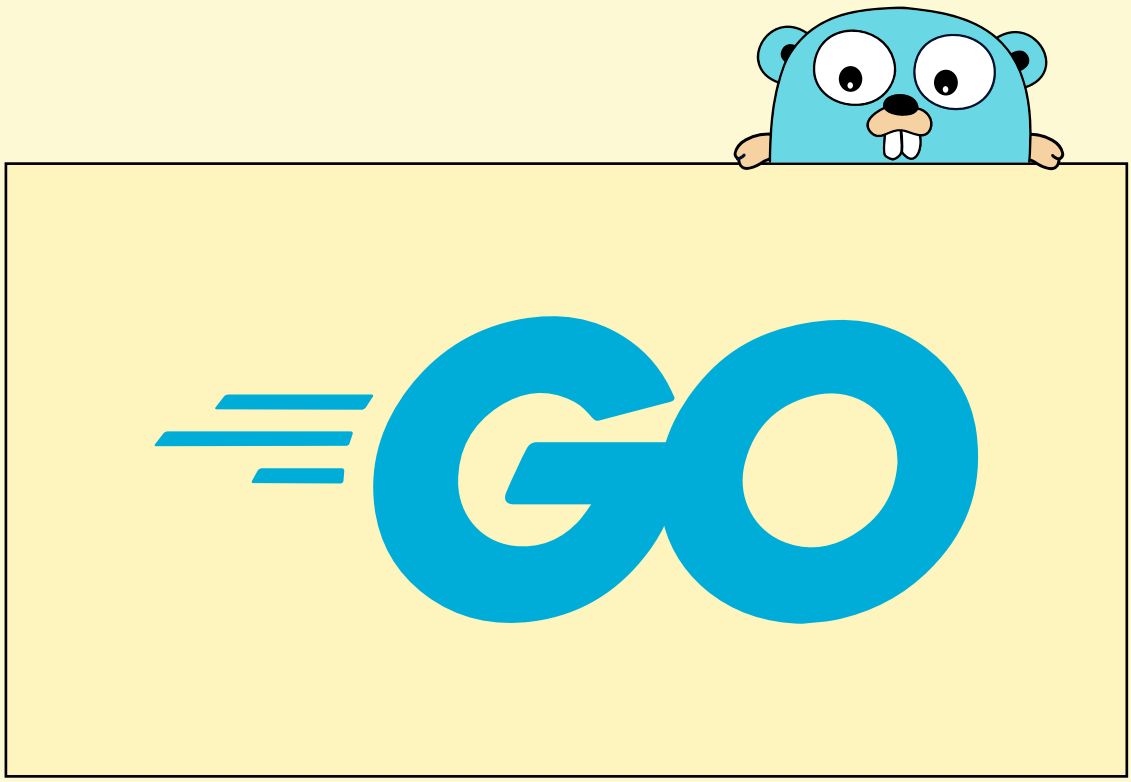


تعلم البرمجة بلغة Go

تأليف

مجموعة من المؤلفين

ترجمة

هدى جبور

أكاديمية حسوب

تعلم البرمجة بلغة Go

تعرف على المفاهيم الأساسية للغة البرمجة Go وكتابة البرامج باستخدامها

اسم الكتاب: تعلم البرمجة بلغة Go

Book Title: How To Code in Go

المؤلف: جوفر جايدس – جامون كاميسو

Author: Gopher Guides - Jamon Camisso

المترجم: هدى جبور

Translator: Huda Jabour

المحرر: غفار الرفاعي - جميل بيلوني

Editor: Ghefar Alrefai - Jamil Bailony

تصميم الغلاف: علا صالح

Cover Design: Ola Saleh

سنة النشر: 2024

Publication Year: 2024

رقم الإصدار: 1.0

Edition: 1.0

https://academy.hsoub.com

academy@hsoub.com

عن الناشـر

أُنتج هذا الكتاب برعاية شركة حسوب وأكاديمية حسوب.

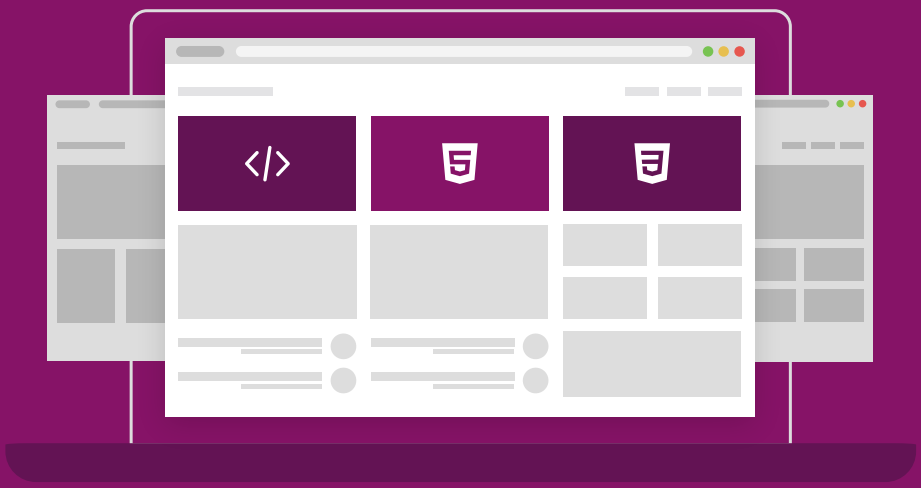
تهدف أكاديمية حسوب إلى تعليم البرمجة باللغة العربية وإثراء المحتوى البرمجي العـربي عـبر توفـير دورات برمجة وكتب ودروس عالية الجودة من متخصصين في مجال البرمجة والمجالات التقنية الأخـرى، بالإضـافة إلى توفير قسم للأسئلة والأجوبة للإجابة على أي سؤال يواجه المتعلم خلال رحلته التعليمية لتكون معه وتؤهلـه حـتى دخول سوق العمل.

