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```

وإن حدث خطأ ما، فإنه سيُنقل من الوعد نفسه إلى التابع `Promise.all`، وبعدها يصبح استثناءً يمكننا التقاطه باستخدام `try..catch` حول الاستدعاء.

11.8.4 الخلاصة

لكلمة `async` المفتاحية قبل الدوال تأثيرين اثنين:

1. تحوّلها لتُعيد وعدًا دومًا.
2. تتيح استعمال `await` فيها.

حين يرى محرّك جافاسكربت الكلمة المفتاحية `await` قبل الوعود، ينتظر حتّى يُنجز الوعد ومن ثمّ:

1. لو كان خطأ فسيُولِّد الاستثناء كما لو استعملنا `throw error`.
2. وإلّا أعاد الناتج.

تقدّم لنا هتين الكلمتين معًا إطار عمل رائع نكتب به شيفرات غير متزامنة تسهل علينا قراءتها كما وكتابتها.

نادرًا ما نستعمل `promise.then/catch` بوجود `async/await`، ولكن علينا ألّا ننسى بأنّ الأخيرتين مبنيّتين على الوعود إذ نضطر أحيانًا (خارج الدوال مثلًا) استعمال `promise.then/catch`. كما وأنّ `Promise.all` جميل جدًا لننتظر أكثر من مهمّة في وقت واحد.

11.8.5 تمارين

ا. إعادة الكتابة باستعمال async/await

أعِد كتابة الشيفرة في المثال من الفصل سلسلة الوعود باستعمال `async/await` بدل `.then/catch`:

```
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
    })
}

loadJson('no-such-user.json') // (3)
  .catch(alert); // Error: 404
```

الحل:

ترى الملاحظات أسفل الشيفرة:

```
async function loadJson(url) { // (1)
  let response = await fetch(url); // (2)
  if (response.status == 200) {
    let json = await response.json(); // (3)
    return json;
  }
  throw new Error(response.status);
}
loadJson('no-such-user.json')
  .catch(alert); // Error: 404 (4)
```

ملاحظات:

1. الدالّة `loadJson` تصبح `async`.
2. جميع المحتوى في `.then` يستبدل بِـ `await`.
3. يمكننا إعادة `return response.json()` بدلًا من انتظارها. هكذا:

```
if (response.status == 200) {
  return response.json(); // (3)
}
```

ثم الشيفرة الخارجية ستنتظر `await` لينفذ الوعد في حالتنا الأمر غير مهم.

4. سيرمى الخطأ من التابع `loadJson` المعالج من قبل `catch`.. لن نستطيع استخدام `await loadJson(…)` هنا، وذلك لأننا لسنا في دالّة غير متزامنة.

ب. أعِد كتابة rethrow باستعمال async/await

في الشيفرة أدناه مثلًا عن إعادة الرمي من فصل سلسلة الوعود. أعد كتابته باستخدام `async/await` بدلًا من `then/catch`..

وتخلص من العودية لصالح الحلقة في `demoGithubUser`: مع استخدام `async/await` سيسهلُ الأمر.

```
class HttpError extends Error {
  constructor(response) {
    super(`${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new HttpError(response);
      }
    })
```

```
}

// اطلب اسم المستخدم إلى أن يعيد لك github مستخدم صحيح
function demoGithubUser() {
  let name = prompt("Enter a name?", "iliakan");

  return loadJson(`https://api.github.com/users/${name}`)
    .then(user => {
      alert(`Full name: ${user.name}.`);
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status == 404) {
        alert("No such user, please reenter.");
        return demoGithubUser();
      } else {
        throw err;
      }
    });
}

demoGithubUser();
```

الحل:

لا يوجد أي صعوبة هنا، إذ كل ما عليك فعله هو استبدال `try...catch` بدلًا من `.catch` بداخل تابع `demoGithubUser` وأضف `async/await` عند الحاجة:

```
class HttpError extends Error {
  constructor(response) {
    super(`${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

async function loadJson(url) {
```

```
  let response = await fetch(url);
  if (response.status == 200) {
    return response.json();
  } else {
    throw new HttpError(response);
  }
}
// اطلب اسم المستخدم إلى أن يعيد لك github مستخدم صحيح
async function demoGithubUser() {

  let user;
  while(true) {
    let name = prompt("Enter a name?", "iliakan");

    try {
      user = await loadJson(`https://api.github.com/users/${name}`);
      break; // لا يوجد خطأ اخرج من الحلقة
    } catch(err) {
      if (err instanceof HttpError && err.response.status == 404) {
        // تستمر الحلقة بعد alert
        alert("No such user, please reenter.");
      } else {
        // خطأ غير معروف , أعد رميه rethrow
        throw err;
      }
    }
  }
  alert(`Full name: ${user.name}.`);
  return user;
}
demoGithubUser();
```

ج. استدعاء async من دالة غير متزامنة

لدينا دالة "عادية"، ونريد استدعاء async منها واستعمال ناتجها، كيف؟

```
async function wait() {
  await new Promise(resolve => setTimeout(resolve, 1000));

  return 10;
}

function f() {
  // ماذا نكتب هنا؟...
  // علينا استدعاء async wait() والانتظار حتى تأتي 10
  // لا تنسَ، لا يمكن استعمال "await" هنا
}
```

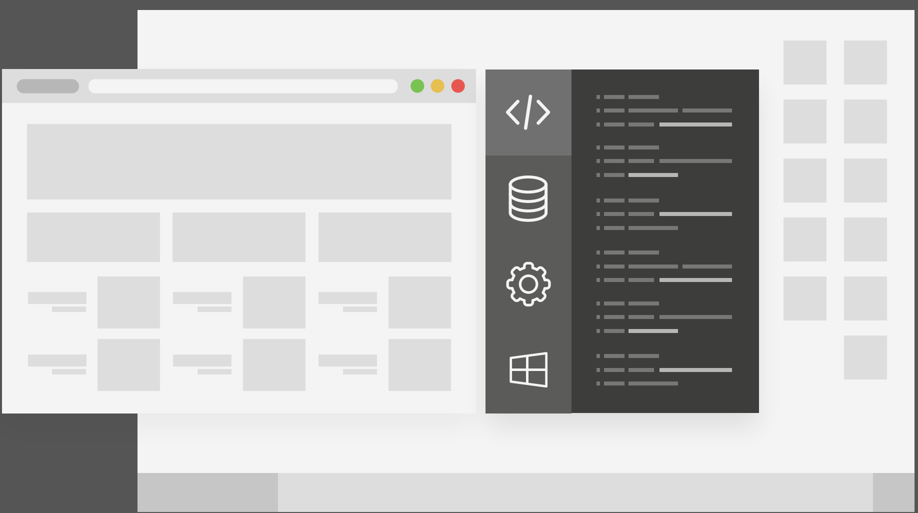
ملاحظة: هذه المهمّة (تقنيًا) بسيطة جدًا، ولكنّ السؤال شائع بين عموم المطوّرين الجدد على صيغة async/await.

الحل:

هنا تأتي فائدة معرفة طريقة عمل هذه الأمور. ليس عليك إلّا معاملة استدعاء async وكأنّه وعد ووضع تابِع then. :

```
async function wait() {
  await new Promise(resolve => setTimeout(resolve, 1000));
  return 10;
}

function f() {
  // يعرض 10 بعد ثانية واحدة
  wait().then(result => alert(result));
}

f();
```

دورة علوم الحاسوب



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

12. المولدات والمكررات المتقدمة

يتضمن هذا الفصل الأقسام التالية:

1. المولدات Generators
2. المكررات والمولدات غير المتزامنة

12.1 المولدات Generators

تُعيد الدوال العادية قيمة واحدة فقط لا غير (أو لا تُعيد شيئًا). بينما يمكن للمولّدات إعـادة (أو إنتـاج yeild) أكـثر من قيمـة واحـدةً بعـد الأخـرى حسـب الطلب. تعمـل المولّـدات عملًا جميلًا جـدًا مـع الكائنـات المكـرَّرة (Iterables) في جافاسكربت وتتيح لنا إنشاء سيول البيانات بسهولة بالغة.

12.1.1 الدوال المولدة

لإنشاء مولّد علينا استعمال صياغة مميّزة: `function*` أو ما يسمّونه "الدالة المولِّدة". هذا شكلها:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

يختلف سلوك الدوال المولِّدة عن تلك العادية، فحين تُستدعى الدالة لا تُشغّل الشيفرة فيها، بل تعيـد كائنًـا مميزًا نسمّيه "كائن المولّد" ليُدير عملية التنفيذ. خُذ نظرة:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
// تنشئ الدالة المولِّدة كائن مولَد
let generator = generateSequence();

alert(generator); // [object Generator]
```

الشيفرة الموجودة بداخل الدالّة لم تُنفذ بعد:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

التابع الأساسي للمولّد هو `next()`. متى استدعيناه بدأ عملية التنفيذ حتّى يصل أقرب جملة yield <value> (يمكن ألّا نكتب `value` وستصير القيمة `undefined`). بعدها يتوقّف تنفيذ الدالة مؤقّتًا وتُعاد القيمة `value` إلى الشيفرة الخارجية.

ناتج التابع `next()` لا يكون إلّا كائنًا له خاصيتين:

- `value`: القيمة التي أنتجها المولّد.
- `done`: القيمة `true` لو اكتملت شيفرة الدالة، وإلّا `false`.

فمثلًا هنا نُنشِئ مولّدًا ونأخذ أوّل قيمة أنتجها:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

let one = generator.next(); // هنا

alert(JSON.stringify(one)); // {value: 1, done: false}
```

حاليًا أخذنا القيمة الأولى فقط، وسير تنفيذ الدالة موجود في السطر الثاني:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

{value: 1, done: false}

فلنستدعِ `generator.next()` ثانيةً الآن. سنراه واصل تنفيذ الشيفرة وأعاد القيمة المُنتَجة `yield` التالية:

```
let two = generator.next();

alert(JSON.stringify(two)); // {value: 2, done: false}
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

{value: 2, done: false}

والآن إن شغّلناه مرّة ثالثة سيصل سير التنفيذ إلى عبارة `return` ويُنهي الدالة:

```
let three = generator.next();

alert(JSON.stringify(three)); // {value: 3, done: true} // لاحِظ قيمة done
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

{value: 3, done: true}

الآن اكتمل المولّد بقيمة `value:3` ويمكننا الآن معالجتها. عرفنا ذلك من `done:true`.

الاستدعاءات اللاحقة على `generator.next()` لن تكون منطقية الآن. ولو حصلت فستُعيد الدالة الكائن نفسه: `{done: true}`.

الصياغة function* f(…) أم function *f(…)؟

في الحقيقة كِلاهما صحيح. ولكن عادةً تكون الصياغة الأولى مفضلة أكثر من الثانية. وتشير النجمة * على أنها دالة مولّد، إذ تصف النوع وليس الاسم، لذلك يجب أن ندمج الكلمة المفتاحية `function` بالنجمة.

12.1.2 المولدات قابلة للتكرار

نفترض أنّك توقّعت ذلك حين رأيت التابع `next()`، إذ أن المولدات قابلة للتكرار iterable، فيمكننا المرور على عناصرها عبر `for..of`:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
```

```
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1 ثم 2
}
```

هذا أجمل من استدعاء `.next().value`، أم لا؟ ولكن... لاحِظ: يعرض المثال أعلاه 1 ثمّ 2 وفقط. لن يعرض 3 مطلقًا! هذا لأنّ عملية التكرار لِـ `for..of` تتجاهل قيمة `value` الأخيرة حين تكون `done: true`. لذا لو أردنا أن تظهر النتائج كلّها لعملية تكرار `for..of`، فعلينا إعادتها باستعمال `yield`:

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3; // هكذا
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1 ثمّ 2 ثمّ 3
}
```

نظرًا من كون المولّدات قابلة للتكرار، يمكننا استدعاء جميع الدوالّ المتعلّقة بذلك، مثل: معامل "البقية" `...`:

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}

let sequence = [0, ...generateSequence()];

alert(sequence); // 0, 1, 2, 3
```

يحول التابع `generateSequence()...` في الشيفرة أعلاه كائن المولد القابل للتكـرار إلى مصـفوفة من العناصر (لمزيد من المعلومات أحيلك إلى هذا الفصل "المُعاملات "البقية" ومُعامل التوزيع").

12.1.3 استعمال المولّدات على أنّها مكرّرات

سابقًا في فصل "المُعاملات "البقية" ومُعامل التوزيع" أنشأنا كائن `range` يمكن تكـراره والـذي يعيـد القيم بين قيميتين `from..to`. لنتذكّر الشيفرة معًا:

```
let range = {
  from: 1,
  to: 5,
  // تستدعي حلقة for..of هذه الدالّة مرة واحدة في البداية
  [Symbol.iterator]() {
    // ...ستعيد كائن مُكرَر
    // لاحقًا ستعمل حلقة for..of مع ذلك الكائن وتطلب منه القيم التالية
    return {
      current: this.from,
      last: this.to,
      // تستدعى next() في كلّ تكرار من خلال الحلقة for..of
      next() {
        // يجب أن تعيد القيم ككائن {done:.., value :...}
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }
};
// عملية التكرار تمرُّ عبر range من range.from إلى range.to
alert([...range]); // 1,2,3,4,5
```

يمكننا استعمال دالة المولِّدة كمكرّرات من خلال `Symbol.iterator`.

إليك نفس الكائن `range`, ولكن بطريقة أكثر إيجازًا:

```
let range = {
  from: 1,
  to: 5,

  *[Symbol.iterator]() { // [Symbol.iterator]: function*() اختصارًا لِ
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};
alert( [...range] ); // 1,2,3,4,5
```

الشيفرة تعمل إذ يُعيد `range[Symbol.iterator]()` الآن مولّدًا، وما تتوقّعه `for..of` هي توابِع تلك المولّدات بعينها:

- إذ لها التابع `.next()`
- وتُعيد القيم على النحو الآتي `{value: ..., done: true/false}`

بالطبع هذه ليست مصادفة أضيفت المولّدات إلى لغة جافاسكربت مع الأخذ بعين الاعتبار للمكرّرات لتنفيذهم بسهولة.

إن التنوع في المولدات أعطى شيفرة موجزة أكثر الشيفرة الأصلية لكائن `range`، وجميعهم لديهم نفس الخصائص الوظيفية.

المولّدات يمكن أن تولد قيمًا للأبد

في الأمثلة أعلاه أنشأنا متتالية منتهية من القيم، ولكن باستخدام المولد نستطيع أن ننتج قيمًا إلى الأبد. على سبيل المثال لننشئ متتالية غير منتهية من الأرقام العشوائية الزائفة.

ومن المؤكد أن هذا المولّد سيحتاجُ إلى طريقة لإيقافه مثل: `break` (أو `return`) في حلقة `for..of`. وإلا فإن الحلقة ستستمر إلى الأبد.

12.1.4 تراكب المولّدات

تراكب المولّدات (Generator composition) هي ميزة خاصّة للمولّدات تتيح لها "تضمين" المولّدات الأخرى فيها دون عناء.فمثلًا لدينا هذه الدالة التي تُولّد سلسلة من الأعداد:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
```

```
}
```

نريد الآن إعادة استعمالها لتوليد سلسلة أعداد معقّدة أكثر من هذه أعلاه:

- أولًا الأرقام `0..9` (ورموز المحارف اليونيكوديّة هو من 48 إلى 57)
- ثمّ الأحرف الأبجدية الإنجليزية بالحالة الكبيرة `A..Z` (ورموزها من 65 إلى 90)
- ثمّ الأحرف الأبجدية الإنجليزية بالحالة الصغيرة `a..z` (ورموزها من 97 إلى 122)

يمكننا استعمال هذه السلسلة لإنشاء كلمات السرّ باختيار المحارف منها مثلًا (ويمكننا إضافة المحارف الخاصّة أيضًا). ولكن لذلك علينا توليدها أولًا.

علينا في الدوال العادية (لتضمين النواتج من دوالّ أخرى) استدعاء تلك الدوال وتخزين نواتجها ومن ثمّ ربطها في نهاية الدالة الأمّ.

وفي المولّدات نستعمل الصياغة المميّزة `yield*` لتضمين (أو تركيب) مولّدين داخل بعض. المولّد المركّب:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generatePasswordCodes() {

  // 0..9
  yield* generateSequence(48, 57);

  // A..Z
  yield* generateSequence(65, 90);

  // a..z
  yield* generateSequence(97, 122);

}

let str = '';

for(let code of generatePasswordCodes()) {
```

```
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```

توجه `yield*` التنفيذ إلى مولّد آخر. مما يعني أن `gen` `yield*` ستكرر عبر المولّد `gen` وستُعيد نتائجه للخارج. أي كما لو أنتجت هذه القيم بمولّد خارجي.

النتيجة نفسها كما لو أننا ضمنًا الشيفرة من المولّدات المتداخلة:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generateAlphaNum() {


  // yield* generateSequence(48, 57);
  for (let i = 48; i <= 57; i++) yield i;

  // yield* generateSequence(65, 90);
  for (let i = 65; i <= 90; i++) yield i;

  // yield* generateSequence(97, 122);
  for (let i = 97; i <= 122; i++) yield i;


}

let str = '';

for(let code of generateAlphaNum()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```

تراكب المولّدات هو فعلًا طريقة بديهيّة لإدخال أكثر من مولّد في مولّـد واحـد، ولا تسـتعمل ذاكـرة إضـافية لتخزين النواتج البينيّة.

12.1.5 عبارة "yield" باتجاهين اثنين

إلى هنا نرى المولّدات تشبه كائنات المكرّرات كثـيرًا، فقـط أنّ لهـا صـياغة مميّـزة لتوليـد القيم. ولكن وعلى أرض الواقع، المولّدات أكثر مرونة وفاعليّة. إذ أنّ `yield` تعمل بـالمجيء وبالإيـاب: فلا تُعيـد النـاتج إلى الخـارج فحسب بل أيضًا يمكنها تمرير القيمة إلى داخل المولّد.

لذلك علينا استدعاء `generator.next(arg)` بتمرير وسيط واحد. هـذا الوسـيط سـيكون نـاتج `yield`. الأفضل لو نرى مثالًا:

```
function* gen() {
  // نمرّر سؤالًا إلى الشيفرة الخارجية وننتظر إجابةً عليه
  let result = yield "2 + 2 = ?"; // (*)

  alert(result);
}

let generator = gen();

let question = generator.next().value; // <-- تُعيد yield القيمة

generator.next(4); // --> نمرّر القيمة إلى المولّد
```

Generator

Calling code

```
function* gen() {
  let result = yield "2 + 2 = ?";

  alert(result); // 4
}
```

question = "2 + 2 = ?"

.next(4)

1. يكون الاستدعاء الأوّل للتابِع `generator.next()` دومًا دون تمرير أيّ وسيط. يبدأ الاستدعاء التنفيـذَ ويُعيد ناتج أوّل `yield "2+2=?"`. هنا يُوقف المولّد التنفيذ مؤقّتًا (على ذلك السطر).

2. ثمّ (كما نرى في الصورة) يُوضع ناتج `yield` في متغير السؤال `question` في الشيفرة التي استدعت المولّد.

3. وعند `generator.next(4)` يُواصل المولّد عمله ويستلم 4 ناتجًا: `let result = 4`.

لاحِظ أنّه ليس على الشيفرة الخارجية استدعاء `next(4)` مباشرةً وفي الحال، بل يمكن أن تأخذ الوقت الذي تريد. سيبقى المولّد منتظرًا ولن تكون مشكلة. مثال:

```
// تُواصل عمل المولّد بعد زمن معيّن
setTimeout(() => generator.next(4), 1000);
```

كما نرى فعلى العكس تمامًا من الدوال العادية، يمكن للمولّد ولشيفرة الاستدعاء تبادل النتائج بتمرير القيم إلى `next/yield`. ليتوضّح هذا أكثر سنرى مثالًا آخر فيه استدعاءات أكثر:

```
function* gen() {
  let ask1 = yield "2 + 2 = ?";

  alert(ask1); // 4

  let ask2 = yield "3 * 3 = ?"

  alert(ask2); // 9
}
let generator = gen();

alert( generator.next().value ); // "2 + 2 = ?"
alert( generator.next(4).value ); // "3 * 3 = ?"
alert( generator.next(9).done ); // true
```

إليك صورة سير التنفيذ:

Generator		Calling code

```
function* gen() {
  let ask1 = yield "2 + 2 = ?";

  alert(ask1); // 4

  let ask2 = yield "3 * 3 = ?"

  alert(ask2); // 9
}
```

"2 + 2 = ?"

"3 * 3 = ?"

1. استدعاء `.next()` الأوّل يبدأ التنفيذ، ويصل إلى أوّل عبارة `yield`.
2. يُعاد الناتج إلى الشيفرة الخارجية.
3. يمرّر استدعاء `.next(4)` الثاني القيمة 4 إلى المولّد ثانيةً على أنّها ناتج أوّل مُنتَج `yield`، ويُواصل التنفيذ.
4. يصل التنفيذ إلى عبارة `yield` الثانية، وتصير هي ناتج الاستدعاء.
5. يُمرّر `next(9)` الثالث القيمة 9 إلى المولّد على أنّها ناتج ثاني مُنتَج `yield` ويُواصل التنفيذ حتّى يصل نهاية الدالة، بذلك تكون `done: true`.

تشبه هذه لعبة تنس الطاولة، حيث يمرّر كلّ تابِع `next(value)` (باستثناء الأوّل طبعًا) القيمة إلى المولّد فتصير ناتج المُنتَج `yield` الحالي، ومن ثمّ

12.1.6 generator.throw

يمكن للشيفرات الخارجية تمرير القيم إلى المولّدات على أنّها نواتج `yield` (كما لاحظنا من الأمثلة أعلاه). ويمكنها أيضًا بدء (أو رمي) خطأ أيضًا. هذا طبيعي إذ الأخطاء هي نواتج، نوعًا ما.

علينا استدعاء التابع `generator.throw(err)` لتمرير الأخطاء إلى عبارة `yield`. في هذه الحال يُرمى الخطأ `err` عند السطر الذي فيه `yield`.

فمثلًا تؤدّي عبارة `"2 + 2 = ?"` هنا إلى خطأ:

```
function* gen() {
  try {
    let result = yield "2 + 2 = ?"; // (1)
```

```
    alert("The execution does not reach here, because the exception is thrown above");
  } catch(e) {
    alert(e); // أظهر الخطأ
  }
}

let generator = gen();
let question = generator.next().value;

generator.throw(new Error("The answer is not found in my database")); // (2)
```

الخطأ الذي رميَ في المولد عند السطر `(2)` يقودنا إلى الخطأ في السطر `(1)` مع `yield`. في المثـال أعلاه التقطت `try..catch` الخطأ وثمّ عرضته. لو لم نلتقطه سيكون مآله (مآل أيّ استثناء آخـر غـيره) أن "يسـقط" من المولّد إلى الشيفرة التي استدعت المولّد.

هل يمكننا التقاط الخطـأ في سـطر شـيفرة الاسـتدعاء الّـتي تحتـوي على `generator.throw`، (المشـار إليه `(2)`)، هكذا؟

```
function* generate() {
  let result = yield "2 + 2 = ?"; // خطأ في هذا السطر
}

let generator = generate();

let question = generator.next().value;


try {
  generator.throw(new Error("The answer is not found in my database"));
} catch(e) {
  alert(e); // عرض الخطأ
}
```

إن لم نلتقـط الخطـأ هنـا فعندئـذ وكمـا هي العـادة سـتنهار الشـيفرة الخارجيـة (إن كـانت موجـودة) وإذا لم يكتشف الخطأ أيضًا عندها سينهار السكربت بالكامل.

12.1.7 الخلاصة

- تُنشِئ الدوال المولِّدة `function* f(…) {…}` المولّدات.
- يوجد المُعامل `yield` داخل المولدات (فقط).
- تتبادل الشيفرة الخارجية مع المولدات النتائج من خلال استدعاءات `next/yield`.

نادرًا ما تستخدم المولّدات في الإصدار الحديث من جافاسكربت. لكن في بعض الأحيان نستخدمها لتبادل البيانات بين الشيفرة المستدعاة أثناء تنفيذ شيفرة وحيدة. وبالتأكيد إنها رائعة لتوليد أشياء قابلة للتكرار.

في الفصل التالي، سنتعرف على مولدات غير متزامنة، والّتي تُستخدم لقراءة تدفقات البيانات غير المتزامنة (على سبيل المثال ، نرى على الإنترنت خدمات كثيرة تقدّم لنا البيانات على صفحات paginated) في حلقات `for await ... of`.

في برمجة الوِب، غالبًا ما نعمل مع البيانات المتدفقة، لذا فهذه حالة استخدام أخرى مهمة جدًا.

12.1.8 تمارين

ا. مولّد أرقام شبه عشوائيّة

نواجه كثيرًا من الأحيان حاجة ماسّة إلى بيانات عشوائية. إحدى هذه الأحيان هي إجراء الاختبارات، فنحتاج إلى بيانات عشوائية كانت نصوص أو أعداد أو أيّ شيء آخر لاختبار الشيفرات والبنى البرمجية.

يمكننا في جافاسكربت استعمال `Math.random()`، ولكن لو حدث خطب ما فنودّ إعادة إجراء الاختبار باستعمال نفس البيانات بالضبط (كي نختبر هذه البيانات).

لهذا الغرض نستعمل ما يسمّى "بمولّدات الأعداد شبه العشوائية المزروعة". تأخذ هذه المولّدات "البذرة" والقيمة الأولى ومن ثمّ تولّد القيم اللاحقة باستعمال معادلة رياضية بحيث أنّ كلّ بذرة تُنتج نفس السلسلة دائمًا، وهكذا يمكننا ببساطة استنساخ التدفق بالكامل من خلال بذورها فقط.

إليك مثال عن هذه المعادلة التي تولّد قيمًا موزّعة توزيعًا

```
next = previous * 16807 % 2147483647
```

لو استعملنا 1 ....، فستكون القيم كالآتي:

1. 16807
2. 282475249
3. 1622650073

4. ...وهكذا...

مهمّة هذا التمرين هو إنشاء الدالـة المولِّدة `pseudoRandom(seed)` فتأخـذ البـذرة `seed` وتُنشِـئ مولّـدًا بالمعادلة أعلاه. طريقة الاستعمال:

```
let generator = pseudoRandom(1);
alert(generator.next().value); // 16807
alert(generator.next().value); // 282475249
alert(generator.next().value); // 1622650073
```

الحل:

```
function* pseudoRandom(seed) {
  let value = seed;

  while(true) {
    value = value * 16807 % 2147483647
    yield value;
  }

};

let generator = pseudoRandom(1);

alert(generator.next().value); // 16807
alert(generator.next().value); // 282475249
alert(generator.next().value); // 1622650073
```

لاحِظ أننا نستطيع تأدية ذات الأمر بالدوال العادية هكذا:

```
function pseudoRandom(seed) {
  let value = seed;

  return function() {
```

```
    value = value * 16807 % 2147483647;
    return value;
  }
}

let generator = pseudoRandom(1);

alert(generator()); // 16807
alert(generator()); // 282475249
alert(generator()); // 1622650073
```

ستعمل هذه أيضًا. ولكن بعد ذلك سنفقدُ قابلية التكـرار باسـتخدام `for..of` واسـتخدام المولّـد المـركب، وتلك ممكن أن تكون مفيدة في مكان آخر.

12.2 المكررات والمولدات غير المتزامنة

تُتيح لنا المُكرّرات غير المتزامنة المرور على البيانات التي تأتينا على نحوٍ غـير مـتزامن مـتى لـزم، مثـل حين نُنزّل شيئًا كتلةً كتلةً عبر الشبكة. المولّدات غير المتزامنة تجعل من ذلك أسهل فأسهل.

لنرى مثالًا أولًا لنفهم الصياغة، بعدها نرى مثالًا من الحياة العملية.

12.2.1 المكررات غير المتزامنة

تتشابه المُكرّرات غير المتزامنة مع تلك العادية، بفروق بسيطة في الصياغة. فكائن المُكرّر العادي (كما رأينـا في فصل الكائنات المكرَّرة Iterables) يكون هكذا:

```
let range = {
  from: 1,
  to: 5,
  // تستدعي حلقة for..of هذه الدالّة مرة واحدة في البداية

  [Symbol.iterator]() {

    // ...ستعيد كائن مُكرَّر:
    // لاحقًا ستعمل حلقة for..of فقط مع ذلك الكائن.
    // وتطلب منه القيم التالية باستخدام دالة next()
    return {
      current: this.from,
      last: this.to,

      // تُستدعى next() في كلّ تكرار من خلال الحلقة for..of

      next() { // (2)
        // يجب أن تعيد القيم ككائن {done:.., value :...}

        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
```

```
    };
  }
};

for(let value of range) {
  alert(value); // 1 then 2, then 3, then 4, then 5
}
```

راجِع ... لتعرف تفاصيل المُكرّرات العادية، لو لزم.

ولنجعل الكائن مُتكرّرًا غير متزامنًا:

1. علينا استعمال `Symbol.asyncIterator` بدل `Symbol.iterator`.
2. على `next()` إعادة وعد.
3. علينا استعمال حلقة `for await (let item of iterable)` لتكرار كائن معين.

فلنصنع كائن `range` مُتكرّرًا (كما أعلاه) ولكن يُعيد القيم بنحوٍ غير متزامن، قيمةً واحدةً كلّ ثانية:

```
let range = {
  from: 1,
  to: 5,
  // تستدعي حلقة for..of هذه الدالّة مرة واحدة في البداية

  [Symbol.asyncIterator]() { // (1)

    // ...ستعيد كائن مُكرِّر:
    // لاحقًا ستعمل حلقة for..of فقط مع ذلك الكائن.
    // وتطلب منه القيم التالية باستخدام دالة next()
    return {
      current: this.from,
      last: this.to,
      // تُستدعى next() في كلّ تكرار من خلال الحلقة for..of

      async next() { // (2)
        // يجب أن تعيد القيم ككائن {done:.., value :...}
        // وستُغلّف تلقائيًا في وعد غير متزامن
        // يمكننا استخدام await لتنفيذ أشياء غير متزامنة:
```

```
        await new Promise(resolve => setTimeout(resolve, 1000)); //(3)
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }
};

(async () => {
  for await (let value of range) { // (4)
    alert(value); // 1,2,3,4,5
  }
})()
```

كما ترى فالبنية تشبه المُكرّرات العادية:

1. ليكـون الكـائن مُتكـرّرًا على نحـوٍ غـير مـتزامن، يجب أن يحتـوي التـابع `Symbol.asyncIterator` (لاحِظ `(1)`).
2. على التابع إعادة الكائن وهو يحتوي التابع `next()` الذي يُعيد وعدًا (لاحِظ `(2)`).
3. ليس على التابع `next()` أن يكون غير متزامن `async`. فيمكن أن يكون تابِعًا عاديًـا يُعيـد وعـدًا، إلّا أنّ `async` تُتيح لنا استعمال `await` وهذا أفضل لنا. هنا نؤخّر التنفيذ ثانيةً واحدةً فقط (لاحِظ `(3)`).
4. ليحدث التكرار نستعمل `for await(let value of range)` (لاحِظ `(4)`)، أي نُضيف `await` قبل الحلقة `for`. هكذا تستدعي الحلقة `range[Symbol.asyncIterator]()` مرةً واحدة، وبعدها تابِع `next()` للقيم التالية.

إليك جدول ملخص:

	المُكرّرات	المُكرّرات غير المتزامنة
تابِع الكائن الذي يُقدّم المُكرّر	`Symbol.iterator`	`Symbol.asyncIterator`
قيمة `next()` المُعادة هي	أيّ قيمة	وعد `Promise`
to loop, use	`for..of`	`for await..of`

> لا يعمل معامل البقية `...` بطريقة غير متزامنة، فالميزات التي تتطلب تكرارات منتظمة ومتزامنة، لا تعمل مع تلك المتزامنة، على سبيل المثال، لن يعمل معامل البقية هنا:
>
> ```
> alert( [...range] ); // Error, no Symbol.iterator
> ```
>
> هذا أمر طبيعي، لأنه يتوقع العثور على `Symbol.iterator`، مثل `for..of` بدون `await`. ولكن ليس `Symbol.asyncIterator`.

12.2.2 المولدات غير المتزامنة

كما نعلم فلغة جافاسكربت تدعم المولّدات أيضًا، وهذه المولّدات مُتكرّرة. فلنتذكّر مولّد السلاسل من فصل المولِّدات كان يولّد سلسلة من القيم من متغير البداية `start` إلى متغير النهاية `end`:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    yield i;
  }
}

for(let value of generateSequence(1, 5)) {
  alert(value); // 1, then 2, then 3, then 4, then 5
}
```

لا يمكننا في الدوال العادية استعمال `await`، فعلى كلّ القيم أن تأتي بنحوٍ متزامن وليس ممكنًا وضع أيّ تأخير في حلقة `for..of`، ولكن، ماذا لو أردنا استعمال `await` في المولّد؟ مثلًا لنُرسل الطلبات عبر الشبكة؟ لا مشكلة، نضع قبله `async` هكذا:

```
async function* generateSequence(start, end) {

  for (let i = start; i <= end; i++) {
    // yay, can use await!
    await new Promise(resolve => setTimeout(resolve, 1000));

    yield i;
  }
}

(async () => {
  let generator = generateSequence(1, 5);
```

```
  for await (let value of generator) {
    alert(value); // 1, then 2, then 3, then 4, then 5
  }

})();
```

الآن لدينا مولّد غير متزامن، وقابل للتكرار مع `for await...of`.

إنها حقًا بسيطة للغاية. نضيف الكلمة المفتاحية `async`، ويستطيع المولّد الآن استخدام `await` بداخله، ويعتمد على الوعود والدوال غير متزامنة الأخرى. وتقنيًا، الفرق الآخر للمولّد غير المتزامن هو أنّ تابِع `generator.next()` صار غير متزامنًا أيضًا ويُعيد الوعود.

في المولّدات العادية نستعمل `result = generator.next()` لنأخذ القيم، بينما في المولّدات غير المتزامنة نُضيف `await` هكذا:

```
result = await generator.next(); // result = {value: ..., done:
true/false}
```

12.2.3 المكررات غير المتزامنة

كما نعلم ليكون الكائن مُتكرّرًا علينا إضافة رمز `Symbol.iterator` إليه.

```
let range = {
  from: 1,
  to: 5,

  [Symbol.iterator]() {
    return <object with next to make range iterable>
  }

}
```

هناك أسلوب شائع هو إعادة `Symbol.iterator` لمولّد بدل كائن صِرف يحمل التابع `next` كما المثال أعلاه. لنستذكر معًا المثال من الفصل المولِّدات:

```
let range = {
  from: 1,
  to: 5,

```

```
  *[Symbol.iterator]() { // [Symbol.iterator]: function*() اختصارًا لِ
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};

for(let value of range) {
  alert(value); // 1, then 2, then 3, then 4, then 5
}
```

نرى الكائن المخصّص `range` هنا مُتكرّرًا، والمولّد `*[Symbol.iterator]` يؤدّي المنطق اللازم لسرد القيم. لو أردنا إضافة أيّ إجراءات غير متزامنة للمولّدة، فعلينا استبدال الرمز `Symbol.iterator` بالرمز `Symbol.asyncIterator` غير المتزامن.

```
let range = {
  from: 1,
  to: 5,

  async *[Symbol.asyncIterator]() { // [Symbol.asyncIterator]: مشابه لِ
async function*()

    for(let value = this.from; value <= this.to; value++) {
      // توقف بين القيم لانتظار شيء ما
      await new Promise(resolve => setTimeout(resolve, 1000));
      yield value;
    }
  }
};

(async () => {
  for await (let value of range) {
    alert(value); // 1, then 2, then 3, then 4, then 5
  }
})();
```

الآن تأتينا القيم متأخرة عن بعضها البعض ثانيةً واحدة.

12.2.4 مثال من الحياة العملية

حتى اللحظة كانت الأمثلة كلّها بسيطة، لنفهم القصة فقط. الآن حان الوقت لنُطالع مثالًا وحالةً من الواقع.

نرى على الإنترنت خدمات كثيرة تقدّم لنا البيانات على صفحات (paginated). فمثلًا حين نطلب قائمة من المستخدمين يُعيد الطلب عددًا تحدّد مسبقًا (مثلًا 100 مستخدم)، أي "صفحة واحدة"، ويعطينا أيضًا عنوان الصفحة التالية.

هذا النمط شائع جدًا بين المطوّرين، وليس مستخدمًا للمستخدمين فقط بل لكلّ شيء. فمثلًا غِتهَب تتيح لك جلب الإيداعات بنفس الطريقة باستعمال الصفحات:

- عليك إرسال طلب إلى المسار `https://api.github.com/repos/<repo>/commits`.
- يستجيب الخادم بكائن JSON فيه 30 إيداعًا، ويُعطيك رابطًا يوصلك إلى الصفحة التالية في ترويسة `Link`.
- بعدها نستعمل ذلك الرابط للطلبات التالية لنجلب إيداعات أكثر، وهكذا.

ولكن ما نريد هو واجهة برمجية أبسط قليلًا: أي كائنًا مُتكرّرًا فيه الإيداعات كي نمرّ عليها بهذا النحو:

```
let repo = 'javascript-tutorial/en.javascript.info'; // المستودع الذي فيه الإيداعات التي نريد على غِتهَب

for await (let commit of fetchCommits(repo)) {
  // نُعالج كلّ إيداع commit
}
```

ونريد كتابة الدالة `fetchCommits(repo)` لتجلب تلك الإيداعات لنا وتؤدّي الطلبات اللازمة متى... لزم. كما ونترك في عهدتها مهمة الصفحات كاملةً. ما سنفعله من جهتنا هو أمر بسيط، `for await..of` فقط.

باستعمال المولّدات غير المتزامنة فهذه المهمّة تصير سهلة:

```
async function* fetchCommits(repo) {
  let url = `https://api.github.com/repos/${repo}/commits`;

  while (url) {
    const response = await fetch(url, { // (1)
      headers: {'User-Agent': 'Our script'},
      // يطلب github ترويسة user-agent
    });
```

```
    const body = await response.json();
    // (2) الرد بصيغة JSON مصفوفة من القيم

    // (3) عنوان الصفحة التالية موجود في الترويسات فاستخرجه
    let nextPage = response.headers.get('Link').match(/<(.*?)>;
rel="next"/);
    nextPage = nextPage && nextPage[1];

    url = nextPage;

    for(let commit of body) {
    // (4) المرور على القيم واحدة واحدة حتى الانتهاء
      yield commit;
    }
  }
}
```

1. نستعمل تابِع fetch من المتصفّح لتنزيل العنوان البعيد. يُتيح لنا التابع تقديم تصاريح الاستيثاق وغيرها من ترويسات مطلوبة. غِتهَب يطلب `User-Agent`.
2. نحلّل نتيجة الجلب على أنّها كائن JSON، وهذا تابِع آخر خاصّ بالتابع `fetch`.
3. نستلم هكذا عنوان الصفحة التالية من ترويسة `Link` داخل الردّ. لهذا العنوان تنسيق خاص فنستعمل تعبيرًا نمطيًا لتحليله. يظهر عنوان الصفحة التالية على هذا الشكل: `https://api.github.com/repositories/93253246/commits?page=2`، وموقع غِتهَب هو من يولّده بنفسه.
4. بعدها نُنتِج كلّ الإيداعات التي استلمناها، ومتى انتهت

مثال على طريقة الاستعمال (تعرض من كتبَ الإيداعات في الطرفيّة):

```
(async () => {

  let count = 0;

  for await (const commit of
fetchCommits('javascript-tutorial/en.javascript.info')) {

```

```
      console.log(commit.author.login);
      if (++count == 100) { // لنتوقف عند 100 إيداع
        break;
      }
    }

  })();
```

كما نريد تمامًا. الطريقة التي تعمل بها الطلبات بهذا النوع لا تظهر للشيفرات الخارجية وتبقى تفاصيل داخلية. المهم لنا هي استعمال مولّد غير متزامن يُعيد الإيداعات.

12.2.5 الخلاصة

تعمل المُكرّرات والمولّدات العادية كما نريد لو لم تأخذ البيانات التي نحتاج وقتًا حتّى تتولّد.

وحين نتوقّع بأنّ البيانات ستأتي بنحوٍ غير متزامن بينها انقطاعات، فيمكن استعمال البدائل غير المتزامنة مثل `for await..of` بدل `for..of`.

الفرق في الصياغة بين المُكرّرات العادية وغير المتزامنة:

	المُتكرّر	المُتكرّر غير المتزامن
التابع الذي يُقدّم المُكرّر	`Symbol.iterator`	`Symbol.asyncIterator`
قيمة `next()` المُعادة	`{value:…, done: true/false}`	`Promise` والّتي تعوض لتصبح `{value:…, done: true/false}`

الفرق في الصياغة بين المولّدات العادية وغير المتزامنة:

	المولّدات	المولّدات غير المتزامنة
التعريف	`*function`	`*async function`
قيمة `next()` المُعادة	`{value:…, done: true/false}`	`Promise` والّتي تعوض لتصبح `{value:…, done: true/false}`

غالبًا ونحن نعمل على تطوير الوِب نواجه سيولًا كبيرة من البيانات، سيولًا تأتينا قطعًا قطعًا (مثل تنزيل أو رفع الملفات الكبيرة).

يمكننا هنا استعمال المولّدات غير المتزامنة لمُعالجة هذه البيانات. كما يجدر بنا الملاحظة كيف أنّ لبعض البيئات (مثل المتصفّحات) واجهات برمجة أخرى باسم السيول (Stream) وهي تقدّم لنا واجهات خاصّة للعمل مع السيول هذه، ذلك لتحويل شكل البيانات ونقلها من سيل إلى آخر (مثل التنزيل من مكان وإرسال ذلك التنزيل إلى مكان آخر مباشرةً).

دورة تطوير واجهات المستخدم

ابدأ مسارك المهني كمطور واجهات المواقع والمتاجر الإلكترونية فور انتهائك من الدورة

التحق بالدورة الآن

13. الوحدات Modules

يتضمن هذا الفصل الأقسام التالية:

1. مقدمة إلى الوحدات Modules
2. تصدير الوحدات واستيرادها
3. استيراد الوحدات ديناميكيًا

13.1 مقدمة إلى الوحدات Modules في جافاسكربت

سنرى سريعًا بينما تطبيقنا يكبُر حجمًا وتعقيدًا بأنّ علينا تقسيمه إلى ملفات متعدّدة، أو "وحدات" (module). عادةً ما تحتوي الوِحدة على صنف أو مكتبة فيها دوالّ.

كانت محرّكات جافاسكربت تعمل لفترة طويلة جدًا دون أيّ صياغة وِحدات على مستوى اللغة، ولم تكن هذه بالمشكلة إذ أنّ السكربتات سابقًا كانت بسيطة وسهلة ولم يكن هناك داعٍ فعلي للوِحدات.

ولكن كالعادة صارت السكربتات هذه أكثر تعقيدًا وأكبر، فكان على المجتمع اختراع طرائق مختلفة لتنظيم الشيفرات في وحدات (أو مكتبات خاصّة تُحمّل تلك الوِحدات حين الطلب).

مثال:

- AMD: هذه إحدى نُظم المكتبات القديمة جدًا والتي كتبت تنفيذها بدايةً المكتبة require.js.
- CommonJS: نظام الوِحدات الذي صُنع لخوادم Node.js.
- UMD: نظام وِحدات آخر (اقتُرح ليكون للعموم أجمعين) وهو متوافق مع AMD وCommonJS.

أمّا الآن فهذه المكتبات صارت (أو تصير، يومًا بعد آخر) جزءًا من التاريخ، ولكن مع ذلك سنراها في السكربتات القديمة.

ظهر نظام الوِحدات (على مستوى اللغة) في المعيار عام 2015، وتطوّر شيئًا فشيئًا منذئذ وصارت الآن أغلب المتصفّحات الرئيسة (كما و Node.js) تدعمه. لذا سيكون أفضل لو بدأنا دراسة عملها من الآن.

13.1.1 ما الوحدة؟

الوِحدة هي ملف، فقط. كلّ نص برمجي يساوي وحدة واحدة.

يمكن أن تُحمّل الوِحدات بعضها البعض وتستعمل توجيهات خاصة مثل التصدير `export` والاستيراد `import` لتتبادل الميزات فيما بينها وتستدعي الدوال الموجودة في وحدة ص، من وحدة س:

- تقول الكلمة المفتاحية `export` للمتغيرات والدوال بأنّ الوصول إليها من خارج الوِحدة الحالية هو أمر مُتاح.
- وتُتيح `import` استيراد تلك الوظائف من الوِحدات الأخرى.

فمثلًا لو كان لدينا الملف `sayHi.js` وهو يُصدّر دالّةً من الدوال:

```
//    sayHi.js
export function sayHi(user) {
```

```
  alert(`Hello, ${user}!`);
}
```

فيمكن لملف آخر استيراده واستعمالها:

```
//    main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function... نوعها دالة
sayHi('Ahmad'); // Hello, Ahmad!
```

تتوجه تعليمة `import` للوِحدة `./sayHi.js` عبر المسار النسبي المُمرر لها. ويسند التابع `sayHi` للمتغيّر الذي يحمل نفس اسم التابع. لنشغّل المثال في المتصفّح.

تدعم الوِحدات كلمات مفتاحية ومزايا خاصة، لذلك علينا إخبار المتصفّح بأنّ هذا السكربت هو وِحدة ويجب أن يُعامل بهذا النحو، ذلك باستعمال الخاصية `<script type="module">`. هكذا:

- ملف index.html:

```
<!doctype html>
<script type="module">
  import {sayHi} from './say.js';

  document.body.innerHTML = sayHi('Ahmad');
</script>
```

- ملف say.js:

```
export function sayHi(user) {
  return `Hello, ${user}!`;
}
```

يجلب المتصفّح الوِحدة تلقائيًا ويقيم الشيفرة البرمجية بداخلها (ويستورد جميع الوحدات المتعلقة بها إن لزم الأمر)، وثمّ يشغلها وتكون نتيجة ما سبق:

```
Hello, Ahmad!
```

13.1.2 ميزات الوحدات الأساسية

ولكن ما الفرق بين الوِحدات والسكربتات (الشيفرات) "العادية" تلك؟

للوِحدات ميزات أساسية تعمل على محرّكات جافاسكربت للمتصفّحات وللخوادم على حدّ سواء.

ا. الوضع الصارم الافتراضي

تستخدم الوِحدات الوضع الصارم تلقائيًا فمثلًا إسناد قيمة لمتحول غير معرّف سينتج خطأ.

```
<script type="module">
  a = 5; // خطأ
</script>
```

ب. النطاق على مستوى الوحدات

كلّ وِحدة لها نطاق عالي المستوى خاص بها. بتعبيرٍ آخر، لن يُنظر للمتغيّرات والدوال من الوحدات الأخرى، وإنما يكون نطاق المتغيرات محلي.

نرى في المثال أدناه أنّا حمّلنا نصّين برمجيين، ويحاول الملف `hello.js` استعمال المتغير `user` المصرّح عنه في الملف `user.js` ولا يقدر:

- ملف index.html:

```
<!doctype html>
<script type="module" src="user.js"></script>
<script type="module" src="hello.js"></script>
```

- ملف user.js:

```
let user = "Ahmad";
```

- الملف hello.js:

```
alert(user);
```

لاحظ أن نتيجة ما سبق هي لا شيء لأن الدالة `alert` لم تتعرف على المتغير `user` الغير موجود، فكل وحدة لها متغيرات خاصة بها ويمكنها تصدير (عبر `export`) ما تريد للآخرين من خارجها رؤيته، واستيراد (عبر `import`) ما تحتاج استعماله. لذا علينا استيراد `user.js` و`hello.js` وأخذ المزايا المطلوبة منهما بدل الاعتماد على المتغيّرات العمومية.

هذه النسخة الصحيحة من الشيفرة:

- ملف index.html:

```
<!doctype html>
```

```
<script type="module" src="hello.js"></script>
```

- ملف user.js:

```
export let user = "Ahmad";
```

- الملف hello.js:

```
import {user} from './user.js';
document.body.innerHTML = user; // Ahmad
```

يوجد في المتصفح نطاق مستقل عالي المستوى. وهو موجود أيضًا للوحدات `<script type="module">`:

```
<script type="module">
  // سيكون المتغير مرئي في مجال هذه الوِحدة فقط
  let user = "Ahmad";
</script>

<script type="module">
  alert(user); // خطأ: المتغير user غير معرّف
</script>
```

ولو أردنا أن ننشئ متغيرًا عامًا على مستوى النافذة يمكننا تعيينه صراحة للمتغيّر `window` ويمكننا الوصول إليه هكذا `window.user`. ولكن لابد من وجود سبب وجيهٍ لذلك.

ج. تقييم شيفرة الوِحدة لمرة واحدة فقط

لو استوردتَ نفس الوِحدة في أكثر من مكان، فلا تُنفّذ شيفرتها إلّا مرة واحدة، وبعدها تُصدّر إلى من استوردها. ولهذا توابع مهمّ معرفتها. لنرى بعض الأمثلة.

أولًا، لو كان لشيفرة الوِحدة التي ستُنفّذ أيّ تأثيرات (مثل عرض رسالة أو ما شابه)، فاستيرادها أكثر من مرّة سيشغّل ذلك التأثير مرة واحدة، وهي أول مرة فقط:

```
// 📁 alert.js
alert("Module is evaluated!"); // نُفّذت شيفرة الوِحدة!
// نستورد نفس الوِحدة من أكثر من ملف
// 📁 1.js
import `./alert.js`; // نُفّذت شيفرة الوِحدة!
```

```
//    2.js
import `./alert.js`; // (لا نرى شيئًا هنا)
```

في الواقع، فشيفرات الوِحدات عالية المستوى في بنية البرمجية لا تُستعمل إلّا لتمهيد بنى البيانات الداخلية وإنشائها. ولو أردنا شيئًا نُعيد استعماله، نُصدّر الوِحدة.

الآن حان وقت مثال مستواه متقدّم أكثر. لنقل بأنّ هناك وحدة تُصدّر كائنًا:

```
//    admin.js
export let admin = {
  name: "Ahmad"
};
```

لو استوردنا هذه الوِحدة من أكثر من ملف، فلا تُنفّذ شيفرة الوِحدة إلّا أول مرة، حينها يُصنع كائن المدير `admin` ويُمرّر إلى كلّ من استورد الوِحدة.

وهكذا تستلم كلّ الشيفرات كائن مدير `admin` واحد فقط لا أكثر ولا أقل:

```
//    1.js
import {admin} from './admin.js';
admin.name = "Pete";

//    2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// كِلا الملفين 1.js و 2.js سيستوردان نفس الكائن
// التغييرات الّتي ستحدث في الملف 1.js ستكون مرئية في الملف 2.js
```

ولنؤكد مجددًا، تُنفذّ الوِحدة لمرة واحدة فقط. وتُنشئ الوِحدات المراد تصديرها وتُشارك بين المستوردين لذا فإن تغير شيء ما في كائن `admin` فسترى الوِحدات الأخرى ذلك.

يتيح لنا هذا السلوك "ضبط" الوِحدة عند أوّل استيراد لها، فنضبط خاصياتها المرة الأولى، ومتى ما استورِدت مرة أخرى تكون جاهزة.

فمثلًا قد تقدّم لنا وحدة `admin.js` بعض المزايا ولكن تطلب أن تأتي امتيازات الإدارة من خارج كائن `admin` إلى داخله:

```
//    admin.js
```

```
export let admin = { };
export function sayHi() {
  alert(`Ready to serve, ${admin.name}!`);
}
```

نضبط في `init.js` (أوّل نص برمجي لتطبيقنا) المتغير `admin.name`. بعدها سيراه كلّ من أراد بما في ذلك الاستدعاءات من داخل وحدة `admin.js` نفسها:

```
//    init.js
import {admin} from './admin.js';
admin.name = "Pete";
```

ويمكن لوحدة أخرى استعمال `admin.name`:

```
//    other.js
import {admin, sayHi} from './admin.js';
alert(admin.name); // Pete
sayHi(); // Ready to serve, Pete!
```

د. import.meta

يحتوي الكائن `import.meta` على معلومات الوِحدة الحالية. ويعتمد محتواها على البيئة الحالية، ففي المتصفّحات يحتوي على عنوان النص البرمجي أو عنوان صفحة الوِب الحالية لو كان داخل HTML:

```
<script type="module">
  alert(import.meta.url);
  // عنوان URL للسكربت (عنوان URL لصفحة HTML للسكربت الضمني)
</script>
```

ه. this في الوِحدات ليست معرّفة

قد تكون هذه الميزة صغيرة، ولكنّا سنذكرها ليكتمل هذا الفصل. في الوحدات، قيمة `this` عالية المستوى غير معرّفة. وازن بينها وبين السكربتات غير المعتمدة على الوحدات، إذ ستكون `this` كائنًا عامًا:

```
<script>
  alert(this); // window
</script>

<script type="module">
```

```
  alert(this); // غير معرَف
</script>
```

13.1.3 الميزات الخاصة بالمتصفّحات

كما أن هناك عدّة فروق تخصّ المتصفحات السكربتات (المعتمدة على الوحدات) بالنوع `type="module"` موازنةً بتلك العادية.

لو كنت تقرأ هذا الفصل لأول مرة، أو لم تكن تستعمل المحرّك في المتصفّح فيمكنك تخطّي هذا القسم.

ا. سكربتات الوِحدات مؤجلة

دائمًا ما تكون سكربتات الوِحدات مؤجلة، ومشابهة لتأثير السِمة `defer` (الموضحة في هذا الفصل)، لكل من السكربتات المضمّنة والخارجية. أي وبعبارة أخرى:

- تنزيل السكربتات المعتمدة على الوِحدات الخارجية `<script type="module" src="...">` لا تُوقف معالجة HTML فتُحمّل بالتوازي مع الموارد الأخرى.
- تنتظر السكربتات المعتمدة على الوِحدات حتّى يجهز مستند HTML تمامًا (حتّى لو كانت صغيرة وحُمّلت بنحوٍ أسرع من HTML) وتُشغّل عندها.
- تحافظ على الترتيب النسبي للسكربتات: فالسكربت ذو الترتيب الأول ينفذّ أولاً.

ويسبّب هذا بأن "ترى" السكربتات المعتمدة على الوِحدات صفحة HTML المحمّلة كاملة بما فيه عناصر الشجرة أسفلها. مثال:

```
<script type="module">
  alert(typeof button); // كائن (object): يستطيع السكربت رؤية العناصر أدناه
  // بما أن الوِحدات مؤجلة. سيُشغل السكربت بعد تحميل كامل الصفحة
</script>
Compare to regular script below:
<script>

  alert(typeof button);
  // خطأ: الزر (button) غير معرّف. لن يستطيع السكربت رؤية العناصر أدناه
  // السكربت العادي سيُشغل مباشرة قبل أن يُستكمل تحميل الصفحة
</script>

<button id="button">Button</button>
```

لاحِظ كيف أنّ النص البرمجي الثاني يُشغّل فعليًا قبل الأول! لذا سنرى أولًا `undefined` وبعدها `object`. وذلك بسبب كون عملية تشغيل الوِحدات مُؤجلة لذلك سننتظر لاكتمال معالجة المستند. نلاحظ أن السكربت العادي سيُشغلّ مباشرة بدون تأجيل ولذا سنرى نتائجه أولًا.

علينا أن نحذر حين نستعمل الوِحدات إذ أنّ صفحة HTML تظهر بينما الوِحدات تُحمّل، وبعدها تعمل الوحدات. بهذا يمكن أن يرى المستخدم أجزاءً من الصفحة قبل أن يجهز تطبيق جافاسكربت، ويرى بأنّ بعض الوظائف في الموقع لا تعمل بعد. علينا هنا وضع "مؤشّرات تحميل" أو التثبّت من أنّ الزائر لن يتشتّت بهذا الأمر.

ب. خاصية Async على السكربتات المضمّنة

بالنسبة للسكربتات غير المعتمدة على الوِحدات فإن خاصية `async` (اختصارًا لكلمة Asynchronous أي غير المتزامن) تعمل على السكربتات الخارجية فقط. وتُشغل السكربتات غير المتزامنة مباشرة عندما تكون جاهزة، بشكل مستقل عن السكربتات الأخرى أو عن مستند HTML.

تعمل السكربتات المعتمدة على الوِحدات طبيعيًا في السكربتات المضمّنة. فمثلًا يحتوي السكربت المُضمن أدناه على الخاصية `async`، لذلك سيُشغّل مباشرة ولن ينتظر أي شيء.

وهو ينفذ عملية الاستيراد (اجلب الملف `./analytics.js`) وشغله عندما يصبح جاهزًا، حتى وإن لم ينتهِ مستند HTML بعد. أو السكربتات الأُخرى لا تزال معلّقة. وهذا جيد للتوابع المستقلة مثل العدادات والإعلانات ومستمع الأحداث على مستوى المستند.

في المثال أدناه، جُلبت جميع التبعيات (من ضمنها analytics.js). ومن ومن ثمّ شُغّل السكربت ولم ينتظر حتى اكتمال تحميل المستند أو السكربتات الأخرى.

```
<script async type="module">
  import {counter} from './analytics.js';

  counter.count();
</script>
```

ج. السكربتات الخارجية

تختلف السكربتات الخارجية التي تحتوي على السمة `type="module"` في جانبين:

1. تنفذ السكربتات الخارجية التي لها نفس القيمة للخاصية `src` مرة واحدة فقط. فهنا مثلًا سيُجلب السكربت `my.js` وينفذ مرة واحدة فقط.

```
<script type="module" src="my.js"></script>
```

```
<script type="module" src="my.js"></script>
```

2. تتطلب السكربتات الخارجية التي تجلب من مصدر مستقل (موقع مختلف عن الأساسي) ترويسات CORS والموضحة في هذا الفصل. بتعبير آخر إن جُلِبَ سكربت يعتمد على الوِحدات من مصدر معين فيجب على الخادم البعيد أن يدعم ترويسات السماح بالجلب `Access-Control-Allow-Origin`. يجب أن يدعم المصدر المستقل `Access-Control-Allow-Origin` (في المثال أدناه المصدر المستقل هو another-site.com) وإلا فلن يعمل السكربت.

```
<script type="module" src="http://another-site.com/their.js"></script>
```

وذلك سيضمن لنا مستوى أمان أفضل افتراضيًا.

د. لا يُسمح بالوحدات المجردة

في المتصفح، يجب أن تحصل تعليمة `import` على عنوان URL نسبي أو مطلق. وتسمى الوِحدات التي بدون أي مسار بالوحدات المجردة. وهي ممنوع في تعليمة `import`.

لنأخذ مثالًا يوضح الأمر، هذا `import` غير صالح:

```
import {sayHi} from 'sayHi'; // خطأ وِحدة مجردة
// يجب أن تمتلك الوِحدة مسارًا مثل: './sayHi.js' أو مهما يك موقع هذه الوِحدة
```

تسمح بعض البيئات، مثل Node.js أو أدوات تجميع الوِحدات باستخدام الوِحدات المجردة، دون أي مسار، حيث أن لديها طرقها الخاصة للعثور على الوِحدات والخطافات لضبطها. ولكن حتى الآن لا تدعم المتصفحات الوِحدات المجردة.

ه. التوافقية باستخدام "nomodule"

لا تفهم المتصفحات القديمة طريقة استخدام الوِحدات في الصفحات `type = "module"`. بل وإنها تتجاهل السكربت ذو النوعٍ غير المعروف. بالنسبة لهم، من الممكن تقديم نسخة مخصصة لهم باستخدام السمة `nomodule`:

```
<script type="module">
  alert("Runs in modern browsers");
</script>

<script nomodule>
  alert("Modern browsers know both type=module and nomodule, so skip this"):
```

```
  alert("Old browsers ignore script with unknown type=module, but
execute this.");
</script>
```

المتصـــفحات الحديثـــة تعـــرف type=module و nomodule لـــذا لن تنفـــذ الأخـــير بينمـــا ســـتتجاهل المتصفحات القديمة الوسم ذا السمة type=module ولكن ستنفذ وسم nododule.

13.1.4 أدوات البناء

في الحياة الواقعية، نادرًا ما تستخدم وحدات المتصفح في شكلها "الخام". بل عادةّ نجمعها مـع أداة خاصـة مثل Webpack وننشرها على خادم النشر.

إحدى مزايا استخدام المجمعات - فهي تمنح المزيد من التحكم في كيفية التعامل مع الوحدات، مما يسمح بالوحدات المجردة بل وأكثر من ذلك بكثير، مثل وحدات HTML/CSS.

تؤدي أدوات البناء بعض الوظائف منها:

1. جلب الوِحدة الرئيسية `main`، وهي الوِحدة المراد وضعها في وسم `<script type ="module">` في ملف HTML.
2. تحليل التبعيات: تحليل تعليمات الاستيراد الخاصة بالملف الرئيسي وثم للملفات المستوردة أيضًا ومـا إلى ذلك.
3. إنشاء ملفًا واحدًا يحتوي على جميع الوِحـدات (مـع إمكانيـة تقسـيمهُ لملفـات متعـددة)، مـع اسـتبدال تعليمة `import` الأصلية بتوابع الحزم لكي يعمـل السـكربت. كمـا تـدعم أنـواع وحـدات "خاصـة" مثـل وحدات HTML/CSS.
4. يمكننا تطبيق عمليات تحويل وتحسينات أخرى في هذه العملية مثل:
   - إزالة الشيفرات الّتي يتعذر الوصول إليها.
   - إزالة تعليمات التصدير غير المستخدمة (مشابهة لعملية هز الأشجار وسقوط الأوراق اليابسة).
   - إزالة العبارات الخاصة بمرحلة التطوير مثل `console` و`debugger`.
   - تحويل شيفرة جافاسكربت الحديثة إلى شيفرة أقدم باستخدام وظائف مماثلة للحزمة Babel.
   - تصغير الملف الناتج (إزالة المسافات، واستبدال المتغيرات بأسماء أقصر، وما إلى ذلك).

عند استخدامنا لأدوات التجميع سيُجمع السـكربت ليصـبح في ملـف واحـد (أو ملفـات قليلـة) ، تُسـتبدل تعليمات `import/export` بداخل السكربتات بتوابع المُجمّع الخاصـة. لـذلك لا يحتـوي السـكربت "المُجَمّـع" الناتج على أي تعليمات `import/export`، ولا يتطلب السِمة `type="module"`، ويمكننا وضعه في سكربت

عـادي، في المثـال أدنـاه لنفـترض أننـا جمعّنـا الشـيفرات في ملـف bundle.js باسـتخدام مجمـع حـزم مثل: Webpack.