حسوب شركة تقنية في مهمة لتطوير العالم العـربي. تبـني حسـوب منتجـات تركِّـز على تحسـين مسـتقبل العمل، والتعليم، والتواصل. تدير حسوب أكبر منصتي عمل حر في العالم العربي، مستقل وخمسات ويعمل في فيها فريق شاب وشغوف من مختلف الدول العربية.

المحتويات باختصار

جدول المحتويات

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

تمهيد

نضع بين أيديكم كتاب تعلم البرمجة بلغة Go والمعروفة أيضًا بلغة GoLang والتي تتميز بكونها لغة برمجة حديثة وذات قواعد syntax عالية المستوى على غرار لغات البرمجة النصية مثل بايثون وروبي وجافاسكربت، وقد طوّرتها شركة جوجل Google عام 2007 لتكون ملائمةً لنوعية احتياجات جوجل الحسابية من حيث التصريف compilation السريع وسهولة البرمجة والتنفيذ الفعّال.

تُعَدّ جو لغة برمجة بسيطة، فعدد الكلمات المفتاحية بها والأنواع الأساسية فيها ضئيل مقارنة بباقي اللغات، كما أنها تقلل كثيرًا من فكرة وجود طرق متعددة لتنفيذ مهمة ما، حتى أنها لا تحتوي على حلقة while وتقتصر فقط على حلقة for، مما يجعل هذه اللغة سهلة التعلم ومناسبةً للمبرمجين الجدد والخبراء ويميّزها عن باقي اللغات، كما تعالِج جو عمليات التزامن بصورة مبتكرة، بالإضافة إلى توفير الأدوات اللازمة لبناء ملفات ثنائية أصيلة native binaries مثل برامج تنفيذية executables أو مكتبات مشتركة shared libraries لاستخدامها في منصات وأماكن أخرى.

كما تتميز لغة جو بكونها لغة برمجة متعددة الاستخدامات يمكن استخدامها في العديد من مشاريع البرمجة، إلا أنها مناسبة بصورة خاصة لبرامج الشبكات والأنظمة الموزعة، ومن هذا المُنطلق اكتسبت لقب "لغة السحابة"، كما تركز لغة جو على مساعدة المبرمجين من خلال تقديم مجموعة مميزة من الأدوات وجعل التنسيق جزءًا من مواصفات اللغة وتسهيل النشر عن طريق تحويل البرنامج إلى ملف تنفيذي.

حول الكتاب

هذا الكتاب مترجم عن سلسلة How To Code in Go لمجموعة من المؤلفين المساهمين في شركة جوفر جايدس Gopher Guides، وهو يشرح لك بالتفصيل كيفية كتابة البرامج بلغة البرمجة Go وكتابة أدوات وتطبيقات مفيدة يمكن تشغيلها على خوادم بعيدة أو حواسيب محلية وهو مرخص بموجب رخصة المشاع الإبداعي CC BY-NC-SA 4.0.

يتكون هذا الكتاب من عدة فصول ويفترض أنك لا تمتلك أي خبرة مسبقة في لغة جو Go حيث يبدأ معك من إعداد بيئة التطوير ويشرح بالتفصيل خطوات تثبيت وإعداد بيئة تطوير Go على جهاز محلي يعمل على أنظمة تشغيل مختلفة مثل لينكس Linux أو ويندوز Windows أو ماك أو إس macOS.

بمجرد إعدادك لبيئة التطوير يمكنك الانتقال للفصول التالية التي تشرح لك كافة المفاهيم التي تحتاجها من أجل كتابة التطبيقات بدءًا من أساسيات لغة جو كالتعليمات الشرطية وتعليمات التحكم، ثم توضح أهم هياكل البيانات الخاصة باللغة وطريقة تعريفها وإنشاء واجهات لها، وطريقة كتابة دوال برمجية مخصصة للتعامل مع الأخطاء بكفاءة وفعالية، ثم تنتقل لمواضيع متقدمة تساعدك في كتابة تطبيقات متقدمة عالية الكفاءة.

أضفنا المصطلحات الأجنبية بجانب المصطلحات العربية لسببين، أولهما التعرف على المصطلحات العربية المقابلة للمصطلحات الأجنبية الأكثر شيوعًا وعدم الخلط بين أي منها، وثانيًا تأهيلك للاطلاع على المراجع فتصبح محيطًا بعد قراءة الكتاب بالمصطلحات الأجنبية التي تخص لغات البرمجة بالعموم ولغة جو على وجه الخصوص، 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.

المساهمة

يرجى 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، وتُعد إضافة أرقام الصفحات والأقسام جيدة أيضًا.

1. تثبيت لغة جو في بيئة برمجة محلية

سنشرح لك في هذا الفصل من الكتاب خطوات تثبيت إصدار أحدث من لغة جو Go على جهـاز الحاسـوب الخاص بك من أجل مختلف أنظمة التشغيل ونكتب برنامج بسيط للتأكد من نجاح عملية التثبيت.

1.1 تثبيت لغة جو وإعداد بيئة برمجة محلية على أبونتو

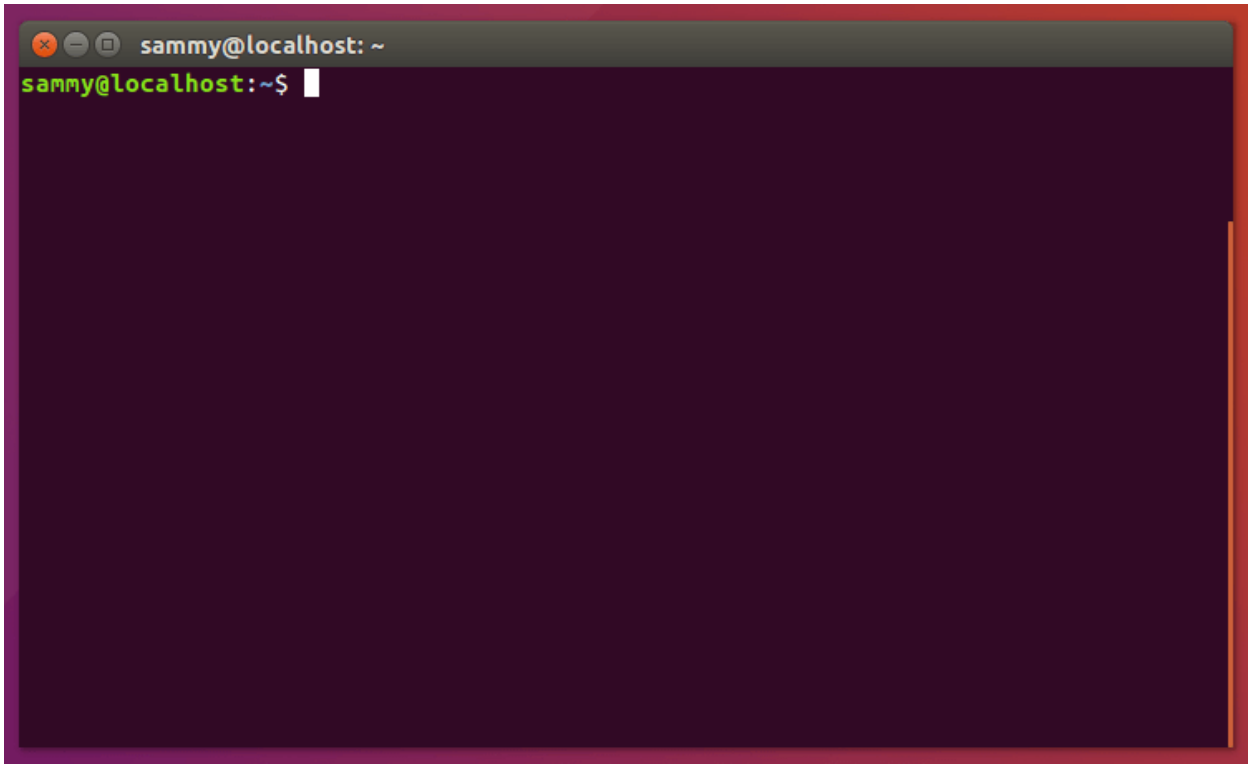
ا. المتطلبات

حاسوب -أو آلة افتراضية- مثبت عليه نظـام أبونتـو مـع إمكانيـة وصـول إداريـة administrative access واتصال بالإنترنت.

1.1.2 الخطوة 1: إعداد جو

الخطوة الأولى هي تثبيت جو عن طريق تحميل الإصدار الحالي من جو من الصفحة الرسـمية، لـذا يجب أن تحصل على عنوان URL لآخر نسـخة من ملـف `Tarball` (أو `tarfile` اسـم المجموعـة أو أرشـيف الملفـات المُصرّفة معًا باستخدام الأمر `tar`)، كما يجب عليك الانتباه إلى قيمة `SHA256` المدرَجة بجوارها لأنك ستحتاجها للتحقق من الملف الذي حُمِّل. في وقت كتابة هذا الكتاب كان أحدث إصدار `go1.16.7`.

سننجز عملية التثبيت من خلال سطر الأوامر command line -والمعـروف أيضًـا باسـم الصـدفة shell أو الطرفية Terminal وهو وسيلة تخاطب مع الحاسوب عبر كتابة الأوامر له- والـذي يُمكّنـك من تعـديل وأتمتـة العديد من المهام التي تؤديها على جهاز الحاسوب كل يوم، وهو أداة أساسية لمطورِي البرامج.



يمكنك العثور على تطبيق سطر الأوامر من خلال النقر على أيقونة أبونتو في الزاوية العلوية اليسرى من شاشتك وكتابة Terminal في شريط البحث ثم انقر على أيقونة تطبيق الطرفية لفتحه أو يمكنك الضغط على مفاتيح Ctrl+ALT+T على لوحة المفاتيح في الوقت نفسه لفتح تطبيق الطرفية تلقائيًا.

يمكنك تثبيت الملفات التنفيذية (الثنائية) binaries لجو مباشرةً بعد فتح الطرفية، إذ يمكنك استخدام مدير حزم مثل `apt-get` أو اتباع خطوات التثبيت اليدوي وهو المُستحسَن لتكون قادرًا على فهم وإجراء أيّة تعديلات مطلوب ضبطها في نظامك لكي يعمل جو بطريقة صحيحة.

تأكد من أنك في المجلد `home (~)` قبل بدء تحميل جو:

```
$ cd ~
```

استخدم الأمر `curl` للحصول على عنوان URL الخاص بالملف tarball الذي نسخته من الصفحة الرسمية للغة البرمجة جو:

```
$ curl -OL https://golang.org/dl/go1.16.7.linux-amd64.tar.gz
```

استخدم التجزئة `sha256sum` للتحقق من صلاحية ملف tarball:

```
$ sha256sum go1.16.7.linux-amd64.tar.gz
```

يجب أن تتطابق التجزئة التي عُرِضَت بعد تنفيذ الأمر السابق مع التجزئة الموجودة في صفحة التحميلات في الصفحة الرسمية لجو وإلا فهذا ليس ملفًا صالحًا ويجب عليك تحميل الملف مرةً أخرى.

```
go1.16.7.linux-amd64.tar.gz
7fe7a73f55ba3e2285da36f8b085e5c0159e9564ef5f63ee0ed6b818ade8ef04
go1.16.7.linux-amd64.tar.gz
```

استخرِج بعد ذلك الأرشيف الذي حمّلت وثبّته في الموقع المطلوب على النظام، ويوصى في المسار `/usr/local`. يتضمن هذا الأمر الراية `-C` التي ترشد tar إلى المجلد المحدد قبل تنفيذ أي عمليات أخرى.

```
$ sudo tar -C /usr/local -xvf go1.16.7.linux-amd64.tar.gz
```

أصبح لديك الآن مجلدًا باسم `go` في المسار `/usr/local`، وبذلك تكون قد حمّلت وثبّت جو على نظام أبونتو الخاص بك.