```
<script src="bundle.js"></script>
```

ومع ذلك يمكننا استخدام الوِحدات الأصـلية (في شـكلها الخـام). لـذلك لن نسـتخدم هنـا أداة Webpack: يمكنك التعرف عليها وضبطها لاحقًا.

13.1.5 الخلاصة

لنلخص المفاهيم الأساسية:

1. الوِحدة هي مجرد ملف. لجعل تعليمـتي `import/export` تعملان، سـتحتاج المتصـفحات إلى وضـع السِمة التالية `<script type ="module">`. تحتوي الوِحدات على عدة مُميزات:
    - مؤجلة افتراضيًا.
    - تعمل الخاصية Async على السكربتات المضمّنة.
    - لتحميل السكربتات الخارجية من مصـدر مسـتقل، يجب اسـتخدام طريقـة (المَنفـذ / الـبروتوكول / المجال)، وسنحتاج لترويسات CORS أيضًا.
    - ستُتجاهل السكربتات الخارجية المكررة.
2. لكل وِحدة من الوِحـدات نطـاق خـاص بهـا، وتتبـادلُ الوظـائف فيمـا بينهـا من خلال اسـتيراد وتصـدير الوِحدات `import/export`.
3. تستخدم الوِحدات الوضع الصارم دومًا `use strict`.
4. تُنفذ شيفرة الوِحدة لمرة واحدة فقط. وتُصدر إلى من استوردها لمـرة واحـدة أيضًـا، ومن ثمّ تُشـارك بين المستوردين.

عندما نستخدم الوحدات، تنفذ كل وِحدة وظيفة معينـة وتُصـدرها. ونسـتخدم تعليمـة `import` لاسـتيرادها مباشرة عند الحاجة. إذ يُحمل المتصفح السكربت ويقيّمه تلقائيًا.

وبالنسبة لوضع النشر، غالبًا ما يستخدم الناس مُحزّم الوِحدات مثل Webpack لتجميع الوِحدات معًـا لرفـع الأداء ولأسباب أخرى.

سنرى في الفصل التالي مزيدًا من الأمثلة عن الوِحدات، وكيفية تصديرها واستيرادها.

13.2 تصدير الوحدات واستيرادها

لمُوجِّهات (تعليمات) الاستيراد والتصدير أكثر من صياغة برمجية واحدة رأينا في الفصل السابق، مقدمة إلى الوحدات استعمالًا بسيطًا له، فهيا نرى بقية الاستعمالات.

13.2.1 التصدير قبل التصريح

يمكننا أن نقول لأيّ تصريح بأنّه مُصدّر بوضع عبارة `export` قبله، كان التصريح عن متغيّر أو عن دالة أو عن صنف. فمثلًا، التصديرات هنا كلّها صحيحة:

```
// تصدير مصفوفة
export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct',
'Nov', 'Dec'];


// تصدير ثابت
export const MODULES_BECAME_STANDARD_YEAR = 2015;


// تصدير صنف
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

لا يوجد فواصل منقوطة بعد تعليمة التصدير للأصناف أو الدوال

لاحظ أن تعليمة `export` قبل الصنف أو الدالة لا يجعلها تعابير الدوالّ. ولو أنه يصُدرها، لكنه لا يزال تعريفًا للدالّة أو الصنف.

لا توصي معظم الأدلة التعليمية بوضع فاصلة منقوطة بعد تعريف الدوال والأصناف. لهذا السبب لا داعي للفاصلة المنقوطة في نهاية التعليمة `export class` والتعليمة `export function`:

```
export function sayHi(user) {
  alert(`Hello, ${user}!`);
} // لاحظ لا يوجد فاصلة منقوطة في نهاية التعريف
```

13.2.2 التصدير بعيدًا عن التصريح

كما يمكننا وضع عبارة `export` لوحدها. هنا نصرّح أولًا عن الدالتين وبعدها نُصدّرهما:

```
//    say.js
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

export {sayHi, sayBye}; // تصدير قائمة من المتغيرات
```

أو... يمكننا تقنيًا وضع `export` أعلى الدوال أيضًا.

13.2.3 عبارة استيراد كلّ شيء

عادةً نضع قائمة بما نريد استيراده في أقواس معقوفة `import {...}`، هكذا:

```
//    main.js
import {sayHi, sayBye} from './say.js';

sayHi('Ahmad'); // Hello, Ahmad!
sayBye('Ahmad'); // Bye, Ahmad!
```

ولكن لو أردنا استيراد وحدات كثيرة، فيمكننا استيراد كلّ شيء كائنًا واحدًا باستعمال `import * as <obj>` هكذا:

```
//    main.js
import * as say from './say.js';

say.sayHi('Ahmad');
say.sayBye('Ahmad');
```

يقول المرء من النظرة الأولى "استيراد كلّ شيء فكرة جميلة جدًا، وكتابة الشيفرة سيكون أسرع. أساسًا لمَ نقول جهارةً ما نريد استيراده؟" ذلك... لأسباب وجيهة:

1. استيراد ما تريد يساعد على اختصار الأسماء مثل استعمال `sayHi()` بدلًا من `say.sayHi()`.

2. الاستيراد الصريح يساعد على تنظيم الشيفرة وتحسين هيكلتها مما يسهل من تعديلها وإعادة ترتيبها.

ولا تخشى أيضًا من الاستيراد بكثرة واستيراد ما هب ودب حتى لو لم تستعمل ما استوردت، إذ أن أغلب أدوات البناء الحديثة (مثل: webpack وغيرها) تعمل على تحسين تجميعة الشيفرة النهائية للوحدات واختصار ما استوردته ولم تستعمله ببساطة. لنقل مثلًا بأنّا أضفنا مكتبة خارجية اسمها say.js إلى مشروعنا، وفيها دوالّ عديدة:

```
//    say.js
export function sayHi() { ... }
export function sayBye() { ... }
export function becomeSilent() { ... }
```

هكذا نستعمل واحدة فقط من دوالّ say.js في مشروعنا:

```
//    main.js
import {sayHi} from './say.js';
```

حينها تأتي أداة التحسين وترى ذلك، فتُزيل الدوال الأخرى من الشيفرة ... بذلك يصغُر حجم الملف المبني. هذا ما نسميه هز الشجر (لتَسقطَ الأوراق اليابسة فقط).

13.2.4 استيراد كذا بالاسم كذا as

يمكننا كذلك استعمال as لاستيراد ما نريد بأسماء مختلفة.

فمثلًا يمكننا استيراد الدالة sayHi في المتغير المحلي hi لنختصر الكلام، واستيراد sayBye على أنّها bye:

```
//    main.js
import {sayHi as hi, sayBye as bye} from './say.js';
hi('Ahmad'); // Hello, Ahmad!
bye('Ahmad'); // Bye, Ahmad!
```

13.2.5 تصدير كذا بالاسم كذا as

نفس صياغة الاستيراد موجودة أيضًا للتصدير export. فلنصدّر الدوال على أنّها hi وbye:

```
//    say.js
...
export {sayHi as hi, sayBye as bye};
```

الآن صارت hi وbye هي الأسماء "الرسمية" للشيفرات الخارجية وستُستعمل عند الاستيراد:

```
//    main.js
import * as say from './say.js';

// لاحِظ الفرق
say.hi('Ahmad'); // Hello, Ahmad!
say.bye('Ahmad'); // Bye, Ahmad!
```

13.2.6 التصدير المبدئي

في الواقع العملي، ثمّة نوعين رئيسين من الوحدات.

1. تلك التي تحتوي مكتبة (أي مجموعة من الدوال) مثل وحدة `say.js` أعلاه.
2. وتلك التي تصرّح عن كيانٍ واحد مثل وحدة `user.js` التي تُصدّر `class User` فقط.

عادةً ما يُحبّذ استعمال الطريقة الثانية كي يكون لكلّ "شيء" وحدةً خاصة به. ولكن هذا بطبيعة الحال يطلب ملفات كثيرة إذ يطلب كلّ شيء وحدةً تخصّه باسمه، ولكنّ هذه ليست بمشكلة، أبدًا. بل على العكس هكذا يصير التنقل في الشيفرة أسهل (لو كانت تسمية الملفات مرضية ومرتّبة في مجلدات).

توفر الوِحدات طريقة لصياغة عبارة `export default` (التصدير المبدئي) لجعل "سطر تصدير واحد لكلّ وِحدة" تبدو أفضل. ضَع `export default` قبل أيّ كيان لتصديره:

```
//    user.js
export default class User { // "default" تُضيف فقط
  constructor(name) {
    this.name = name;
  }
}
```

لكلّ ملف سطر تصدير `export default` واحد لا أكثر. وبعدها نستورد الكيان بدون الأقواس المعقوفة:

```
//    main.js
import User from './user.js'; // لا نضع {User}، بل User

new User('Ahmad');
```

أسطر الاستيراد التي لا تحتوي الأقواس المعقوفة أجمل من تلك التي تحتويها. يشيع خطأ نسيان تلك الأقواس حين يبدأ المطورون باستعمال الوِحدات. لذا تذكّر دائمًا، يطلب سطر الاستيراد `import` أقواس معقوفة للكيانات المُصدّرة والتي لها أسماء، ولا يطلبها لتلك المبدئية.

التصدير الذي له اسم	التصدير المبدئي
`export class User {...}`	`export default class User {...}`
`import {User} from ...`	`import User from ...`

يمكننا نظريًا وضع النوعين من التصدير معًا في نفس الوحدة (الذي له اسم والمبدئي)، ولكن عمليًا لا يخلط الناس عادةً بينها، بل للوِحدة إمّا تصديرات لها أسماء، أو التصدير المبدئي.

ولأنّه لا يمكن أن يكون لكلّ ملف إلا تصديرًا مبدئيًا واحدًا، فيمكن للكيان الذي صُدّر ألّا يحمل أيّ اسم. فمثلًا التصديرات أسفله كلّها صحيحة مئة في المئة:

```
export default class { // لا اسم للصنف
  constructor() { ... }
}

export default function(user) { // لا اسم للدالة
  alert(`Hello, ${user}!`);
}

// نُصدّر قيمةً واحدة دون صنع متغيّر
export default ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

لا مشكلة بتاتًا بعدم كتابة الاسم إذ لا نرى `export default` إلّا مرّة في الملف، بهذا تعرف تمامًا أسطر `import` (بدون استعمال الأقواس المعقوفة) ما عليها استيراده.

ولكن دون `default` فهذا التصدير سيُعطينا خطأً:

```
export class { // Error! (non-default export needs a name)
  constructor() {}
}
```

ا. الاسم المبدئي

تُستعمل في حالات معيّنة الكلمة المفتاحية `default` للإشارة إلى التصدير المبدئي. فمثلًا لتصدير الدالة بنحوٍ منفصل عن تعريفها:

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

// كما لو أضفنا "export default" قبل الدالة
export {sayHi as default};
```

أو لنقل بأنّ الوحدة `user.js` تُصدّر شيئًا واحدًا "مبدئيًا" وأخرى لها أسماء (نادرًا ما يحدث، ولكنّه يحدث):

```
//    user.js
export default class User {
  constructor(name) {
    this.name = name;
  }
}

export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

هكذا نستورد التصدير المبدئي مع ذلك الذي لديه اسم:

```
//    main.js
import {default as User, sayHi} from './user.js';

new User('Ahmad');
```

وأخيرًا، حين نستورد كلّ شيء `*` على أنّه كائن، فستكون خاصية `default` هي كما التصدير المبدئي:

```
//    main.js
import * as user from './user.js';

let User = user.default; // the default export
new User('Ahmad');
```

كلمتين بخصوص سوء التصديرات المبدئية

التصديرات التي لها أسماء تكون صريحة، أي أنّها تقول تمامًا ما الّذي يجب أن نستورده، وبذلك يكون لـدينا هذه المعلومات منهم، وهذا شيء جيد.

تُجبرنا التصديرات التي لها أسماء باستعمال الاسم الصحيح كما هو بالضبط لاستيراد الوحدة:

```
import {User} from './user.js';
// {User} ولن تعمل import {MyUser} إذ يجب أن يكون الاسم
```

بينما في حالة التصدير المبدئي نختار نحن الاسم حين نستورد الوِحدة:

```
import User from './user.js'; // works
import MyUser from './user.js'; // works too
// ويمكن أيضًا أن تكون "استورِد كل شيء" import Anything... وستعمل بلا أدنى مشكلة
```

هذا قد يؤدّي إلى أن يستعمل أعضاء الفريق أسماء مختلفة لاستيراد الشيء ذاته، وهذا طبعًا ليس بالجيد.

عادةً ولنتجنّب ذلك ونُحافظ على اتساق الشيفرة، نستعمل القاعدة القائلة بأنّ أسماء المتغيرات المُستورَدة يجب أن تُوافق أسماء الملفات، هكذا مثلًا:

```
import User from './user.js';
import LoginForm from './loginForm.js';
import func from '/path/to/func.js';
...
```

مع ذلك تنظُر بعض الفِرق لهذا الأمر على أنه عقبة للتصديرات المبدئية فتفضّل استعمال التصديرات التي لها اسم دومًا. فحتّى لو كانت نصدّر شيئًا واحدًا فقط فما زالت تُصدّره باسم دون استعمال `default`.

كما يسهّل هذا إعادة التصدير (طالِع أسفله).

13.2.7 إعادة التصدير

تُتيح لنا صياغة "إعادة التصدير" `export ... from ...` استيراد الأشياء وتصديرها مباشرةً (ربما باسم آخر) هكذا:

```
export {sayHi} from './say.js'; // sayHi نُعيد تصدير

export {default as User} from './user.js'; // نُعيد تصدير المبدئي
```

ولكن فيمَ نستعمل هذا أصلًا؟ لنرى مثالًا عمليًا.

لنقل بأننا نكتب "حزمة"، أي مجلدًا فيه وحدات كثيرة وأردنا تصدير بعض ميزاتها إلى الخارج (تتيح لنا الأدوات مثل NPM نشر هذه الحزم وتوزيعها)، ونعلم أيضًا أن الكثير من وحداتها ما هي إلّا وحدات مُساعِدة

يمكن أن تكون بنية الملفات هكذا:

```
auth/
    index.js
    user.js
    helpers.js
    tests/
        login.js
    providers/
        github.js
        facebook.js
        ...
```

ونريد عرض مزايا الحزمة باستعمال نقطة واحدة (أي الملف الأساسي `auth/index.js`) لتُستعمل هكذا:

```
import {login, logout} from 'auth/index.js'
```

الفكرة هي عدم السماح للغرباء (أي المطوّرين مستعملي الحزمة) بالتعديل على البنية الداخلية والبحث عن الملفات داخل مجلد الحزمة. نريد تصدير المطلوب فقط في `auth/index.js` وإخفاء الباقي عن أعين المتطفّلين.

نظرًا لكون الوظيفة الفعلية المصدّرة مبعثرة بين الحزمة، يمكننا استيرادها إلى `auth/index.js` وتصديرها من هنالك أيضًا:

```
//    auth/index.js

// login/logout إستورد وصدِرهن مباشرةً
import {login, logout} from './helpers.js';
export {login, logout};

// استورد الملف المبدئي كـ User وصدره من جديد
import User from './user.js';
export {User};
...
```

والآن يمكن لمستخدمي الحزمة الخاصة بنا استيرادها هكذا:

```
import {login} from "auth/index.js".
```

إن الصياغة `export ... from ...` ماهي إلا اختصار للاستيراد والتصدير:

```
//    auth/index.js
// إِستورد login/logout وصدِرهن مباشرةً
export {login, logout} from './helpers.js';

// استورد الملف المبدئي كـ User وصدره من جديد
export {default as User} from './user.js';
...
```

ا. إعادة تصدير التصديرات المبدئية

يحتاج التصدير المبدئي لمعالجة منفصلة عند إعادة التصدير. لنفترض أن لدينا `user.js`، ونود إعادة تصدير الصنف `User` منه:

```
//    user.js
export default class User {
  // ...
}
```

لن تعمل التعليمة `export User from './user.js'`. ما الخطأ الذي حدث؟ ولكن هذا الخطأ في صياغة! لإعادة تصدير الملفات المصدرة افتراضيًا، علينا كتابة `export {default as User}`، كما في المثال أعلاه.

تعيد التعليمة `export * from './user.js'` تصدير التصديرات الّتي لها أسماء فقط، ولكنها تتجاهل التصديرات المبدئية.

إذا رغبنا في إعادة تصدير التصديرات المبدئية والتي لها أسماء أيضًا، فسنحتاج إلى العبارتين:

```
export * from './user.js'; // لإعادة تصدير التصديرات الّتي لها أسماء
export {default} from './user.js'; // لإعادة تصدير التصديرات المبدئية
```

هذه الغرابة في طريقة إعادة تصدير التصديرات المبدئية هي من أحد الأسباب لجعل بعض المطورين لا يحبونها.

13.2.8 الخلاصة

والآن سنراجع جميع أنواع طرق التصدير `export` التي تحدثنا عنها في هذا الفصل والفصول السابقة. تحقق من معلوماتك بقراءتك لهم وتذكر ما تعنيه كلٌّ واحدةٍ منهم:

- قبل التعريف عن صنف / دالّة / ..:
  - `export [default] class/function/variable ...`
- تصدير مستقل:
  - `export {x [as y], ...}`.
- إعادة التصدير:
  - `export {x [as y], ...} from "module"`
  - `export * from "module"` (لا يُعيد التصدير المبدئي).
  - `export {default [as y]} from "module"` (يعيد التصدير المبدئي).

استيراد:

- الصادرات التي لها أسماء من الوِحدة:
  - `import {x [as y], ...} from "module"`
- التصدير المبدئي:
  - `import x from "module"`
  - `import {default as x} from "module"`
- استيراد كل شيء:
  - `import * as obj from "module"`
- استيراد الوحدة (وشغّل شيفرتها البرمجية)، ولكن لا تُسندها لمتغير:
  - `import "module"`

لا يهم مكان وضع عبارات (تعليمات) `import/export` سواءً في أعلى أو أسفل السكربت فلن يغير ذلك أي شيء. لذا تقنيًا تعدُّ هذه الشيفرة البرمجية لا بأس بها:

```
sayHi();
// ...
import {sayHi} from './say.js'; // إستورد في نهاية الملف
```

عمليًا عادة ما تكون تعليمات الاستيراد في بداية الملف فقط لتنسيق أفضل للشيفرة. لاحظ أن تعليمتي import/export لن يعملا إن كانا في داخل جملة شرطية.

لن يعمل الاستيراد الشرطي مثل هذا المثال:

```
if (something) {
  import {sayHi} from "./say.js"; // Error: import must be at top level
}
```

.. ولكن ماذا لو احتجنا حقًا لاستيراد شيء ما بشروط معينة؟ أو في وقتٍ ما؟ مثل: تحميل الوِحدة عند الطلب، عندما تكون هناك حاجة إليها حقًا؟

سنرى الاستيراد الديناميكي في الفصل التالية.

13.3 استيراد الوحدات ديناميكيًا

إن طريقة الاستيراد والتصدير التي تحدثنا عنهـا في الفصـل السـابق، تصـدير الوحـدات واسـتيرادها تـدعى بالطريقة "الثابتة". إذ أنّ صياغتها بسيطة وصارمة للغاية.

دعنا في البداية نوضح بعض الأمور، أولًا، لا يمكننا إنشاء أي وسطاء للتعليمة `import` إنشـاءً ديناميكيًـا. إذ يجب أن يكون مسار الوِحدة سلسلة أولية (primitive)، ولا يجب أن تكون استدعاءً لدالة معينة. فهذا لن ينجح:

```
import ... from getModuleName(); // خطأ، مسموح استخدام السلاسل فقط
```

ثانيًا، لا يمكننا استخدام الاسـتيراد المشـروط (في حـال حـدوث شـرط معين اسـتورد مكتبـة) أو الاسـتيراد أثناء التشغيل:

```
if(...) {
  import ...; // !خطأ غير مسموح بذلك
}

{

  import ...;  // خطأ، لا يمكننا وضع تعليمة import في أي كتلة
}
```

وذلك لأن تعليمتي `import`/`export` تهدفان لتوفيـر العمـود الفقـري لبنيـة الشـيفرة. وهـذا أمـر جيـد، إذ يمكننا تحليل بنية الشيفرة، وتجميع الوحدات وتحزيمها في ملـف واحـد من خلال أدوات خاصـة، ويمكننـا أيضًـا إزالة عمليات التصدير غير المستخدمة (هزّ الشجرة -بهدف سقوط الأوراق اليابسة). هذا ممكن فقط لأن هيكليـة التعليمتين `import`/`export` بسيطة وثابتة.

ولكن كيف يمكننا استيراد وِحدة استيرادًا ديناميكيًا بحسب الطلب؟

13.3.1 تعبير الاستيراد

يُحمّل التعبير `import (module)` الوِحدة ويُرجع وعدًا، والّذي يُستبدل بكائن الوِحدة، ويحتوي هذا الأخير على كافة عمليات التصدير الخاصة بالكائن. ويُستدعى من أي مكان في الشيفرة البرمجية.

ونستطيع استخدامه ديناميكيًا في أي مكان من الشيفرة البرمجية، فمثلًا:

```
let modulePath = prompt("Which module to load?");
```

```
import(modulePath)
  .then(obj => <module object>)
  .catch(err => <loading error, e.g. if no such module>)
```

أو يمكننا استخدام `let module = await import(modulePath)` إن كنا بداخل دالّة غير متزامنة.

فمثلًا، ليكن لدينا الوِحدة التالية `say.js`:

```
//    say.js
export function hi() {
  alert(`Hello`);
}

export function bye() {
  alert(`Bye`);
}
```

... ثم يكون الاستيراد الديناميكي هكذا:

```
let {hi, bye} = await import('./say.js');

hi();
bye();
```

أو إذا كان `say.js` يحتوي على التصدير المبدئي:

```
//    say.js
export default function() {
  alert("Module loaded (export default)!");
}
```

... بعد ذلك، من أجل الوصول إليه، يمكننا استخدام الخاصية `default` لكائن الوِحدة:

```
let obj = await import('./say.js');
let say = obj.default;
// let {default: say} = await import('./say.js'); أو بسطرٍ واحد هكذا:

say();
```

إليك المثال الكامل:

- الملف `say.js`:

```
export function hi() {
  alert(`Hello`);
}

export function bye() {
  alert(`Bye`);
}

export default function() {
  alert("Module loaded (export default)!");
}
```

- الملف `index.html`:

```
<!doctype html>
<script>
  async function load() {
    let say = await import('./say.js');
    say.hi(); // Hello!
    say.bye(); // Bye!
    say.default(); // Module loaded (export default)!
  }
</script>
<button onclick="load()">Click me</button>
```

وإليك النتائج في هذا المثال الحي.

> لاحظ كيف تعمل عمليات الاستيراد الديناميكية في السكربتات العادية، ولا تتطلب استعمال `script type ="module"`.
>
> لاحظ أيضًا على الرغم من أن تعليمة `import()` تشبه طريقة استدعاء دالّة، إلا أنها صياغة خاصة ويحدث هذا التشابه فقط لاستخدام الأقواس (على غرار `super()`). لذلك لا يمكننا نسخ تعليمة `import` إلى متغير، أو استخدام `call/apply` معها. إذ هي ليست دالّة.

14. مواضيع متفرقة

يتضمن هذا الفصل الأقسام التالية:

1. الوسيط Proxy والمنعكس Reflect
2. الدالة Eval لتنفيذ الشيفرة البرمجية
3. تقنية Currying
4. النوع المرجعي Reference
5. الأعداد الكبيرة BigInt

14.1 الوسيط Proxy والمنعكس Reflect

يغلّف كائن الوكيل Proxy كائنًا آخر ويعترض عملياته مثل: خاصيات القراءة أو الكتابة وغيرهما. ويعالجها اختياريًا بمفرده، أو يسمح بشفافية للكائن التعامل معها بنفسه. تستخدم العديد من المكتبات وأُطر عمل المتصفح الوسطاء. وسنرى في هذا الفصل العديد من تطبيقاته العملية.

14.1.1 الوسيط Proxy

صياغته:

```
let proxy = new Proxy(target, handler)
```

- target: وهو الكائن الّذي سنغلّفه يمكن أي يكون أي شيء بما في ذلك التوابع.
- Handler: لإعداد الوسيط: وهو كائن يحتوي على "الاعتراضات"، أي دوالّ اعتراض العمليات. مثلًا: الاعتراض get لاعتراض خاصية القراءة في الهدف target، الاعتراض set لاعتراض خاصية الكتابة في الهدف target، وهكذا.

سيراقب الوسيط العمليات فإن كان لديه اعتراض مطابق في المعالج handler للعملية المنفذة عند الهدف، فعندها سينفذ الوسيط هذه العملية ويعالجها وإلا ستُنفذ هذه العملية من قِبل الهدف نفسه.

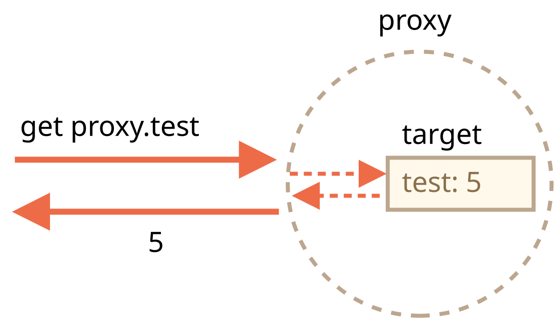
لنأخذ مثالًا بسيطًا يوضح الأمر، لننشئ وسيطًا بدون أي اعتراضات:

```
let target = {};
let proxy = new Proxy(target, {}); // المعالج فارغ

proxy.test = 5; // (1) الكتابة على الوسيط
alert(target.test); // النتيجة: 5، ظهرت الخاصية في كائن الهدا !

alert(proxy.test); // (2) النتيجة: 5، يمكننا قراءتها من الوسيط أيضًا
for(let key in proxy) alert(key); // (3) النتيجة: test, عملية التكرار تعمل
```

انطلاقًا من عدم وجود اعتراضات سيُعاد توجيه جميع العمليات في الوسيط proxy إلى كائن الهدف target.

1. تحدد عملية الكتابة proxy.test= القيمة عند الهدف target، لاحظ السطر (1).
2. تعيد عملية القراءة proxy.test القيمة من عند الهدف target. لاحظ السطر (2).
3. تعيد عملية التكرار على الوسيط proxy القيم من الهدف target. لاحظ السطر (3).

كما نرى، الوسيط proxy في هذه الحالة عبارة عن غِلاف شفاف حول الكائن الهدف target.



لنُضف بعض الاعتراضات من أجل تفعيل المزيد من الإمكانيات. ما الذي يمكننا اعتراضه؟

بالنسبة لمعظم العمليات على الكائنات، هناك ما يسمى "الدوال الداخلية" في مواصفات القياسية في اللغة، والّتي تصف كيفية عمل الكائن عند أدنى مستوى. فمثلًا الدالة الداخلية [[Get]] لقراءة خاصية ما، والدالة الداخلية [[Set]] لكتابة خاصية ما، وهكذا. وتستخدم هذه الدوال من قبل المواصفات فقط، ولا يمكننا استدعاؤها مباشرة من خلال أسمائها.

اعتراضات الوسيط تتدخلّ عند استدعاء هذه الدوالّ. وهي مدرجة في المواصفات القياسية للوسيط وفي الجدول أدناه. يوجد اعتراض مخصص (دالّة معالجة) لكل دالّة داخلية في هذا الجدول: يمكننا إضافة اسم الدالّة إلى المعالج handler من خلال تمريرها كوسيط new Proxy لاعتراض العملية.

ستعمل عند	الدالة المعالجة	الدالة الداخلية
قراءة خاصية	get	[[Get]]
الكتابة على خاصية	set	[[Set]]
التأكد من وجود خاصية ما in	has	[[HasProperty]]
عملية الحذف delete	deleteProperty	[[Delete]]
استدعاء تابِع	apply	[[Call]]
عملية الإنشاء -الباني- new	construct	[[Construct]]
Object.getPrototypeOf	getPrototypeOf	[[GetPrototypeOf]]
Object.setPrototypeOf	setPrototypeOf	[[SetPrototypeOf]]
Object.isExtensible	isExtensible	[[IsExtensible]]
Object.preventExtensions	preventExtensions	[[PreventExtensions]]
Object.defineProperty Object.defineProperties	defineProperty	[[DefineOwnProperty]]
Object.getOwnPropertyDescriptor for..in Object.keys/values/entries	getOwnPropertyDescriptor	[[GetOwnProperty]]

ستعمل عند	الدالة المعالجة	الدالة الداخلية
Object.getOwnPropertyNames Object.getOwnPropertySymbols for..in Object/keys/values/entries	ownKeys	[[OwnPropertyKeys]]

تفرض علينا لغة جافاسكربت بعض الثوابت - الشروط الواجب تحقيقها من قِبل الدوال الداخلية والاعتراضات. معظمها للقيم المُعادة من الدوال:

- يجب أن تعيد الدالّة `[[Set]]` النتيجة `true` إذا عُدلت القيمة بنجاح، وإلا ستُعيد `false`.
- يجب أن تعيد الدالّة `[[Delete]]` النتيجة `true` إذا حُذفت القيمة بنجاح، وإلا ستُعيد `false`.
- …وهكذا، سنرى المزيد من الأمثلة لاحقًا.

هنالك بعض الثوابت الأخرى، مثل:

- يجب أن تُعيد الدالة `[[GetPrototypeOf]]` المطبقة على كائن الوسيط نفس القيمة التّي ستُعيدها الدالة `[[GetPrototypeOf]]` المطبقة على كائن الهدف لكائن الوسيط، بتعبير آخر، يجب أن تعرض دائمًا قراءة النموذج الأولي لكائن الوسيط نفس قراءة النموذج الأولي لكائن الهدف.

يمكن للاعتراضات التدخل في هذه العمليات ولكن لابد لها من اتباع هذه القواعد.

تضمن هذه الثوابت السلوك الصحيح والمُتسق لمميّزات اللغة. وجميع هذه الثوابت موجودة في المواصفات القياسية للغة. وغالبًا لن تكسرَهم إن لم تنفذّ شيئًا غريبًا.

لنرى كيف تعمل في مثال عملي.

14.1.2 إضافة اعتراض للقيم المبدئية

تعدُّ خاصيات القراءة / الكتابة من أكثر الاعتراضات شيوعًا.

لاعتراض القراءة، يجب أن يكون لدى المعالج `handler` دالّة `get(target, property, receiver)`.

وتُشغّل عند قراءة خاصية ما، وتكون الوسطاء:

- `target` - هو كائن الهدف، والذي سيمرر كوسيط أول لِـ `new Proxy`،
- `property` - اسم الخاصية،
- `receiver` - إذا كانت الخاصية المستهدفة هي الجالِب (getter)، فإن `receiver` هو الكائن الذي سيُستخدم على أنه بديل للكلمة المفتاحية `this` في الاستدعاء. عادةً ما يكون هذا هو كائن `proxy` نفسه (أو كائن يرث منه، في حال ورِثنا من الوسيط). لا نحتاج الآن هذا الوسيط، لذلك سنشرحها بمزيد من التفصيل لاحقًا.

لنستخدم الجالِب get لجلب القيم الافتراضية لكائن ما. سننشئ مصفوفة رقمية تُعيد القيمة 0 للقيم غـير الموجودة. عادةً عندما نحاول الحصول على عنصر من مصفوفة، وكان هـذا العنصر غـير موجـود، سنحصـل على النتيجة غير معرّف undefined، لكننا هنا سنُغلف المصفوفة العادية داخل الوسيط والّـذي سـيعترضُ خاصـية القراءة ويعيد 0 إذا لم تكُ الخاصية المطلوبة موجودة في المصفوفة:

```
let numbers = [0, 1, 2];

numbers = new Proxy(numbers, {
  get(target, prop) {
    if (prop in target) {
      return target[prop];
    } else {
      return 0; // القيمة الافتراضية
    }
  }
});

alert( numbers[1] ); // 1
alert( numbers[123] ); // 0 (هذا العنصر غير موجود)
```

كما رأينا، من السهل جدًا تنفيذ ذلك باعتراض الجـالِب get. يمكننـا اسـتخدام الوسـيط proxy لتنفيـذ أي منطق للقيم "الافتراضية".

تخيل أن لدينا قاموسًا (يربط القاموس مفاتيح مع قيم على هيئة أزواج) يربط العبارات مع ترجمتها:

```
let dictionary = {
  'Hello': 'Hola',
  'Bye': 'Adiós'
};

alert( dictionary['Hello'] ); // Hola
alert( dictionary['Welcome'] ); // undefined
```

مبدئيًا إن حاولنا الوصول إلى عبـارة غـير موجـودة في القـاموس فسـيُعيد غـير معـرّف undefined، ولكن عمليًا إن ترك العبارة بدون ترجمة أفضل من undefined، لذا لنجعلها تُعيـد العبـارة بـدون مترجمـة بـدلاً من undefined. لتحقيق ذلك، سنغلّف القاموس dictionary بالوسيط ليعترض عمليات القراءة:

```
let dictionary = {
  'Hello': 'Hola',
  'Bye': 'Adiós'
};

dictionary = new Proxy(dictionary, {
  get(target, phrase) { // اعترض خاصية القراءة من القاموس
    if (phrase in target) { // إن كانت موجودة في القاموس
      return target[phrase]; // أعد الترجمة
    } else {
      // وإلا أعدها بدون ترجمة
      return phrase;
    }
  }
});

// ابحث عن عبارة عشوائية في القاموس!
// بأسوء حالة سيُعيد العبارة غير مترجمة
alert( dictionary['Hello'] ); // Hola
alert( dictionary['Welcome to Proxy']); // Welcome to Proxy (يعيد العبارة بدون ترجمة)
```

لاحظ كيف أن الوسيط يعد الكتابة على المتغير الافتراضي:

```
dictionary = new Proxy(dictionary, ...);
```

يجب على الوسيط استبدال كائن الهدف بالكامل في كل مكان. ولا يجب أن يشير أي شيء للكائن الهدف بعد أن يستبدلهُ الوسيط. وإلا فذلك سيُحدثُ فوضى عارمة.

14.1.3 تدقيق المدخلات باعتراض الضابِط set

لنفترض أننا نريد إنشاء مصفوفة مخصصة للأرقام فقط. وإن أضيفت قيمة من نوع آخر، يجب أن يظهر خطأ. يُشغّل اعتراض الضابِط `set` عند الكتابة على الخاصية.

```
set(target, property, value, receiver)
```

- `target`: الكائن الهدف، الذي سنمرره كوسيط أول `new Proxy`،
- `property` : اسم الخاصية،

- value: قيمة الخاصية،
- receiver: مشابه للدالّة get، ولكن هنا لضبط الخاصيات فقط.

يجب أن يُعيد الاعتراض set القيمة true إذا نجح ضبط القيمة في الخاصية، وإلا يعيد القيمة false (يُشغّل خطأ من نوع TypeError).

لنأخذ مثالًا عن كيفية استخدامها للتحقق من القيم الجديدة:

```
let numbers = [];

numbers = new Proxy(numbers, { // (*)
  set(target, prop, val) { // لاعتراض خاصية ضبط القيمة
    if (typeof val == 'number') {
      target[prop] = val;
      return true;
    } else {
      return false;
    }
  }
});

numbers.push(1); // أضيفت بنجاح
numbers.push(2); // أضيفت بنجاح
alert("Length is: " + numbers.length); // 2

numbers.push("test"); // TypeError (الدالّة 'set' في الوسيط أعادت القيمة false)

alert("This line is never reached (error in the line above)");
```

لاحظ أن الدوال المدمجة للمصفوفات ما تزال تعمل! إذ تضاف القيم عن طريق push. فتزداد خاصية length تلقائيًا عند إضافة هذه القيم. لذا فإن الوسيط لن يكسر أي شيء.

يجب علينا ألا نعيد كتابة الدوال المُدمجة للمصفوفات الّتي تضيف القيم مثل push و unshift، وما إلى ذلك، إن كان هدفنا إضافة عمليات تحقق، لأنهم يستخدمون داخليًا دالّة [[Set]] والّتي ستُعترضُ من قِبل الوسيط. إذن الشيفرة نظيفة ومختصرة.

> لا تنسَ أن تعيد القيمة `true` عند نجاح عملية الكتابة، فكما ذكر أعلاه، هناك ثوابت ستعقّد، فبالنسبة للضابط `set`، يجب أن يُرجِع`true` عند نجاح عملية الكتابة، وإذا نسينا القيام بذلك أو إعادة أي قيمة زائفة، فإن العملية تؤدي إلى خطأ `TypeError`.

14.1.4 التكرار باستخدام ownKeys و getOwnPropertyDescriptor

إن الدالة `Object.keys` وحلقة التكرار `for..in` ومعظم الطرق الأخرى الّتي تستعرض خاصيات الكائن، وتستخدم الدالّة الداخلية `[[OwnPropertyKeys]]` للحصول على قائمة بالخاصيات يمكننا اعتراضها من خلال `ownKeys`.

تختلف هذه الدوال وحلقات التكرار على الخاصيات تحديدًا في:

- `Object.getOwnPropertyNames(obj)`: تُعيد الخاصيات غير الرمزية.
- `Object.getOwnPropertySymbols(obj)`: تُعيد الخاصيات الرمزية.
- `Object.keys/values()`: تعيد الخاصيات/القيم غير الرمزية والتي تحمل راية قابلية الإحصاء `enumerable` (شرحنا في فصل سابق ما هي رايات الخواص وواصفاتها يمكنك الاطلاع عليه لمزيد من التفاصيل).
- حلقات `for..in`: تمرّ على الخاصيات غير الرمزية التي تحمل راية قابلية الإحصاء `enumerable`، وكذلك تمرّ على خاصيات النموذج الأولي (prototype).

لكن كلهم يبدؤون بهذه القائمة.

في المثال أدناه، نستخدم اعتراض `ownKeys` لجعل حلقة `for..in` تمر على `user`، وكذلك `Object.keys` و `Object.values`، وتتخطى الخاصيات التي تبدأ بشرطة سفلية _:

```
let user = {
  name: "Ahmad",
  age: 30,
  _password: "***"
};

user = new Proxy(user, {
  ownKeys(target) {
    return Object.keys(target).filter(key => !key.startsWith('_'));
  }
});
```

```
// _password سيزيل الخاصّية "ownKeys" الاعتراض
for(let key in user) alert(key); // name, then: age

// نفس التأثير سيحدث على هذه التوابع:
alert( Object.keys(user) ); // name,age
alert( Object.values(user) ); // Ahmad,30
```

حتى الآن، لا يزال مثالنا يعمل وفق المطلوب.

على الرغم من ذلك، إذا أضفنا خاصية ما غير موجودة في الكائن الأصلي وذلك بإرجاعها من خلال الوسيط، فلن يعيدها التابع `Object.keys`:

```
let user = { };

user = new Proxy(user, {
  ownKeys(target) {
    return ['a', 'b', 'c'];
  }
});

alert( Object.keys(user) ); // <empty>
```

هل تسأل نفسك لماذا؟ السبب بسيط: تُرجع الدالّة `Object.keys` فقط الخاصيات الّتي تحمل راية قابلية الإحصاء `enumerable`. وهي تحقق من رايات الخاصيات لديها باستدعاء الدالّة الداخلية `[[GetOwnProperty]]` لكل خاصية للحصول على واصِفها. وهنا، نظرًا لعدم وجود الخاصّية، فإن واصفها فارغ، ولا يوجد راية قابلية الإحصاء `enumerable`، وبناءً عليه تخطت الدالّة الخاصّية.

من أجل أن ترجع الدالّة `Object.keys` الخاصية، فيجب أن تكون إما موجودة في الكائن الأصلي، وتحمل راية قابلية الإحصاء `enumerable`، أو يمكننا وضع اعتراض عند استدعاء الدالة الداخلية `[[GetOwnProperty]]` (اعتراض `getOwnPropertyDescriptor` سيحقق المطلوب)، وإرجاع واصف لخاصية قابلية الإحصاء بالإيجاب هكذا `enumerable: true`.

إليك المثال ليتوضح الأمر:

```
let user = { };

user = new Proxy(user, {
  ownKeys(target) { // يستدعى مرة واحدة عند طلب قائمة الخاصيات
```

```
    return ['a', 'b', 'c'];
  },

  getOwnPropertyDescriptor(target, prop) { // تستدعى من أجل كلّ خاصّية
    return {
      enumerable: true,
      configurable: true
      /* ...لها المناسبة القيم مع أخرى رايات إضافة يمكننا..." */
    };
  }

});

alert( Object.keys(user) ); // a, b, c
```

نعيد مرة أخرى: نضيف اعتراض `[[GetOwnProperty]]` إذا كانت الخاصّية غير موجودة في الكائن الأصلي.

14.1.5 الخاصيات المحمية والاعتراض "deleteProperty" وغيره

هناك إجماع كبير في مجتمع المطورين بأن الخاصيات والدوال المسبوقة بشرطة سفلية _ هي للاستخدام الداخلي. ولا ينبغي الوصول إليها من خارج الكائن. هذا ممكن تقنيًا:

```
let user = {
  name: "Ahmad",
  _password: "secret"
};
// لاحظ أن في الوضع الطبيعي يمكننا الوصول لها
alert(user._password); // secret
```

لنستخدم الوسيط لمنع الوصول إلى الخاصيات الّتي تبدأ بشرطة سفلية _. سنحتاج إلى الاستثناءات التالية:

- `get` لإلقاء خطأ عند قراءة الخاصية
- `set` لإلقاء خطأ عند الكتابة على الخاصية
- `deleteProperty` لإلقاء خطأ عند حذف الخاصية
- `ownKeys` لاستبعاد الخصائص التي تبدأ بشرطة سفلية _ من حلقة `for..in`، والدوال التي تستعرض الخاصيات مثل: `Object.keys`

هكذا ستكون الشيفرة:

```
let user = {
  name: "Ahmad",
  _password: "***"
};

user = new Proxy(user, {
  get(target, prop) {
    if (prop.startsWith('_')) {
      throw new Error("Access denied");
    }
    let value = target[prop];
    return (typeof value === 'function') ? value.bind(target) : value;
// (*)
  },
  set(target, prop, val) { // لاعتراض عملية الكتابة على الخاصّية
    if (prop.startsWith('_')) {
      throw new Error("Access denied");
    } else {
      target[prop] = val;
      return true;
    }
  },
  deleteProperty(target, prop) { // لاعتراض عملية حذف الخاصّية
    if (prop.startsWith('_')) {
      throw new Error("Access denied");
    } else {
      delete target[prop];
      return true;
    }
  },
  ownKeys(target) { // لاعتراض رؤية الخاصية من خلال الحلقات أو الدوالّ
    return Object.keys(target).filter(key => !key.startsWith('_'));
  }
});
```

```
// _password سيمنعُ قراءة الخاصّية "get" الاعتراض
try {
  alert(user._password); // Error: Access denied
} catch(e) { alert(e.message); }
// _password سيمنع الكتابة على الخاصّية "set" الاعتراض
try {
  user._password = "test"; // Error: Access denied
} catch(e) { alert(e.message); }
// _password سيمنعُ حذف الخاصّية "deleteProperty" الاعتراض
try {
  delete user._password; // Error: Access denied
} catch(e) { alert(e.message); }
// الحلقات في _password سيمنعُ إمكانية رؤية الخاصّية "ownKeys" الاعتراض
for(let key in user) alert(key); // name
```

لاحظ التفاصيل المهمة في الاعتراض `get`، وذلك في السطر `(*)`:

```
get(target, prop) {
  // ...
  let value = target[prop];
  return (typeof value === 'function') ? value.bind(target) : value;
// (*)
}
```

لماذا نحتاج لدالّة لاستدعاء `value.bind(target)`؟ وسبب ذلك هو أن دوال الكائن مثل: `user.checkPassword()` يجب تحافظ على إمكانية الوصول للخاصّية `_password`:

```
user = {
  // ...
  checkPassword(value) {
    // _password دوال الكائن يجب أن تحافظ على إمكانية الوصول للخاصية
    return value === this._password;
  }
}
```

تستدعي الدالة `user.checkPassword()` وسيط الكائن `user` ليحلّ محلّ `this` (الكائن قبل النقطة يصبح بدل `this`)، لذلك عندما نحاول الوصول إلى الخاصّية `this._password`، سيُنشّط الاعتراض `get` (والّذي يُشغلّ عند قراءة خاصية ما) ويلقي الخطأ.

لذلك نربط سياق الدوال الخاصة بالكائن مع دوالّ الكائن الأصلي، أيّ الكائن الهدف `target` لاحظ السطر `(*)`. بعد ذلك، الاستدعاءات المستقبلية ستستخدم `target` بدلّ `this`، دون الاعتراضات.

هذا الحل ليس مثاليًا ولكنه يفي بالغرض، ولكن يمكن لدالّةٍ ما أن تُمرّر الكائن غير المغَلّف بالوسيط (أي الكائن الأصلي) إلى مكان آخر، وبذلك ستخرّب الشيفرة ولن نستطع الإجابة على أسئلة مثل: أين يستخدم الكائن الأصلي؟ وأين يستخدم الكائن الوسيط؟

أضف إلى ذلك، يمكن للكائن أن يغلف بأكثر من وسيط (تضيف عدة وسطاء "تعديلات" مختلفة على الكائن الأصلي)، وإن مرّرنا كائن غير مغلّف بالوسيط إلى دالّة ما، ستكون هناك عواقب غير متوقعة، لذا، لا ينبغي استخدام هذا الوسيط في كل الحالات.

> **الخاصيات الخاصة بالصنف**
>
> تدعم محركات جافاسكربت الحديثة الخاصيات الخاصّة بالأصناف، وتكون مسبوقة بـ #. تطرقنا لها سابقًا في فصل الخصائص والتوابع الخاصّة والمحمية. وبدون استخدام أيّ الوسيط.
> على الرغم من ذلك هذه الخاصيات لها مشاكلها الخاصة أيضًا. وتحديدًا، مشكلة عدم إمكانية توريث هذه الخاصيات.

14.1.6 استخدام الاعتراض In range مع has

لنرى مزيدًا من الأمثلة. لدينا الكائن `range`:

```
let range = {
  start: 1,
  end: 10
};
```

نريد استخدام المعامل `in` للتحقق من أن الرقم موجود في `range`. إن الاعتراض `has` سيعترض استدعاءات `in`.

```
has(target, property)
```

- `target`: هو الكائن الهدف، الذي سيمرر كوسيط أولnew Proxy،
- `property`: اسم الخاصية

إليك المثال:

```
let range = {
  start: 1,
```

```
  end: 10
};

range = new Proxy(range, {
  has(target, prop) {
    return prop >= target.start && prop <= target.end;
  }
});

alert(5 in range); // true
alert(50 in range); // false
```

تجميلٌ لغويٌّ رائع، أليس كذلك؟ وتنفيذه بسيط جدًا.

14.1.7 تغليف التوابع باستخدام apply

يمكننا أيضًا تغليف دالّة ما باستخدام كائن الوسيط.

يعالج الاعتراض `apply(target, thisArg, args)` استدعاء كائن الوسيط كتابع:

- `target`: الكائن الهدف (التابع - أو الدالّة - هي كائن في لغة جافا سكربت)،
- `thisArg`: قيمة `this`.
- `args`: قائمة الوسطاء.

فمثلًا، لنتذكر المُزخرف `delay(f, ms)`، الذي أنشأناه في فصل المزخرفات والتمرير. في تلك الفصلة أنشأناه بدون استخدام الوسيط. عند استدعاء الدالة `delay(f, ms)` ستُعيد دالّة أخرى، والّتي بدورها ستوجه جميع الاستدعاءات إلى `f` بعد `ms` ملّي ثانية.

إليك التطبيق المثال بالاعتماد على التوابع:

```
function delay(f, ms) {
  // يعيد المغلف والذي بدوره سيوجه الاستدعاءات إلى f بعد انتهاء مهلة زمنية معينة
  return function() { // (*)
    setTimeout(() => f.apply(this, arguments), ms);
  };
}
```

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

// بعد عملية التغليف استدعي الدالّة sayHi بعد 3 ثواني
sayHi = delay(sayHi, 3000);

sayHi("Ahmad"); // Hello, Ahmad! (بعد 3 ثواني)
```

كما رأينا، غالبًا ستعمل هذه الطريقة وفق المطلوب. تُنفذّ الدالّة المغلفة في السطر `(*)` استدعاءً بعد انتهاء المهلة. لكن دالة المغلف لا تعيد توجيه خاصيات القراءة أو الكتابة للعمليات أو أيّ شيء آخر. بعد عملية التغليف، يُفقدُ إمكانية الوصول لخاصيات التوابع الأصلية، مثل: `name` و`length` وغيرها:

```
function delay(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(user) {
  alert(`Hello, ${user}!`);
}

alert(sayHi.length); // 1 (طول الدالة هو عدد الوسطاء في تعريفها)
sayHi = delay(sayHi, 3000);
alert(sayHi.length); // 0 (إن عدد وسطاء عند تعريف المغلّف هو 0)
```

في الحقيقة إن إمكانيات الوسيط `Proxy` أقوى بكثير من ذلك، إذ إنه يعيد توجيه كل شيء إلى الكائن الهدف. لنستخدم الوسيط `Proxy` بدلاً من الدالّة المُغلِّفة:

```
function delay(f, ms) {
  return new Proxy(f, {
    apply(target, thisArg, args) {
      setTimeout(() => target.apply(thisArg, args), ms);
    }
  });
```

```
}

function sayHi(user) {
  alert(`Hello, ${user}!`);
}

sayHi = delay(sayHi, 3000);
alert(sayHi.length); // 1 (*) سيعيد الوسيط توجيه عملية "get length" إلى الهدف
sayHi("Ahmad"); // Hello, Ahmad! (بعد 3 ثواني)
```

نلاحظ أن النتيجة نفسها، ولكن الآن ليس مجرد استدعاءات فقط، وإنما كل العمليات في الوسيط يُعاد توجيها إلى التوابع الأصلية. لذلك سيعيد الاستدعاء `sayHi.length` النتيجة الصحيحة بعد التغليف في السطر `(*)`. وبذلك حصلنا على مُغلِّف أغنى بالمميزات من الطريقة السابقة.

هنالك العديد من الاعتراضات الأخرى: يمكنك العودة لبداية الفصل لقراءة القائمة الكاملة للاعتراضات. كما أن طريقة استخدامها مشابه كثيرًا لما سبق.

14.1.8 الانعكاس

الانعكاس `Reflect` هو عبارة عن كائن مضمّن في اللغة يبسط إنشاء الوسيط `Proxy`.

ذكرنا سابقًا أن الدوالّ الداخلية، مثل: `[[Get]]` و`[[Set]]` وغيرها مخصصة للاستخدام في مواصفات اللغة فقط، ولا يمكننا استدعاؤها مباشرة.

يمكن لكائن المنعكس `Reflect` من فعل ذلك إلى حد ما. إذ أن الدوالّ الخاصة به عبارة مُغلِّفات صغيرة حول الدوالّ الداخلية.

فيما يلي أمثلة للعمليات واستدعاءات المنعكس `Reflect` الّتي ستُؤدي نفس المهمة:

الدالة الداخلية	الدالة المقابلة في المنعكس	العملية
`[[Get]]`	`Reflect.get(obj, prop)`	`obj[prop]`
`[[Set]]`	`Reflect.set(obj, prop, value)`	`obj[prop] = value`
`[[Delete]]`	`Reflect.deleteProperty(obj, prop)`	`delete obj[prop]`
`[[Construct]]`	`Reflect.construct(F, value)`	`new F(value)`
…	…	…

فمثلًا:

```
let user = {};
```

```
Reflect.set(user, 'name', 'Ahmad');

alert(user.name); // Ahmad
```

تحديدًا، يتيح لنا المنعكس Reflect استدعاء العمليات (new, delete...) كتوابع هكذا (Reflect.construct,Reflect.deleteProperty, ...). وهذه الإمكانيات مثيرة للاهتمام، ولكن هنالك شيء آخر مهم.

لكل دالّة داخلية، والّتي يمكننا تتبعها من خلال الوسيط Proxy، يوجد دالّة مقابلة لها في المنعكس Reflect، بنفس الاسم والوسطاء أي مشابه تمامًا للاعتراض في الوسيط Proxy. لذا يمكننا استخدام المنعكس Reflect لإعادة توجيه عملية ما إلى الكائن الأصلي.

في هذا المثال، سيكون كلًا من الاعتراضين get وset شفافين (كما لو أنهما غير موجودين) وسيُوجهان عمليات القراءة والكتابة إلى الكائن، مع إظهار رسالة:

```
let user = {
  name: "Ahmad",
};

user = new Proxy(user, {
  get(target, prop, receiver) {
    alert(`GET ${prop}`);
    return Reflect.get(target, prop, receiver); // (1)
  },
  set(target, prop, val, receiver) {
    alert(`SET ${prop}=${val}`);
    return Reflect.set(target, prop, val, receiver); // (2)
  }
});

let name = user.name; // shows "GET name"
user.name = "Pete"; // shows "SET name=Pete"
```

في الشيفرة السابقة:

- Reflect.get: يقرأ خاصية الكائن.
- Reflect.set: يكتب خاصية الكائن، سيُعيد true إن نجحت، وإلا سيُعيد false.

أي أن كـل شـيء بسـيط: إذا كـان الاعـتراض يُعيـد توجيـه الاسـتدعاء إلى الكـائن، فيكفي اسـتدعاء `Reflect.<method>` بنفس الوسطاء.

في معظم الحـالات ، يمكننـا فعـل الشـيء نفسـه بـدون `Reflect`، على سـبيل المثـال، يمكن اسـتبدال `Reflect.get(target, prop, receiver)` بـ `target[prop]`. يوجد فروقٍ بينهم ولكن لا تكاد تذكر.

ا. استخدام الوسيط مع الجالِب

لـنرى مثـالاً يوضـح لمـاذا `Reflect.get` أفضـل. وسـنرى أيضًـا سـبب وجـود الوسـيط الرابـع في دوالّ `get/set`، تحديدًا `receiver` والّذي لم نستخدمه بعد.

لدينا كائن `user` مع خاصية `_name` وجالب مخصص لها. لنُغلّفه باستخدام الوسيط:

```
let user = {
  _name: "Guest",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop];
  }
});

alert(userProxy.name); // Guest
```

إن الاعتراض `get` في هذا المثال "شفاف"، فهو يعيد الخاصية الأصلية ولا يفعل أي شيء آخـر. هـذا يكفي لمثالنا. يبدو أن كل شيء على ما يرام. لكن لنُزد تعقيد المثال قليلًا.

بعد وراثة كائن `admin` من الكائن `user`، نلاحظ السلوك الخاطئ:

```
let user = {
  _name: "Guest",
  get name() {
    return this._name;
  }
};
```

```
let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop]; // (*) target = user
  }
});

let admin = {
  __proto__: userProxy,
  _name: "Admin"
};

// Admin :نتوقع ظهور الكلمة
alert(admin.name); // (لماذا؟) Guest :النتيجة
```

إن قراءة `admin.name` يجب أن تُعيد كلمة `"Admin"` وليس `"Guest"`! ما الذي حدث؟ لعلنا أخطأنا بشيء ما في عملية الوراثة؟ ولكن إذا أزلنا كائن الوسيط، فسيكون كل شيء على ما يرام. إذًا المشكلة الحقيقية في الوسيط، تحديدًا في السطر `(*)`.

1. عندما نقرأ خاصّية الاسم `admin.name` من كائن `admin`، لا يحتوي كائن `admin` على هذه الخاصية، وبذلك ينتقل للبحث عنها في النموذج الأولي الخاص به.
2. النموذج الأولي الخاص به هو `userProxy`.
3. عند قراءة الخاصية `name` من الوسيط، سيُشغّل اعتراض `get` الخاص به، وسيُعيدihمن الكائن الأصلي هكذا `target[prop]` في السطر `(*)`.

يؤدي الاستدعاء `target [prop]`، عندما يكون قيمة `prop` هي الجالب (getter)، سيؤدي ذلك لتشغيل الشيفرة بالسياقِ `target = this`. لذلك تكون النتيجة `this._name` من الكائن الأصلي للهدف `target`، أي: من `user`.

لإصلاح مثل هذه المواقف، نحتاج إلى `receiver`، الوسيط الثالث للاعتراض `get`. إذ سيُحافظ على قيمة `this` الصحيحة لتُمرر بعد ذلك إلى الجالِب (getter). في حالتنا تكون قيمتها هي `admin`.

كيفية يمرر سياق الاستدعاء للحصول الجالِب الصحيح؟ بالنسبة للتابع العادي، يمكننا استخدام `call/apply`، ولكن بالنسبة للجالِب فلن يستدعى بهذه الطريقة.

لنستخدم الدالّة `Reflect.get` والتي يمكنها القيام بذلك. وكل شيء سيعمل مثلما نريد.

وإليك الشكل الصحيح:

```
let user = {
  _name: "Guest",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) { // receiver = admin
    return Reflect.get(target, prop, receiver); // (*)
  }
});


let admin = {
  __proto__: userProxy,
  _name: "Admin"
};

alert(admin.name); // Admin
```

الآن يحافظ `receiver` بمرجع لقيمة `this` الصحيحة (وهي `admin`)، والّتي ستُمرر من خلال `Reflect.get` في السطر `(*)`.

يمكننا إعادة كتابة الاعتراض بطريقة أقصر:

```
get(target, prop, receiver) {
  return Reflect.get(...arguments);
}
```

استدعاءات المنعكس `Reflect` لها نفس أسماء اعتراضات الوسيط وتقبل نفس وسطائه أيضًا. إذ صُمّمت خصيصًا لهذا الغرض.

لذا ، فإن استخدام المنعكس `return Reflect...` يزودنا بالأمان والثقة لتوجيه العملية والتأكد تمامًا من أننا لن ننسَ أي شيء متعلقٌ بها.

14.1.9 قيود الوسيط

لدى الوسيط طريقة فريدة لتغيير أو تعديل سلوك الكائنات الموجودة عند أدنى مستوى. ومع ذلك، هذه الطريقة ليست مثالية. وإنما هناك قيود.

ا. كائنات مضمّنة: فتحات داخلية

تستخدم العديد من الكائنات المضمنة، مثل الكائنات Map وSet وDate وPromise وغيرها ما يسمى بـ" الفتحات الداخلية ".

وهي مشابهة للخاصيات، لكنها محفوظة للأغراض داخلية فقط، وللمواصفات القياسية للغة فقط. فمثلًا تخزن Map العناصر في الفتحة الداخلية [[MapData]]. وتستطيع الدوال المُضمّنة الوصول إليها مباشرةً، وليس عبر الدوال الداخلية مثل [[Get]]/[[Set]]. لذا فإن الوسيط Proxy لا يمكنه اعتراض ذلك.

ولكن ما سبب اهتمامنا بذلك؟ إنها بكلّ الأحوال داخلية! حسنًا، إليك المشكلة. بعد أن يغلف الكائن المضمن مثل: Map باستخدام الوسيط، لن يمتلك الوسيط هذه الفتحات الداخلية، لذلك ستفشل الدوالّ المضمنة. إليك مثالًا يوضح الأمر:

```
let map = new Map();

let proxy = new Proxy(map, {});

proxy.set('test', 1); // Error
```

داخليًا ، تخزن Map جميع البيانات في الفتحة الداخلية [[MapData]]. ولكن الوسيط ليس لديه مثل هذه الفتحة. تحاول الدالّة المضمنة Map.prototype.set الوصول إلى الخاصية الداخلية this.[[MapData]]، ولكن لأن this=proxy، لا يمكن العثور عليه في الوسيط proxy مما سيؤدي لفشل العملية.

لحسن الحظ، هناك طريقة لإصلاحها:

```
let map = new Map();

let proxy = new Proxy(map, {
  get(target, prop, receiver) {
    let value = Reflect.get(...arguments);
    return typeof value == 'function' ? value.bind(target) : value;
  }
});
```

```
proxy.set('test', 1);
alert(proxy.get('test')); // 1 (works!)
```

الآن تعمل وفق المطلوب، لأن اعتراض `get` يربط خاصيات الدوال، مثل `map.set`، بالكائن الهدف `map` نفسه. بخلاف المثال السابق، فإن قيمة `this` داخل `proxy.set (...)` لن تكون `proxy`، وإنما `map` الأصلية. لذا عند التطبيق الداخلي لـ `set` سيحاول الوصول إلى الفتحة الداخلية هكذا `this.[[MapData]]`، ولحسن الحظ سينجح.

المصفوفة العادية Array ليس لها فتحات داخلية

استثناء ملحوظ: لا تستخدم المصفوفات المضمنة `Array` الفتحات الداخلية. هذا لأسباب تاريخية، إذ إنها ظهرت منذ وقت طويل. لذلك لا توجد مشكلة عند تغليف المصفوفة إلى باستخدام الوسيط.

ب. الخاصيات الخاصة

يحدث شيء مشابه للأمر مع خاصيات الصنف الخاصة.

فمثلًا، يمكن للدالّة `getName()` الوصول إلى الخاصية الخاصة `#name` بدون استخدام الوسيط، ولكن بعد تغليفنا للكائن باستخدام الوسيط ستتوقف إمكانية وصول الدالّة السابقة للخاصّية الخاصّة:

```
class User {
  #name = "Guest";

  getName() {
    return this.#name;
  }
}

let user = new User();

user = new Proxy(user, {});

alert(user.getName()); // Error
```

وذلك بسبب أن التنفيذ الفعلي للخاصّيات الخاصّة يكون باستخدام الفتحات داخلية. ولا تستخدم لغة جافاسكربت الدوال `[[Set]]/[[Get]]` للوصول إليها.

في استدعاء `getName()`، تكون قيمة `this` هي كائن `user` المغلّف بالوسيط، ولا يحتـوي -هـذا الكـائن- على فتحة داخلية مع هذه الخاصّيات الخاصّة. وللمرة الثانية، يكون ربط الدالّة بالكائن من سيحل الأمر:

```
class User {
  #name = "Guest";

  getName() {
    return this.#name;
  }
}

let user = new User();

user = new Proxy(user, {
  get(target, prop, receiver) {
    let value = Reflect.get(...arguments);
    return typeof value == 'function' ? value.bind(target) : value;
  }
});

alert(user.getName()); // Guest
```

ومع ذلك، فإن لهذا الحل بعض العيوب، كمـا وضـحنا سـابقًا: سـيعرض هـذا الحـل الكـائن الأصـلي لبعض الدوال، مما سيُسمح بتمريره لدوالٍ أكثر وبذلك كسر الدوال الأخرى المتعلقة بالوسيط.

ج. Proxy! = target

إن كلًا من الوسيط والكائن الأصلي مختلفان. وهذا أمر طـبيعي، أليس كـذلك؟ لـذلك إذا اسـتخدمنا الكـائن الأصلي كخاصية في المجموعة `Set`، ثم غلفناه بالوسيط، فعندئذ لن نتمكن من العثور على الوسيط:

```
let allUsers = new Set();

class User {
  constructor(name) {
    this.name = name;
    allUsers.add(this);
  }
}
```

```
let user = new User("Ahmad");

alert(allUsers.has(user)); // true

user = new Proxy(user, {});
// لاحظ
alert(allUsers.has(user)); // false
```

كما رأينا، بعد التغليف باستخدام الوسيط لن نتمكن من العثور على كائن `user` في المجموعة `allUsers`، لأن الوسيط هو كائن مختلف عن الكائن الأصلي.

> لا يمكن للوسطاء proxies اعتراض اختبار المساواة الصارم ===. يمكنها اعتراض العديد من العمليات الأخرى، مثل: new (مع build) وin (مع has) وdelete (مع deleteProperty) وما إلى ذلك.
> ولكن لا توجد طريقة لاعتراض اختبار المساواة الصارم للكائنات. الكائن يساوي نفسه تمامًا ولا يساوي أيّ كائنٍ آخر، لذا فإن جميع العمليات والأصناف المضمّنة في اللغة التي توازن بين الكائنات من أجل المساواة ستميز الفرق بين الكائن الأصلي والوسيط. ولا يوجد من يحل محله لإصلاح الأمر.

14.1.10 الوسيط القابل للتعطيل

وسيط revocable هو وسيط يمكن تعطيله.

لنفترض أن لدينا موردًا ونود منع الوصول إليه في لحظةٍ ما. أحد الأشياء التي يمكننا فعلها هو تغليف هذا المورد بالوسيط القابل للتعطيل، بدون أي اعتراضات. بهذه الحالة سيُعيد الوسيط توجيه العمليات للكائن، ويمكننا تعطيله بأي لحظة نريدها.

وصياغته:

```
let {proxy, revoke} = Proxy.revocable(target, handler)
```

الاستدعاء من خلال `proxy` سيُعيد الكائن والاستدعاء من خلال `revoke` سيعطل إمكانية الوصول إليه.

إليك المثال لتوضيح الأمر:

```
let object = {
  data: "Valuable data"
};

let {proxy, revoke} = Proxy.revocable(object, {});
```

```
// مرر الوسيط لمكان آخر بدل الكائن
alert(proxy.data); // Valuable data

// لاحقا نستدعي التابع
revoke();

// لن يعمل الوسيط الآن (لأنه معطّل)
alert(proxy.data); // Error
```

يؤدي استدعاء `revoke()` لإزالة جميع المراجع الداخلية للكائن الهدف من الوسيط، وبذلك لم يعد متصل بأي شيء بعد الآن. كما يمكننا بعد ذلك كنس المخلفات من الذاكرة بإزالة الكائن الهدف.