1.1.3 الخطوة 2 : إنشاء مساحة العمل Workspace الخاصة بك

يمكنك إنشاء مساحة العمل الخاصة بك بعد إكمال الخطوة الأولى، إذ ستحتوي على مجلدين في جذرها:

- `src`: مجلد ستوضع فيه ملفات جو المصدرية، وهي الملفات التي تُكتب وتُنشأ باستخدام لغة جو، إذ سيستخدِمها مُصرِّف جو لإنشاء ملفات قابلة للتنفيذ أي ملفات ثنائية يمكن تشغيلها على نظامك لتنفيذ المهام التي تتضمنها.
- `bin`: مجلد ستوضع فيه الملفات الثنائية التي أنشِئت وثُبِّتت بواسطة أدوات جو؛ بعبارة أخرى هي البرامج التي صُرِّفت من التعليمات البرمجية المصدرية الخاصة بك أو غيرها من التعليمات البرمجية المصدرية المرتبطة بجو والتي حمّلتها.

من المحتمل أن يحتوي المجلد `src` على عدة مستودعات للتحكم في الإصدارات مثل `Git` و `Mercurial` و `Bazaar`، إذ سيسمح لك هذا باستيراد أساسي canonical import للشيفرة البرمجية في مشروعك، والاستيراد الأساسي هو عملية استيراد تشير إلى حزمة مؤهلة وجاهزة بالكامل مثل `github.com/digitalocean/godo`.

سترى مجلدات مثل `github.com` أو `golang.org` أو غيرها عندما يستورد برنامجك مكتبات تابعة لجهات خارجية، فإذا كنت تستخدِم -أو لديك- شيفرات برمجية على إحدى المواقع مثل github.com، فستضع أيضًا هذه المشاريع أو ملفات المصدر ضمن هذا المجلد وستتعرّف على ذلك بعد قليل.

تبدو مساحة العمل النموذجية كما يلي:

```
.
├── bin
│   ├── buffalo                # أمر تنفيذي
│   ├── dlv                    # أمر تنفيذي
│   └── packr                  # أمر تنفيذي
└── src
    └── github.com
        └── digitalocean
            └── godo
                ├── .git               # البيانات الوصفية لمستودع جيت
                ├── account.go         # ملف الحزمة
                ├── account_test.go    # اختبار ملف الحزمة
                ├── ...
                ├── timestamp.go
                ├── timestamp_test.go
                └── util
                ├── droplet.go
                └── droplet_test.go
```

يُعَدّ المجلد الافتراضي لمساحة العمل في جو بدءًا من الإصدار 1.8 هو المجلد الرئيسي `home` للمستخدِم الذي يحتوي على مجلد فرعي باسم `go` أي `$HOME/go`، فإذا استخدَمت إصدارًا أقدم من 1.8، فمن الأفضل الاستمرار في استخدام الموقع `$HOME/go` لمساحة عملك.

نفّذ الأمر التالي لإنشاء بنية مجلد لمساحة العمل الخاصة بك في جو:

```
$ mkdir -p $HOME/go/{bin,src}
```

يطلب الخيار `-p` من `mkdir` إنشاء جميع العناصر الرئيسية parents في المجلد حتى لو لم تكن موجودة حاليًا، إذ يؤدي استخدام `{bin,src}` إلى إنشاء وسيطين لـ `mkdir` وإخباره بإنشاء كل من مجلد `bin` و `src`.

سيؤدي ذلك إلى إنشاء بنية المجلد التالية:

```
└── $HOME
    └── go
        ├── bin
        └── src
```

سابقًا وقبل الإصدار 1.8 كان يجب عليك استخدام متغير بيئة محلي يسمى `$GOPATH` من أجل تحديد المكان الذي يمكن للمُصرِّف فيه العثور على الشيفرة المصدرية المطلوب استخدامها في مشروعك، سواءً كانت

الشيفرة محلية أو على موقع استضافة خارجي (عمومًا، تُعَدّ هذه الطريقة جيدةً أكثر كما أن بعض الأدوات مازالت تعتمد على استخدامه).

يمكن ضبط المتغير `$GOPATH` الخاص بك من خلال إضافة المتغيرات العامة إلى `~/.profile` وربما تحتاج أيضًا إلى إضافته إلى ملف `.zshrc` أو `.bashrc` تبعًا لتهيئة الصدَفة shell الخاصة بك.

أولًا، افتح `~/.profile` باستخدام nano أو محرر النصوص المفضل لديك:

```
$ nano ~/.profile
```

حدِّد المتغير `$GOPATH` الخاص بك من خلال إضافة ما يلي إلى الملف:

```
export GOPATH=$HOME/go
```

عندما يُصرّف جو الأدوات ويثبّتها، فسيضعها في المجلد `$GOPATH/bin`، ومن الشائع إضافة المجلد الفرعي `/bin` الخاص بمساحة العمل إلى `PATH` في `~/.profile` كما يلي:

```
export PATH=$PATH:$GOPATH/bin
```

سيسمح لك ذلك بتشغيل أيّ برامج مُصرَّفة بواسطة جو أو محمّلة عبر أدوات جو من أيّ مكان على نظامك.

أخيرًا، ستحتاج إلى إضافة المجلد `bin` إلى المسار PATH من خلال إضافة المسار `/usr/local/go/bin` في نهاية السطر كما يلي:

```
export PATH=$PATH:$GOPATH/bin:/usr/local/go/bin
```

أصبح بإمكانك الآن الوصول إلى جميع أدوات جو من أيّ مكان على نظامك.

نفّذ الأمر التالي لتحميل المتغيرات العامة وتحديث ضبط الصدَفة:

```
. ~/.profile
```

يمكنك التحقق من تحديث المتغير `$PATH` باستخدام الأمر `echo` وقراءة الخرج:

```
$ echo $PATH
```

ستشاهد `$GOPATH/bin` الخاص بك والذي سيظهر في مجلد `home`، فإذا سجّلت الدخول على أساس جذر، فسترى `/root/go/bin` في المسار.

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/
games:/usr/local/games:/snap/bin:/root/go/bin:/usr/local/go/bin
```

ستشاهد أيضًا المسار الخاص بأدوات جو ضمن المجلد `/usr/local/go/bin`:

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/
games:/usr/local/games:/snap/bin:/root/go/bin:/usr/local/go/bin
```

تحقق الآن من التثبيت عن طريق التحقق من الإصدار الحالي من جو:

```
$ go version
```

سيظهر لك خرج يشبه التالي:

```
go version go1.12.1 linux/amd64
```

الآن بعد أن أنشأتَ المجلد الجذر لمساحة العمل وضبطت مجموعة متغيرات البيئة `$GOPATH` الخاصة بك، أصبح بإمكانك إنشاء مشاريعك باستخدام بنية المجلد التالية، وسيفترض هذا المثال أنك تستخدِم موقع github.com لاستضافة مشروعك:

```
$GOPATH/src/github.com/username/project
```

إذا كنت تعمل على مشروع https://github.com/digitalocean/godo على سبيل المثال، فسيُخزَّن في المجلد التالي:

```
$GOPATH/src/github.com/digitalocean/godo
```

ستجعل هذه البنية المشاريع متاحةً باستخدام أداة `go get`، كما أنها ستساعدك أيضًا في عمليات القراءة، كما يمكنك الوصول إلى مكتبة `godo` من خلال الأمر السابق كما يلي:

```
$ go get github.com/digitalocean/godo
```

سيؤدي الأمر السابق إلى تحميل كامل محتويات المكتبة godo، وإنشاء المجلد `$GOPATH/src/github.com/digitalocean/godo` على جهازك.

يمكنك أيضًا التحقق من نجاح عملية تحميل حزمة godo من خلال سرد محتويات المجلد:

```
$ ll $GOPATH/src/github.com/digitalocean/godo
```

يجب أن تشاهد خرجًا يشبه الخرج التالي:

```
drwxr-xr-x 4 root root 4096 Apr 5 00:43 ./
drwxr-xr-x 3 root root 4096 Apr 5 00:43 ../
drwxr-xr-x 8 root root 4096 Apr 5 00:43 .git/
-rwxr-xr-x 1 root root    8 Apr 5 00:43 .gitignore*
-rw-r--r-- 1 root root   61 Apr 5 00:43 .travis.yml
-rw-r--r-- 1 root root 2808 Apr 5 00:43 CHANGELOG.md
```

```
-rw-r--r-- 1 root root 1851 Apr 5 00:43 CONTRIBUTING.md
.
.
-rw-r--r-- 1 root root 4893 Apr 5 00:43 vpcs.go
-rw-r--r-- 1 root root 4091 Apr 5 00:43 vpcs_test.go
```

بذلك تكون قد أنشأت مساحة عمل خاصة بك وهيأت متغيرات البيئة اللازمة، وسنختبر في الخطـوة التاليـة مساحة العمل هذه من خلال كتابة برنامج بسيط.

1.1.4 الخطوة 3: إنشاء برنامج بسيط في جو

ستنشئ برنامج "Hello, World!" بغية اختبار مساحة العمل والتعـرف أكـثر على جـو. ستنشـئ هنـا ملـف مصدري واحد لجو وليس مشروعًا فعليًا، لذا لا داعي لأن تكون ضمن مساحة العمل الخاصة بك لإنجاز ذلك.

افتح محرر نصوص سطر الأوامر nano من المجلد الرئيسي وأنشئ ملفًا جديدًا:

```
$ nano hello.go
```

اكتب برنامجك في الملف الجديد:

```
package main
import "fmt"
func main() {
  fmt.Println("Hello, World!")
}
```

تسـتخدِم هـذه الشـيفرة الحزمـة `fmt` وتسـتدعي الدالـة `Println` لطباعـة Hello, World! الممـررة كوسيط للدالة.

اخرج الآن من المحرر nano بالضـغط على مفتـاحَي Ctrl+X، وعنـد مطالبتـك بحفـظ الملـف، اضـغط على Y ثم ENTER.

شغِّل برنامج hello.go الذي أنشأته عند الخروج من nano والعودة إلى الصدَفة:

```
go run hello.go
```

سترى الخرج التالي:

```
Hello, World!
```

بذلك يكون اختبار مساحة عملك قد اكتمل، وتكون قد أنشأت مسـاحة عمـل خاصـة بـك لكتابـة الشـيفرات البرمجية وإنشاء المشاريع بلغة جو على جهاز يعمل بنظام تشغيل أبونتو.

1.2 تثبيت لغة جو وإعداد بيئة برمجة محلية على نظام ماك macOS

ستتعلم في هذه الفقرات كيفية تثبيت جو على حاسب يعمل بنظام ماك وإعداد مساحة عمل برمجية خاصة بك من خلال سطر الأوامر لبدء العمل مع جو.