يمكننا أيضًا تخزين `revoke` في `WeakMap`، حتى نتمكن من العثور عليه بسهولة من خلال كائن الوسيط:

```
let revokes = new WeakMap();

let object = {
  data: "Valuable data"
};

let {proxy, revoke} = Proxy.revocable(object, {});

revokes.set(proxy, revoke);

// ..لاحقًا في الشيفرة..
revoke = revokes.get(proxy);
revoke();

alert(proxy.data); // Error (revoked)
```

تفيدنا هذه الطريقة بأنه ليس علينا بعد الآن حمل `revoke`. وإنما يمكننا الحصول عليها من `map` من خلال الوسيط `proxy` عند الحاجة.

نستخدم `WeakMap` بدلاً من `Map` هنا لأنه لن يمنع كنس المخلّفات في الذاكرة. إذا أصبح كائن الوسيط "غير قابل للوصول" (مثلًا، في حال لم يعد هناك متغيّر يشير إليه بعد الآن)، فإن `WeakMap` يسمح بمسحه من الذاكرة مع `revoke` خاصته والّتي لن نحتاج إليها بعد الآن.

14.1.11 المصادر

- المواصفات القياسية للوسيط: Proxy.
- توثيق الوسيط الرسمي من مركز مطوري موزيلا MDN.

14.1.12 الخلاصة

الوسيط `Proxy` عبارة عن غلاف حول كائن، يُعيد توجيه العمليات عليه إلى الكائن، ويحبس بعضها بشكل اختياري. يمكنه تغليف أي نوع من الكائنات، بما في ذلك الأصناف والدوال.

صياغته:

```
let proxy = new Proxy(target, {
  /* traps */
});
```

... ثم يجب علينا استخدام "الوسيط" في كل مكان بدلاً من كائن "الهدف". لا يمتلك الوكيل خاصيات أو توابع. يعترض عملية ما إذا زُودَ بالاعتراض المناسب، وإلا سيعيد توجيهها إلى كائن الهدف `target`.

يمكننا اعتراض:

- قراءة (`get`) وكتابة (`set`) وحذف (`deleteProperty`) خاصية (حتى الخاصية غير موجودة).
- استدعاء دالّة ما (الاعتراض `apply`).
- المعامل `new` (الاعتراض `construct`).
- العديد من العمليات الأخرى (القائمة الكاملة في بداية الفصل وفي التوثيق الرسمي).

مما سيسمح لنا بإنشاء خاصيات ودوالّ "افتراضية"، وتطبيق قيم افتراضية، وكائنات المراقبة، وزخرفة الدوالّ، وأكثر من ذلك بكثير.

يمكننا أيضًا تغليف كائن ما عدة مرات في وسطاء مختلفة، وزخرفته بمختلف أنواع الوظائف. صُممّت الواجهة البرمجية للمنعكس لتكمل عمل الوسيط. بالنسبة لأي اعتراض `Proxy`، هناك استدعاء للمنعكس `Reflect` مقابل له بنفس الوسطاء. يجب علينا استخدامها لإعادة توجيه الاستدعاءات إلى الكائنات المستهدفة. لدى الوسيط بعض القيود:

- تحتوي الكائنات المضمّنة في اللغة على "فتحات داخلية"، ولا يمكن الوصول إلى تلك الأشياء بالوسيط. راجع الفقرة المخصصة لها أعلاه.

- وينطبق الشيء نفسه على خاصيات الصنف الخاصة، إذ تنفيذها داخليًا باستخدام الفتحات. لذا يجب أن تحتوي استدعاءات دوالّ الوسيط على الكائن المستهدف بدل `this` للوصول إليها.
- لا يمكن اعتراض اختبارات المساواة الصارمة للكائن `===`.
- الأداء: تعتمد المقاييس على المحرك، ولكن عمومًا إن الوصول إلى الخاصية باستخدام وكيل بسيط سيستغرق وقتًا أطول بعض الشيء. عمليًا يهتم بها البعض لعدم حدوث اختناق في الأداء "عنق الزجاجة".

14.1.13 تمارين

ا. خطأ في قراءة الخاصيات غير موجودة في الكائن الأصلي

عادةً ، تؤدي محاولة قراءة خاصية غير موجودة إلى إعادة النتيجة `undefined`.

أنشئ وسيطًا يعيد خطأ عند محاولة قراءة خاصية غير موجودة في الكائن الأصلي بدلًا من ذلك. يمكن أن يساعد ذلك في الكشف عن الأخطاء البرمجية مبكرًا.

اكتب دالّة `wrap(target)` والّتي تأخذ كائنًا `target` وتعيد وسيطًا والّذي سيُضيف خصائص وظيفية أخرى. هكذا يجب أن تعمل:

```
let user = {
  name: "Ahmad"
};

function wrap(target) {
  return new Proxy(target, {
      /* your code */
  });
}
user = wrap(user);3
alert(user.name); // Ahmad
alert(user.age); // ReferenceError: Property doesn't exist "age"
```

الحل:

```
let user = {
  name: "Ahmad"
};
```

```
function wrap(target) {
  return new Proxy(target, {
    get(target, prop, receiver) {
      if (prop in target) {
        return Reflect.get(target, prop, receiver);
      } else {
        throw new ReferenceError(`Property doesn't exist: "${prop}"`)
      }
    }
  });
}

user = wrap(user);

alert(user.name); // Ahmad
alert(user.age); // ReferenceError: Property doesn't exist "age"
```

ب. الوصول إلى الدليل [-1] في فهرس المصفوفة

يمكننا الوصول إلى عناصر المصفوفة باستخدام الفهارس السلبية في بعض لغات البرمجة، محسوبةً بـذلك من نهاية المصفوفة. هكذا:

```
let array = [1, 2, 3];

array[-1]; // 3, آخر عنصر في المصفوفة
array[-2]; // 2, خطوة للوراء من نهاية المصفوفة
array[-3]; // 1, خطوتين للوراء من نهاية المصفوفة
```

بتعبيرٍ آخر ، فإن `array[-N]` هي نفس `array[array.length - N]`.

أنشئ وسيطًا لتنفيذ هذا السلوك.

هكذا يجب أن تعمل:

```
let array = [1, 2, 3];

array = new Proxy(array, {
  /* your code */
```

```
});

alert( array[-1] ); // 3
alert( array[-2] ); // 2

// بقية الخصائص الوظيفية الأخرى يجب أن تبقى كما هي
```

الحل:

```
let array = [1, 2, 3];

array = new Proxy(array, {
  get(target, prop, receiver) {
    if (prop < 0) {
      // حتى وإن وصلنا للمصفوفة هكذا arr[1]
      // إن المتغيّر prop عبارة عن سلسلة نصية لذا نحتاج لتحويله إلى رقم
      prop = +prop + target.length;
    }
    return Reflect.get(target, prop, receiver);
  }
});


alert(array[-1]); // 3
alert(array[-2]); // 2
```

ج. المراقب

أنشئ تابع `makeObservable(target)` الّذي تجعل الكائن قابلاً للمراقبة 'من خلال إعادة وسيط.

إليك كيفية العمل:

```
function makeObservable(target) {
  /* your code */
}

let user = {};
user = makeObservable(user);
```

```
user.observe((key, value) => {
  alert(`SET ${key}=${value}`);
});

user.name = "Ahmad"; // alerts: SET name=Ahmad
```

وبعبارة أخرى، فإن الكائن الّذي سيعاد من خلال `makeObservable` يشبه تمامًا الكائن الأصلي، ولكنه يحتوي أيضًا على الطريقة `observe(handler)` الّتي ستضبط تابع المُعالج ليستدعى عند أي تغيير في الخاصية. عندما تتغير خاصية ما، يستدعى `handler(key, value)` مع اسم وقيمة الخاصية.

> في هذا التمرين، يرجى الاهتمام بضبط الخاصيات فقط. يمكن تنفيذ عمليات أخرى بطريقة مماثلة.

الحل:

يتكون الحل من جزئين:

1. عندما يستدعى `.observe(handler)` نحتاج إلى حفظ المعالج في مكان ما، حتى نتمكن من الاتصال به لاحقًا. يمكننا تخزين المعالجات في الكائن مباشرة، باستخدام الرمز الخاص بنا كمفتاح خاصية.
2. سنحتاج لوسيط مع الاعتراض `set` لاستدعاء المعالجات عند حدوث أي تغيير.

```
let handlers = Symbol('handlers');

function makeObservable(target) {
  // 1. هيئ مخزّن المعالجات
  target[handlers] = [];

  // احتفظ بتوابع المعالج في مصفوفة للاستدعاءات اللاحقة
  target.observe = function(handler) {
    this[handlers].push(handler);
  };

  // 2. أنشئ وسيط ليعالج التغييرات
  return new Proxy(target, {
    set(target, property, value, receiver) {
      let success = Reflect.set(...arguments); // وجه العملية إلى الكائن
```

```
      if (success) { // إن حدث خطأ ما في ضبط الخاصية
        // استدعي جميع المعالجات
        target[handlers].forEach(handler => handler(property, value));
      }
      return success;
    }
  });
}

let user = {};

user = makeObservable(user);

user.observe((key, value) => {
  alert(`SET ${key}=${value}`);
});

user.name = "Ahmad";
```

14.2 الدالة "Eval" لتنفيذ الشيفرة البرمجية

تنفذ الدالّة Eval المضمّنة في اللغة الشيفرات البرمجية المُمرّرة لها كسلسلة نصية string.

وصياغتها هكذا:

```
let result = eval(code);
```

فمثلًا:

```
let code = 'alert("Hello")';
eval(code); // Hello
```

يمكن أن تكون الشيفرة المُمررة للدالّة كبيرة وتحتوي على فواصل أسطر وتعريف دوال ومتغيّرات، وما إلى ذلك. ولكن نتيجة الدالّة Eval هي نتيجة أخر عبارة منفذة في الشيفرة.

وإليك المثال التالي:

```
let value = eval('1+1');
alert(value); // 2
let value = eval('let i = 0; ++i');
alert(value); // 1
```

تُنفذّ الشيفرة في البيئة الحالية للدالّة، ولذا فيمكنها رؤية المتغيرات الخارجية:

```
let a = 1;

function f() {
  let a = 2;
  eval('alert(a)'); // 2
}
f();
```

كما يمكنها تعديل المتغيّرات الخارجية أيضًا:

```
let x = 5;
eval("x = 10");
alert(x); // النتيجة: 10، تعدلت القيمة بنجاح
```

في الوضع الصارم، تملك الدالّة Eval بيئة متغيّرات خاصة بها. لذا فلن تظهر الدوال والمتغيرات، المعرفة -داخل الدالة- للخارج وإنما ستبقى بداخلها:

```
// تذكر أن في الوضع الصارم يُشغَّل تلقائيًا في الأمثلة الحيّة
eval("let x = 5; function f() {}");

alert(typeof x); // undefined (المتحول غير مرئي هنا)
// الدالّة f غير مرئية هنا أيضًا
```

بدون تفعيل "الوضع صارم"، لن يكون للدالّة Eval بيئة متغيرات خاصة بها، ولذلك سنرى المتغيّر x والدالّة f من خارج الدالّة.

14.2.1 استخدامات الدالة Eval

في طرق البرمجة الحديثة، نادرًا ما تستخدم الدالّة Eval. وغالبًا ما يقال عنها أنها أصل الشرور.

والسبب بسيط: إذ كانت لغة جافاسكربت منذ زمن بعيد أضعف بكثير من الآن، ولم يكُ بالإمكان فعل إيّ شيء إلا باستخدام الدالّة Eval. ولكن ذلك الوقت مضى عليه عقد من الزمن.

حاليًا، لا يوجد سبب وجيه لاستخدامها. ولو أن شخصًا يستخدمها الآن فلديه الإمكانية لاستبدالها بالبنية الحديثة للغة أو بالوحدات. لاحظ أن إمكانية وصول الدالة eval للمتغيرات الخارجية لها عواقب سيئة.

إن عملية تصغير الشيفرة (هي الأدوات تستخدم لتصغير شيفرة الجافاسكربت قبل نشرها وذلك لتصغير حجمها أكثر من ذي قبل) تعيد تسمية المتغيّرات المحلية لأسماء أقصر (مثل a وb وما إلى ذلك) لتصغير الشيفرة. وعادةً ما تكون هذه العلمية آمنة، ولكن ليس في حال استخدام الدالّة Eval، إذ يمكننا الوصول للمتغيّرات المحلية من الشيفرة المُمررة للدالّة. لذا، لن تصغّر المتغيرات التي يحتمل أن تكون مرئية من الدالة Eval. مما سيُؤثر سلبًا على نسبة ضغط الشيفرة.

يُعدّ استخدام المتغيّرات المحلية في الشيفرة بداخل الدالّة Eval من الممارسات البرمجية السيئة، لأنه يزيد صعوبة صيانة الشيفرة.

هناك طريقتان لضمان الأمان الكامل عند مصادفتك مثل هذه المشاكل. إذا لم تستخدم الشيفرة الممررة للدالّة المتغيرات الخارجية، فمن الأفضل استدعاء الدالّة هكذا: `window.eval(...)`

بهذه الطريقة ستُنفذّ الشيفرة في النطاق العام:

```
let x = 1;
{
  let x = 5;
  window.eval('alert(x)'); // 1 (global variable)
}
```

إن احتاجت الشيفرة الممررة للدالة `Eval` لمتغيّرات خارجية، فغيّر `Eval` لتصبح `new Function` ومرّر المتغير كوسيط. هكذا:

```
let f = new Function('a', 'alert(a)');
f(5); // 5
```

شرحنا في فصلٍ سابق تعلمنا كيفية استخدام صياغة "الدالة الجديدة" new Function. إذ باستخدام هذه الصياغة ستُنشأ دالة جديدة من السلسلة (String)، في النطاق العام. لذا لن تتمكن من رؤية المتغيرات المحلية. ولكن من الواضح أن تمريرها المتغيرات صراحة كوسطاء سيحلّ المشكلة، كما رأينا في المثال أعلاه.

14.2.2 الخلاصة

سيُشغّل استدعاء الدالّة `eval(code)` الشيفرة البرمجية المُمرّرة ويعيد نتيجة العبارة الأخيرة.

- نادرًا ما تستخدم هذه الدالّة في الإصدارات الحديثة للغة، إذ لا توجد حاجة ماسّة لها.
- يمكننا الوصول دائمًا للمتغيّرات الخارجية في الدالّة `eval`. ولكن يعدّ ذلك من الممارسات السيئة.
- بدلًا من ذلك يمكننا استخدام الدالة `eval` في النطاق العام، هكذا `window.eval(code)`.
- أو، إذا كانت الشيفرة الخاصة بك تحتاج لبعض البيانات من النطاق الخارجي، فاستخدم صياغة الدالّة الجديدة ومرّر لها المتغيرات كوسطاء.

14.2.3 التمارين

ا. آلة حاسبة باستخدام الدالة Eval

الأهمية: ★★★★☆

أنشئ آلة حاسبة تطالب بتعبير رياضي وتُعيد نتيجته. لا داعي للتحقق من صحة التعبير في هذا التمرين. فقط قيّم التعبير وأعد نتيجته.

لرؤية المثال الحي

الحل:

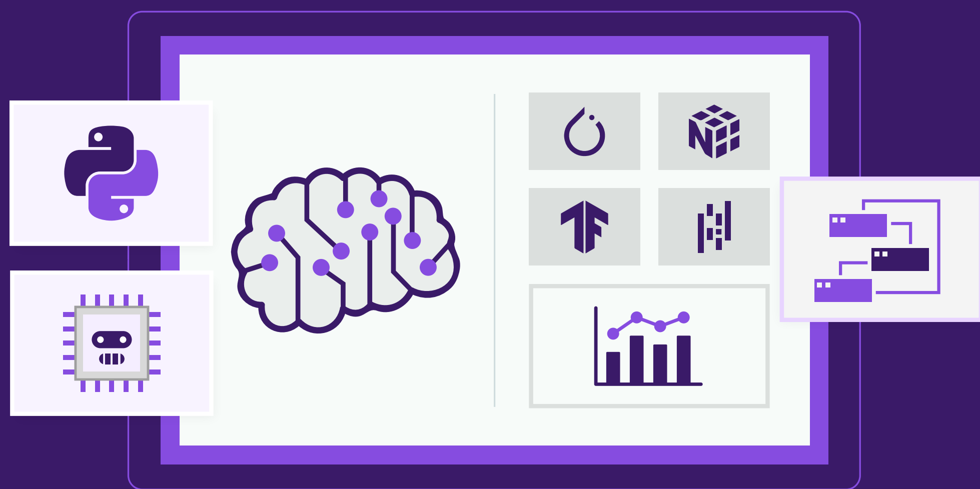
لنستخدم الدالة `eval` لحساب التعبير الرياضي:

```
let expr = prompt("Type an arithmetic expression?", '2*3+2');

alert( eval(expr) );
```

يستطيع المستخدم أيضًا إدخال أي نص أو شيفرة.

لجعل الشيفرة آمنة، وحصرها للعمليات الرياضية فحسب، سنتحقق من `expr` باستخدام التعـابير النمطية، لكي لا تحتوي إلا على الأرقام والمعاملات رياضية.

دورة الذكاء الاصطناعي



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

14.3 تقنية Currying في جافاسكربت

مفهوم Currying هو تقنية متقدمة للعمل مع الدوالّ. يستخدم في العديد من اللغات البرمجية الأخرى من بينهم جافاسكربت.

Currying عبارة عن طريقة لتحويل الدوالّ التي تقيم الدالّة ذات الاستدعاء-أكثر من وسيط- `f (a، b، c)` لتصبح قابلة للاستدعاء -بوسيط واحد- هكذا `f(a)(b)(c)`. تحول تقنية Currying الدالّة فقط ولا تستدعها.

لنرى في البداية مثالًا، لفهم ما نتحدث عنه فهمًا أفضل، وبعدها ننتقل للتطبيقات العملية. سننشئ دالة مساعدة باسم `curry (f)` والتي ستُنفذّ تقنية Currying على الدالّة `f` التي تقبل وسيطين. بتعبير آخر، تحول الدالة `curry(f)` الدالّة `f(a, b)` ذات الوسيطين إلى دالة تعمل كوسيط واحد`f(a)(b)`:

```
function curry(f) { // curry(f) الدالة هي من ستنفذّ تحويل currying
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}

// طريقة الاستخدام
function sum(a, b) {
  return a + b;
}

let curriedSum = curry(sum);
alert( curriedSum(1)(2) ); // 3
```

كما نرى، فإن التنفيذ بسيط: إنه مجرد مغلفين للوسطاء.

- نتيجة `curry(func)` هي دالة مغلّفة `function(a)`.
- عندما تسمى هكذا `curriedSum(1)`، تُحفظ الوسطاء في البيئة اللغوية للجافاسكربت (وهي نوع مواصفات اللغة تستخدم لتعريف ارتباط المعرّفات بالمتغيّرات والدوالّ المحددة وذلك بناءً على بنية الترابط اللغوية في شيفرة ECMAScript)، وتعيد غلاف جديد `function(b)`.
- ثمّ يُسمى هذا المغلّف باسم 2 نسبةً لوسطائه، ويُمرّر الاستدعاء إلى الدالّة الأصليّة `sum(a,b)`.

من الأمثلة المتقدمة باستخدام تقنية currying هو _curry من مكتبة Lodash، والتي تُعيد غِلافًا الّذي يسمح باستدعاء الدالّة طبيعيًا وجزئيًا:

```
function sum(a, b) {
  return a + b;
}

let curriedSum = _.curry(sum); // lodash استخدام _.curry من مكتبة

alert( curriedSum(1, 2) ); // النتيجة: 3، لايزال بإمكاننا استدعاؤه طبيعيًا
alert( curriedSum(1)(2) ); // النتيجة: 3، الاستدعاء الجزئي
```

14.3.1 لماذا نحتاج لتقنية currying؟

لابد لنا من مثالٍ واقعي لفهم فوائد هذه التقنية. مثلًا، ليكن لدينا دالّة التسجيل log(date, importance, message) والّتي ستُنسّق المعلومات وتعرضها. مثل هذه الدوال مفيدة جدًا في المشاريع الحقيقة مثل: إرسال السجلات عبر الشبكة، في مثالنا سنستخدم فقط alert:

```
function log(date, importance, message) {
  alert(`[${date.getHours()}:${date.getMinutes()}] [${importance}] ${message}`);
}
```

لنُنفذ تقنية currying عليها!

```
log = _.curry(log);
```

بعد ذلك ستعمل دالّة log وفق المطلوب:

```
log(new Date(), "DEBUG", "some debug"); // log(a, b, c)
```

ولكنها تعمل أيضًا بعد تحويلها بتقنية currying:

```
log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)
```

الآن يمكننا بسهولة إنشاء دالّة مناسبة للسجلات الحالية:

```
// logNow سيكون دالة جزئية من log مع وسيط أول ثابت
let logNow = log(new Date());
```

```
// استخدامه
logNow("INFO", "message"); // [HH:mm] INFO message
```

الآن `logNow` هو نفس الدالة `log` بوسيط أول ثابت، بمعنى آخر "دالّة مطبقة جزئيًا" أو "جزئية" للاختصار.

يمكننا المضي قدمًا وإنشاء دالّة مناسبة لسجلات تصحيح الأخطاء الحالية:

```
let debugNow = logNow("DEBUG");

debugNow("message"); // [HH:mm] DEBUG message
```

إذا:

1. لم نفقد أي شيء بعد التحويل بتقنية currying: ولا يزال يمكننا أيضًا استدعاء الدالّة `log` طبيعيًا.
2. يمكننا بسهولة إنشاء دوال جزئية مثل: سجلات اليوم.