1.2.1 المتطلبات

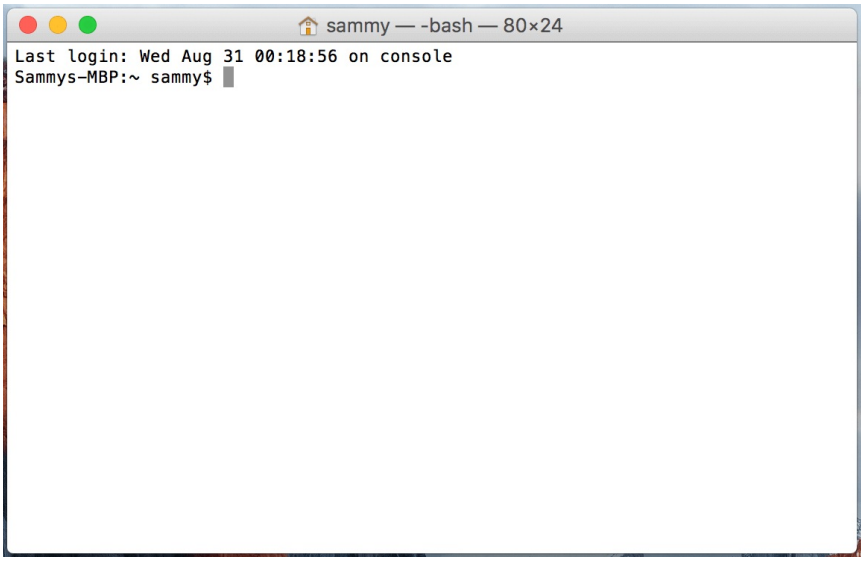
ستحتاج إلى حاسب يعمل بنظام ماك macOS مع إمكانية وصول إدارية administrative access واتصال بالإنترنت.

1.2.2 الخطوة 1: فتح الطرفية Terminal

سننجز عملية التثبيت والإعداد من خلال سطر الأوامر command line وهو معروف أيضًا باسم الصدَفة shell وهو وسيلة تخاطب مع الحاسوب عبر كتابة الأوامر له والذي يُمكّنك من تعديل وأتمتة العديد من المهام التي تقوم بها على جهاز الحاسوب كل يوم وهو أداة أساسية لمطورِي البرامج.

تُعَدّ طرفية نظام ماك تطبيقًا يمكنك استخدامه للوصول إلى واجهة سطر الأوامر، ومثل أي تطبيق آخر يمكنك العثور عليه بالذهاب إلى Finder ثم الانتقال إلى مجلد التطبيقات Applications ثم إلى مجلد الأدوات المساعدة Utilities، وانقر بعد ذلك نقرًا مزدوجًا فوق أيقونة الطرفية.

يمكنك أيضًا العثور على الطرفية من خلال فتح Spotlight عن طريق الضغط باستمرار على مفتاحَي CMD+SPACE وكتابتها في المربع الذي يظهر.



هناك العديد من الأوامر الخاصة بالطرفية والتي يمكنك تعلّمها من مقال مـدخل إلى طرفيّـة لينكس Linux Terminal، وبعد فتح تطبيق الطرفية يمكنك تحميل حزمة أدوات المطورXcode وتثبيتها، إذ ستحتاجها لتثبيت لغة جو.

1.2.3 الخطوة 2: تثبيت Xcode

تُعَدّ Xcode بيئة تطوير متكاملة integrated development environment أو IDE اختصارًا، وتحتـوي على أدوات تطوير البرامج لنظام ماك، كما يمكنك التحقق مما إذا كان Xcode مثبتًا بالفعل عن طريـق كتابـة مـا يلي في الطرفية:

```
$ xcode-select -p
```

يعني الخرج التالي أنّ Xcode مُثبّت:

```
/Library/Developer/CommandLineTools
```

إذا تلقيت خطأً ما، فثبّتXcode App Store من متجر App Store واضغط زر قبول الخيارات الافتراضية.

عُد إلى نافذة الطرفيـة بعـد تثـبيت Xcode، وبعـد ذلـك سـتحتاج إلى تثـبيت تطـبيق أدوات سـطر الأوامـر Command Line Tools الذي يخص Xcode عن طريق كتابة الأمر التالي:

```
$ xcode-select --install
```

إلى هنا تكون قد ثبّت كل من Xcode وتطبيق Command Line Tools وأدوات سطر الأوامر الخاص به بالكامل، ويمكنك الآن تثبيت مدير الحزم Homebrew.

1.2.4 الخطوة 3: تثبيت وإعداد Homebrew

على الرغم من احتواء طرفية نظام ماك على الكثـير من الوظـائف المفيـدة مثـل تلـك الموجـودة في أنظمـة لينكس ويونكس، إلا أنها لا تحتوي على مدير حزم الذي يُعَدّ مجموعةً من الأدوات البرمجية التي تعمل على أتمتة عمليات التثبيت بما في ذلك التثبيت الأولي للبرامج وترقيتهـا وضـبطها وإزالتهـا حسـب الحاجـة، كمـا يحتفـظ بالحزم المثبَّتة في موقع مركزي وحسب التنسيقات الشائعة.

يزوِّد Homebrew نظام ماك بنظام إدارة حزمة برمجيات مجانية ومفتوحة المصدر يعمل على تبسيط عملية تثبيت الحزم على نظام ماك، كما يمكنك تثبيته من خلال تنفيذ الأمر التالي في الطرفية:

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

كُتب Homebrew باستخدام لغة روبي Ruby، لذلك سيعدّل مسار Ruby على حاسبك، كما سيسحب الأمر `curl` برمجيةً أو سكربتًا من عنوان URL المحدد وسيطبع لك هذا السكربت رسالةً تشرح فيها ما الذي ستفعله وستطلب منك الإذن لتنفيذ ذلك، إذًا ستوفر لك تلك الرسائل إمكانية معرفة ما الذي ستنفّذه البرمجية وستمنحك الفرصة للتحقق والموافقة على التنفيذ.

قد يتطلب تنفيذ الأمر إدخال كلمة المرور الخاصة بك، وفي هذه الحالة يجب أن تدرك أنّ ضغطات المفاتيح التي تُجرّبها لن تظهر في الطرفية (تُخفى أثناء كتابتها)، وبعد الانتهاء من كتابة كلمة المرور ما عليك سوى الضغط على مفتاح العودة، ففي حال لم تُطلَب منك كلمة المرور، فاضغط على المحرف `Y` (أي نعم yes) كلما طُلب منك ذلك لتأكيد عملية التثبيت.

فيما يلي بعض الرايات flags المرتبطة بالأمر `curl`:

- الراية `-f` أو `--fail` تُخبر الطرفية بعدم تقديم مستند HTML عند حدوث أخطاء في الخادم.
- الراية `-s` أو `--silent` تُستخدَم لكتم الأمر `curl`، أي لن تظهر لك معلومات أثناء عملية التثبيت، وبالتالي لن تُظهِر لك مقياس التقدم، وعند دمج هذه الراية مع الراية `-S` أو `--show-error`، فسيُظهر لك `curl` رسالة خطأ في حالة الفشل.
- الراية `-L` أو `--location` ستُخبِر `curl` بأنه عليه إعادة طلب العنوان من المكان الجديد في حال أبلغ الخادم عن انتقال الصفحة المطلوبة إلى موقع مختلف.

ضع مجلد Homebrew في قمة متغير البيئة `PATH` بعد اكتمال عملية التثبيت، وذلك لضمان استدعاء عمليات تثبيت Homebrew بدلًا من الأدوات التي قد يحددها نظام macOS تلقائيًا والتي يمكن أن تتعارض مع بيئة التطوير التي نقوم بإنشائها.

يجب عليك إنشاء أو فتح ملف `~/.bash_profile` باستخدام محرر نصوص سطر الأوامر nano من خلال الأمر `nano` كما يلي:

```
$ nano ~/.bash_profile
```

اكتب الأمر التالي بعد فتح الملف في الطرفية:

```
export PATH=/usr/local/bin:$PATH
```

اضغط الآن باستمرار على مفتاح Ctrl+O لحفظ التغييرات، وعندما يُطلب منك ذلك اضغط على مفتاح RETURN، كما يمكنك الآن الخروج من nano بالضغط على مفتاحي Ctrl+X.

نشِّط هذه التغييرات عن طريق تنفيذ الأمر التالي:

```
$ source ~/.bash_profile
```

بذلك ستدخل التغييرات التي أجريتها على متغير البيئة PATH حيّز التنفيذ، كما يمكنك التأكد من تثبيت Homebrew بنجاح عن طريق كتابة ما يلي:

```
$ brew doctor
```

إذا لم تكن هناك حاجة إلى إجراء عمليات تحديث، فستحصل على الخرج التالي:

```
Your system is ready to brew.
```

بالنسبة لعمليات التحديث التي قد تُطالب بها، فربما تكون تحذيرًا لتشغيل أمر آخر مثل `brew update`، وذلك للتأكد من أنّ تثبيت Homebrew الخاص بك بدوره محدث.

بعد تجهيز Homebrew يمكنك تثبيت جو.

1.2.5 الخطوة 4: تثبيت جو

يمكنك البحث عن جميع الحزم المتاحة عن طريق Homebrew من خلال الأمر `brew search`، وهنا ستبحث عن الحزم أو الوحدات المتعلقة بجو:

```
$ brew search golang
```

> لم يُستخدم في هذا الكتاب بحث brew باستخدام كلمة go، أي أننا لم نكتب `brew search go` لأنه يُعيد عددًا كبيرًا جدًا من النتائج، فكلمة go عبارة عن كلمة صغيرة وتتطابق مع العديد من الحزم، وبالتالي أصبح من الشائع استخدام كلمة golang على أساس مصطلح بحث، كما تُعَدّ هذه ممارسة شائعةً عند البحث على الإنترنت عن مقالات متعلقة بجو أيضًا، وقد وُلِد مصطلح Golang من عنوان موقع اللغة الرسمي المستخدم سابقًا golang.org.

سيكون الخرج قائمةً من الحزم والوحدات المتعلقة بلغة جو كما ذكرنا:

```
golang  golang-migrate
```

يمكنك الآن تثبيت لغة جو عبر تنفيذ الأمر التالي:

```
$ brew install golang
```

ستعطيك الطرفية ملاحظات تتعلق بتثبيت جو، وقد يستغرق التثبيت بضع دقائق قبل اكتمال التثبيت.

اكتب ما يلي للتحقق من إصدار جو الذي ثبّته:

```
$ go version
```

سيطبع الأمر السابق الإصدار المُثبت لديك من جو وسيكون الإصدار الحديث والمستقر.

يمكنك لاحقًا تحديث جو من خلال تنفيذ الأوامر التالية لتحديث Homebrew ثم تحديث جو:

```
$ brew update
$ brew upgrade golang
```

سيحدِّث الأمر الأول صيغة Homebrew نفسها، وبالتالي ضمان حصولك على أحدث المعلومات للحزم التي تريد تثبيتها؛ أما الأمر الثاني فسيُحدّث الحزمة `golang` إلى أحدث إصدار متوفر.