14.3.2 الاستخدام المتقدم لتقنية currying

في حالة رغبتك في الدخول في التفاصيل، إليك طريقة الاستخدام المتقدمة لتقنية currying للـدوال ذات الوسطاء المتعددة والّتي يمكننا استخدامها أعلاه.

وهي مختصرة جدًا:

```
function curry(func) {

  return function curried(...args) {
    if (args.length >= func.length) {
      return func.apply(this, args);
    } else {
      return function(...args2) {
        return curried.apply(this, args.concat(args2));
      }
    }
  };

}
```

إليك مثالًا لطريقة استخدامه:

```
function sum(a, b, c) {
  return a + b + c;
}

let curriedSum = curry(sum);

alert( curriedSum(1, 2, 3) ); // 6, still callable normally
alert( curriedSum(1)(2,3) ); // 6, currying of 1st arg
alert( curriedSum(1)(2)(3) ); // 6, full currying
```

تبدو تقنية currying للوهلة الأولى معقدةً، ولكنها في الحقيقة سهلة الفهم جدًا. نتيجة استدعاء `curry(func)` هي دالّة مُغلّفة `curried` والّتي تبدو هكذا:

```
// func is the function to transform
function curried(...args) {
  if (args.length >= func.length) { // (1)
    return func.apply(this, args);
  } else {
    return function pass(...args2) { // (2)
      return curried.apply(this, args.concat(args2));
    }
  }
};
```

عند تشغيله، هناك فرعين للتنفيذ من الجملة الشرطية `if`:

1. سيكون الاستدعاء الآن هكذا: إن كان عدد الوسطاء `args` المُمرّرة هو نفس العدد الدالة الأصليّة المعرّفة لدينا (`func.length`) أو أكثر، عندها نمرّر الاستدعاء له فقط.
2. وإلا سيكون الاستدعاء جزئيًا: لم تُستدعى الدالّة `func` بعد. وإنما أعيد بدلًا منها دالّة المغلِّفة أخرى `pass`، والتي ستُعيد تطبيق الدالة `curried` مع تقديم الوسطاء السابقين مع الوسطاء الجدد. وثمّ في استدعاء الجديد سنحصل إما على دالة جزئية جديدة (إن لم يكُ عدد الوسطاء كافي) أو النتيجة النهائية.

لنرى مثلًا ما يحدث في حال الاستدعاء الدالة هكذا `sum(a, b, c)`. أي بثلاث وسطاء، وبذلك يكون `sum.length = 3`.

عند استدعاء `curried(1)(2)(3)`:

1. الاستدعاء الأول `curried (1)` تحفظ `1` في بيئته اللغوية، ويُعيد دالّة المغلف `pass`.

2. يُستدعى المغلّف `pass` مع الوسيط المُمرّر `(2)`: إذ يأخذ الوسطاء السابقين (`1`)، ويدمجهم مع الوسيط الذي حصل عليه وهو `(2)` ويستدعي الدالّة `curried(1, 2)` مع استخدام جميع ما حصل عليه من وسطاء. وبما أن عدد الوسطاء لا يزال أقل من 3 ، فإن الدالّة `curry` ستُعيد الدالّة `pass`.
3. يُستدعى المغلّف `pass` مرة أخرى مع الوسيط المُمرّر `(3)`: ومن أجل الاستدعاء التالي `pass (3)` سيأخذ الوسطاء السابقين (`1`, `2`) ويضيف لهم الوسيط `3`، ليكون الاستدعاء `curried(1, 2, 3)`-

أخيرًا لدينا ثلاث وسطاء، والّذين سيمرّروا للدالّة الأصلية. إذا لم تتوضح الفكرة حتى الآن ، فما عليك إلا تتبع تسلسل الاستدعاءات في عقلك أو على الورقة وستتوضح الأمور أكثر.

تعمل مع الدوال ثابتة الطول فقط

يجب أن يكون للدالّة عدد ثابت من الوسطاء لتطبيق تقنية currying.

إن استخدمت دالّة ما معاملات البقية، مثل: `f(...args)`، فلا يمكن معالجتها بهذه التقنية.

أكثر بقليل من مجرد تقنية تحويل

انطلاقًا من التعريف، يجب على تقنية currying تحويل الدالّة `sum(a, b, c)` إلى `sum(a)(b)(c)`.

لكن غالبية تطبيقات هذه التقنية في جافاسكربت متقدمة، وكما وضحنا سابقًا: فهي تحافظ على الدالّة قابلة للاستدعاء بعدة تنويعات للوسطاء المُمرّرة.

14.3.3 الخلاصة

تقنية Currying هو عملية تحويل تجعل `f(a,b,c)` قابلة للاستدعاء كـ `f(a)(b)(c)`. عادةً ما تحافظ تطبيقات الجافاسكربت على الدوال بحيث تكون قابلة للاستدعاء بالشكل الطبيعي أو الجزئي إن كان عدد الوسطاء غير كافٍ.

كما تسمح لنا هذه التقنية أيضًا بالحصول على دوالّ جزئية بسهولة. كما رأينا في مثال التسجيل، بعد تنفيذ هذه التقنية على الدالّة العالمية ذات الثلاث وسطاء `log(date, importance, message)` فإن ذلك سيمنحنا دوالّ جزئية عند استدعاؤها باستخدام وسيط واحد هكذا `log(date)` أو وسيطين هكذا `log(date, importance)`.

14.4 النوع المرجعي eference

> هذه الشرح لميزات اللغة المتقدمة، إذ يتناول هذا الفصل موضوعًا متقدمًا، لفهم بعض الحالات الهامشية بطريقة أفضل.
> مع العلم أنها ليست مهمة. والعديد من المطورين ذوي الخبرة الجيدة ينفذون مشاريعهم بدون معرفتها. ولكن إذا أردت معرفة كيفية عمل الأشياء تحت الطاولة فتابع القراءة.

إن استدعاء التابع المقيم ديناميكيًا يمكن أن يفقد قيمة `this`. فمثلًا:

```
let user = {
  name: "Ahmad",
  hi() { alert(this.name); },
  bye() { alert("Bye"); }
};

user.hi(); // works

// لنستدعي الآن user.hi أو user.bye بحسب الاسم
_(user.name == "Ahmad" ? user.hi : user.bye)(); // خطأ
```

يوجد في السطر الأخير معامل شرطي يختار إما `user.hi` أو `user.bye`. في هذه الحالة تكون النتيجة `user.hi`، ثم تستدعى الدالّة ذات الأقواس `()` على الفور لكنها لن تعمل بطريقة صحيحة!

كما تلاحظ، نتج عن الاستدعاء بالطريقة السابقة خطأ، وذلك لأن قيمة `this` داخل الاستدعاء تصبح غير معرفة `undefined`.

نلاحظ هنا أن هذه الطريقة تعمل بصورة صحيحة (اسم الكائن ثم نقطة ثم اسم الدالّة):

```
user.hi();
```

أما هذه الطريقة فلن تعلم (طريقة تقييم الدالّة):

```
(user.name == "Ahmad" ? user.hi : user.bye)(); // Error!
```

لماذا حدث ذلك؟ إذا أردنا أن نفهم سبب حدوث ذلك، فلنتعرف أولًا على كيفية عمل الاستدعاء `obj.method()`.

14.4.1 شرح النوع المرجعي

بالنظر عن كثب، قد نلاحظ عمليتين في العبارة `obj.method()`:

1. أولًا، تسترجع النقطة `'.'` الخاصية `obj.method`.
2. ثم تأتي الأقواس `()` لتنفّذها.

إذًا كيف تنتقل المعلومات الخاصة بقيمة `this` من الجزء الأول إلى الجزء الثاني؟ إذا وضعنا هذه العمليات في سطور منفصلة، فسنفقدُ قيمة `this` بكل تأكيد:

```
let user = {
  name: "Ahmad",
  hi() { alert(this.name); }
}
// جزّء استدعاء التوابع على سطرين
let hi = user.hi;
hi(); // خطأ لأنها غير معرّفة
```

إن التعليمة `hi = user.hi` تضع الدالّة في المتغير، ثم الاستدعاء في السطر الأخير يكون مستقلًا تمامًا، وبالتالي لا يوجد قيمة لـ `this`.

لجعل استدعاءات `user.hi()` تعمل بصورة صحيحة، تستخدم لغة جافاسكربت خدعة - بأن لا تُرجع النقطة `'.'` دالة، وإنما قيمة من نوع مرجعي.

إن القيمة من نوع مرجعي هو "نوع من المواصفات". لا يمكننا استخدامها بطريقة مباشرة، ولكن تستخدم داخليًا بواسطة اللغة.

إن القيمة من نوع مرجعي هي مجموعة من ثلاث قيم `(base, name, strict)`، ويشير كلٌّ منها إلى:

- `base`: وهو الكائن.
- `name` وهو اسم الخاصية.
- `strict` تكون قيمتها `true` إذا كان الوضع الصارم مفعلًا.

إن نتيجة وصول الخاصية `user.hi` ليست دالة، ولكنها قيمة من نوع مرجعي. بالنسبة إلى `user.hi` في الوضع الصارم، تكون هكذا:

```
// قيمة من نوع مرجعي
(user, "hi", true)
```

عندما تستدعى الأقـواس `()` في النـوع المـرجعي، فإنهـا تتلقى المعلومـات الكاملـة حـول الكـائن ودوالـه، ويمكنهم تعيين قيمة `this` (ستكون user في حالتنا).

النـوع المـرجعي هـو نـوع داخلي خـاص "وسـيط"، بهـدف تمريـر المعلومـات من النقطـة `.` إلى أقـواس الاستدعاء `()`. أي عملية أخرى مثل الإسناد `hi = user.hi` تتجاهل نوع المرجع ككلّ، وتأخذ قيمة `user.hi` (كدالّة) وتمررها. لذا فإن أي عملية أخرى تفقد قيمة `this`.

لذلك وكنتيجة لما سبق، تُمرر قيمة `this` بالطريقة الصحيحة فقط إذا استدعيت الدالّـة مباشـرة باسـتخدام نقطة `obj.method()` أو الأقواس المربعة `obj['method']()` (الصياغتين تؤديـان الشـيء نفسـه). لاحقًـا في هذا الكتاب، سنتعلم طرقًا مختلفة لحل هذه المشكلة مثل الدالّة func.bind().

14.4.2 الخلاصة

النوع المرجعي هو نوع داخلي في لغة جافاسكربت.

قراءة خاصية ما، مثل النقطة `.` في`obj.method()` لا تُرجع قيمة الخاصية فحسـب، بـل تُرجـع قيمـة من "نوع مرجعي" خاص والّتي تخزن فيها قيمة الخاصية والكائن المأخوذة منه.

هذا الطريقة لاستدعاء الدالة اللاحقة `()` للحصول على الكائن وتعيين قيمة `this` إليه.

بالنسبة لجميع العمليات الأخرى، يصبح قيمة النوع المرجعي تلقائيًا قيمة الخاصية (الدالّة في حالتنا).

جميع هذه الآليات مخفية عن أعيننا. ولا تهمنا إلا في الحالات الدقيقة، مثل عندما نريـد الحصـول على تـابع ديناميكيًا من الكائن ما، باستخدام تعبير معيَّن.

14.4.3 المهام

ا. التحقق من الصياغة

الأهمية: ★★☆☆☆

ما هي نتيجة هذه الشيفرة البرمجية؟

```
let user = {
  name: "Ahmad",
  go: function() { alert(this.name) }
}

(user.go)()
```

انتبه. هنالك فخ :)

الحل:

سينتج عن تنفيذ الشيفرة السابقة خطأ!

جربها بنفسك:

```
let user = {
  name: "Ahmad",
  go: function() { alert(this.name) }
}

(user.go)() // error!
```

لا تعطينا رسالة الخطأ في معظم المتصفحات فكرة عن الخطأ الّذي حدث.

يظهر الخطأ بسبب فقدان فاصلة منقوطة بعد `user = {...}` إذ لا تُدرج لغة جافاسكربت تلقائيًا فاصلة منقوطة قبل قوس `(user.go)()`، لذلك تُقرأ الشيفرة البرمجية هكذا:

```
let user = { go:... }(user.go)()
```

ثم يمكننا أن نرى أيضًا أن مثل هذا التعبير المشترك هو من الناحية التركيبية استدعاء للكائن `{ go: ... }` كدالة مع الوسيط `(user.go)`. ويحدث ذلك أيضًا على نفس السطر مع `let user`، لذلك إن الكائن `user` لم يُعرّف حتى الآن، ومن هنا ظهر الخطأ.

إذا أدخلنا الفاصلة المنقوطة، فسيكون كل شيء على ما يرام:

```
let user = {
  name: "Ahmad",
  go: function() { alert(this.name) }
};

(user.go)() // Ahmad
```

يرجى ملاحظة أن الأقواس حول `(user.go)` لا تفعل شيئًا مميزًا هنا. وهي عادةً تعيد ترتيب العمليات، ولكن هنا تعمل النقطة `.` أولًا على أي حال، لذلك ليس هناك أي تأثير لها. الشيء الوحيد المهم هو الفاصلة المنقوطة.

ب. اشرح قيمة this

الأهمية: ★★★☆☆

في الشيفرة أدناه، نعتزم استدعاء التابع `obj.go()` لأربع مرات على متتالية، ولكن لماذا يعمل النداءان `(1)` و`(2)` يعملان بطريقة مختلفة عن النداءين `(3)` و`(4)`؟

```
let obj, method;

obj = {
  go: function() { alert(this); }
};

obj.go();               // (1) [object Object]
(obj.go)();             // (2) [object Object]
(method = obj.go)();    // (3) undefined
(obj.go || obj.stop)(); // (4) undefined
```

الحل:

إليك تفسير ما حدث.

1. هذا هو الطريقة العادية لاستدعاء تابع الكائن.
2. نفس الأمر يتكرر هنا، الأقواس لا تغير ترتيب العمليات هنا، النقطة هي الأولى على أي حال.
3. هنا لدينا استدعاء أكثر تعقيدًا `(expression).method()`. يعمل الاستدعاء كما لو كانت مقسمة على سطرين:

```
f = obj.go; // يحسب نتيجة التعبير
f();        // يستدعي ما لدينا
```

هنا تُنفذ `f()` كدالة، بدون `this`. مثل ما حدث مع الحالة `(3)` على يسار النقطة `.` لدينا تعبير.

لشرح سلوك `(3)` و`(4)` نحتاج إلى أن نتذكر أن توابع الوصول (accessors) للخاصيات (النقطة أو الأقواس المربعة) تُعيد قيمة النوع المرجعي.

أي عملية نطبقها عليها باستثناء استدعاء التابع (مثل عملية الإسناد `=` أو `||`) تحولها إلى قيمة عادية، والّتي لا تحمل المعلومات التي تسمح لها بتعيين قيمة `this`.

14.5 النوع BigInt: الأعداد الكبيرة

هذه إضافة حديثة للغة، ويمكنك العثور على الحالة الحالية للدعم من هنا.

الأعداد الكبيرة BigInt هو متغيّر عددي خاص، يوفر دعمًا للأعداد الصحيحة ذات الطول العشوائي.

تُنشأ الأعداد الكبيرة من خلال إلحاق الحرف n بنهاية العدد العادي، أو من خلال استدعاء الدالّة BigInt والّتي بدورها ستُنشئ عدد كبير من السلاسل أو الأعداد العادية وما إلى ذلك.

```
const bigint = 1234567890123456789012345678901234567890n;

const sameBigint = BigInt("1234567890123456789012345678901234567890");

const bigintFromNumber = BigInt(10); // 10n   مشابه تمامًا للطريقة
```

14.5.1 المعاملات الرياضية

عمومًا يمكننا استخدام العدد الكبير مثل العدد العادي، فمثلًا:

```
alert(1n + 2n); // 3

alert(5n / 2n); // 2
```

الرجاء ملاحظة أن القسمة 5/2 تُعيد نتيجة مُقرّبة للصفر، بدون الجزء العشري. جميع العمليات على الأعداد الكبيرة ستُعيد أعداد كبيرة.

لا يمكننا جمع الأعداد الكبيرة مع الأعداد العادية:

```
alert(1n + 2); // خطأ
```

يجب علينا تحويلها بطريقة واضحة إن لزم الأمر: باستخدام BigInt() أو Number()، هكذا:

```
let bigint = 1n;
let number = 2;

// تحويل عدد عادي إلى عدد كبير
alert(bigint + BigInt(number)); // 3

// تحويل عدد كبير إلى عدد عادي
```

```
alert(Number(bigint) + number); // 3
```

تكون عمليات التحويل صامتة دائمًا، ولا تخطئ أبدًا، ولكن إذا كانت الأعداد الكبيرة ذات حجم كبير جدًا بحيث لن تتناسب مع العدد العادي، فستُحذف البتات الإضافية من العدد الكبير، لذلك يجب أن نكون حذرين عند إجراء مثل هذه التحويلات.

لاحظ أن معامل الجمع الأحادي لا يطبق على الأعداد الكبيرة، فالمعامل `+value`: هو طريقة معروفة لتحويل المتغير `value` إلى رقم، ولا تدعم الأعداد الكبيرة هذه الطريقة لتجنب الفوضى:

```
let bigint = 1n;
alert( +bigint ); // خطأ
```

لذلك يجب أن نستخدم `Number()` لتحويل العدد الكبير إلى عدد عادي.

14.5.2 عمليات الموازنة

تعمل عمليات الموازنة، مثل: `<` و`>` مع الأعداد الكبيرة والعادية على حدٍ سواء:

```
alert( 2n > 1n ); // true
alert( 2n > 1 ); // true
```

لاحظ أنه نظرًا لأن الأعداد العادية والأعداد الكبيرة تنتميان لأنواع مختلفة، فيمكن أن تكون متساوية `==` ولكن ليست متساوية تمامًا `===`:

```
alert( 1 == 1n ); // true
alert( 1 === 1n ); // false
```

14.5.3 العمليات المنطقية

تتصرف الأعداد الكبيرة مثل الأعداد العادية عندما تكون أداخل الجملة الشرطية `if` أو أي عمليات منطقية الأخرى. فمثلًا، في الجملة الشرطية `if` أدناه، تكون قيمة `0n` خاطئة، والقيم الأخرى صحيحة:

```
if (0n) {
  // لن يُشغّل مطلقًا
}
```

تعمل المعاملات المنطقية مثل: `||` و`&&` وغيرهما مع الأعداد الكبيرة بطريقة مشابهة للأعداد العادية:

```
alert( 1n || 2 ); // 1 (1n is considered truthy)
alert( 0n || 2 ); // 2 (0n is considered falsy)
```

14.5.4 ترقيع مشاكل نقص الدعم

تعد عملية ترقيع الدعم للأعداد الكبيرة صعبة بعض الشيء. والسبب هو أن العديد من معاملات جافاسكربت، مثل: + و - وما إلى ذلك تتصرف بطريقة مختلفة مع الأعداد الكبيرة بالموازنة مع الأعداد العادية.

فمثلًا، تُعيد عملية القسمة على الأعداد الكبيرة دائمًا أعداد كبيرة (وتقربها إن لزم الأمر).

لتنفيذ مثل هذه السلوك عن طريق ترقيع نقص الدعم في هذه الحالة سنحتاج لتحليل الشيفرة واستبدال جميع هذه المعاملات بالدوالّ المناسبة. لكن القيام بذلك أمر مُرهق وسيُكون على حساب انخفاض الأداء.

بالإضافة إلى ذلك لا يوجد طريقة شائعة ومعتمد بين مجتمع المطورين لترقيع نقص الدعم.

على الرغم من ذلك، هنالك بعض المحاولات الجيدة لحل هذه المشكلة مثل المكتبة JSBI.

تزودنا هذه المكتبة بالأعداد الكبيرة من خلال توابعها الخاصة. ويمكننا استخدامها بدلًا من الأعداد الكبيرة الأصيلة -بدون أي طريقة لترقيع لنقص الدعم:-

العملية في مكتبة JSBI	العملية في الأعداد الكبيرة الأصيلة	العملية
`a = JSBI.BigInt(789)`	`a = BigInt(789)`	إنشاء عدد كبير من عدد عادي
`c = JSBI.add(a, b)`	`c = a + b`	الجمع
`c = JSBI.subtract(a, b)`	`c = a - b`	الطرح
...	...	...

ثم نستخدم لترقيع الدعم ملحقات إضافية (مثل الملحق الإضافي Babel) وذلك لتحويل استدعاءات المكتبة JSBI إلى الأعداد الكبيرة الأصيلة للمتصفحات التي تدعمها.

بتعبير آخر، يقترح هذا النهج أن نكتب الشيفرة في JSBI بدلًا من الأعداد الكبيرة الأصيلة. وتتعامل مكتبة JSBI مع الأعداد العادية والأعداد الكبيرة داخليًا، إذ تحاكي طريقة الاستخدام الصحيحة باتباع المواصفات عن كثب، لذلك ستكون الشيفرة جاهزة للتعامل مع الأعداد الكبيرة.

يمكننا استخدام شيفرة البرمجية للمكتبة JSBI على حالها للمحركات التي لا تدعم الأعداد الكبيرة والّتي تدعمها على حدٍ سواء - إذ ستُرقع نقص الدعم بتحويل الاستدعاءات إلى الأعداد الكبيرة الأصيلة.

14.5.5 المصادر

- موقع مطوري موزيلا MDN
- المواصفات القياسية للمتغيّر

أحدث إصدارات أكاديمية حسوب