من الممارسات الجيدة التأكد من أن إصدار جو المُثبت على جهازك مُحدّث أي لديك أحدث إصدار، وذلك من خلال الاطلاع على أحدث الإصدارات الجديدة وتحديثه بما يتوافق معها.

بعد تثبيت جو أصبحت جاهزًا لإنشاء مساحة عمل لمشاريع جو الخاصة بك.

1.2.6 الخطوة 5: إنشاء مساحة العمل الخاصة بك لبناء مشاريع جو

الآن يمكنك المتابعة وإنشاء مساحة عمل البرمجة الخاصة بك بعد أن ثبّتَّ Xcode و Homebrew و Go.

ستحتوي مساحة العمل على مجلدَين في جذرها:

- `src`: ستوضع فيه ملفات جو المصدرية، وهي الملفات التي تُكتَب وتُنشأ باستخدام لغة جو، كما يستخدِمها مُصرِّف جو لإنشاء ملفات قابلة للتنفيذ (ملفات ثنائية يمكن تشغيلها على نظامك لتنفيذ المهام التي تتضمنها).
- `bin`: ستوضع فيه الملفات القابلة للتنفيذ التي أنشئت وثبِّتت بواسطة أدوات جو، وبعبارة أخرى هي البرامج التي تم تصريفها بواسطة الشيفرة المصدرية الخاصة بك أو غيرها من الشيفرة المصدرية المرتبطة بجو والتي جرى تحميلها.

من المحتمل أن يحتوي المجلد `src` على عدة مستودعات للتحكم في الإصدارات مثل Git و Mercurial و Bazaar، ويسمح لك هذا باستيراد أساسي canonical import للشفرة في مشروعك، إذ يُعَدّ الاستيراد الأساسي عملية استيراد تشير إلى حزمة مؤهلة وجاهزة بالكامل مثل github.com/digitalocean/godo.

سترى مجلدات مثل `github.com` أو `golang.org` عندما يستورد برنامجك مكتبات خارجية، فإذا كنت تستخدم -أو لديك- شيفرات برمجية على إحدى المواقع مثل GitHub، فستضع أيضًا هذه المشاريع أو ملفات المصدر ضمن هذا المجلد وستتعرّف على ذلك بعد قليل.

ستبدو مساحة العمل النموذجية كما يلي:

```
.
├── bin
│   ├── buffalo                # أمر تنفيذي
│   ├── dlv                    # أمر تنفيذي
│   └── packr                  # أمر تنفيذي
```

```
└── src
    └── github.com
    └── digitalocean
        └── godo
            ├── .git              # البيانات الوصفية لمستودع جيت
            ├── account.go        # ملف الحزمة
            ├── account_test.go   # اختبار ملف الحزمة
            ├── ...
            ├── timestamp.go
            ├── timestamp_test.go
            └── util
            ├── droplet.go
            └── droplet_test.go
```

بدءًا من الإصدار 1.8 يُعَدّ المجلد الافتراضي لمساحة العمل في جو هو المجلد الرئيسي `home` للمستخدِم الذي يحتوي على مجلد فرعي باسم `go` أي `$HOME/go`، فإذا كنت تستخدِم إصدارًا أقدم من 1.8، فمن الأفضل الاستمرار في استخدام الموقع `$HOME/go` لمساحة عملك.

نفِّذ الأمر التالي لإنشاء بنية مجلد لمساحة العمل الخاصة بك في جو:

```
$ mkdir -p $HOME/go/{bin,src}
```

يطلب الخيار `-p` من `mkdir` إنشاء جميع العناصر الرئيسية parents في المجلد حتى لو لم تكن موجودة حاليًا، كما يؤدي استخدام `{bin,src}` إلى إنشاء مجموعة من الوسائط لـ `mkdir` وإخباره بإنشاء كل من مجلدَي `bin` و `src`.

سيؤدي ذلك إلى إنشاء بنية المجلد التالية:

```
└── $HOME
    └── go
    ├── bin
    └── src
```

سابقًا أي قبل الإصدار 1.8 كان يجب عليك استخدام متغير بيئة محلي يسمى `$GOPATH` من أجل تحديد المكان الذي يمكن للمترجم فيه العثور على الشيفرة المصدرية المطلوب استخدامها في مشروعك، سواءً كانت الشيفرة محلية أو على موقع استضافة خارجي، وتُعَدّ هذه الطريقة جيدةً أكثر كما أن بعض الأدوات مازالت تعتمد على استخدامه ولكن لم يعد يُطلب ذكر متغير البيئة ذاك صراحة.

يمكن ضبط المتغير `$GOPATH` الخاص بك من خلال إضافة المتغيرات العامة إلى `~/.bash_profile`.

أولًا، افتح `~/.bash_profile` باستخدام nano أو محرر النصوص المفضل لديك:

```
$ nano ~/.bash_profile
```

اضبط المتغير `$GOPATH` الخاص بك من خلال إضافة ما يلي إلى الملف:

```
export GOPATH=$HOME/go
```

عندما يُصرّف ويثبّت جو الأدوات، فإنه سيضعها في المجلد `$GOPATH/bin`، ومن الشائع إضافة المجلد الفرعي `/bin` الخاص بمساحة العمل إلى PATH في `~/.bash_profile`:

```
export PATH=$PATH:$GOPATH/bin
```

أضف الآن ما يلي إلى ملف `~/.bash_profile`:

```
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin
```

سيسمح لك ذلك بتشغيل أية برامج تصرفها أو تحملها عبر أدوات جو في أيّ مكان على نظامك.

نفّذ الأمر التالي لتحميل المتغيرات العامة وتحديث ضبط الصدَفة:

```
$ . ~/.bash_profile
```

يمكنك يمكنك التحقق من تحديث المتغير `$PATH` باستخدام الأمر `echo` وقراءة الخرج:

```
$ echo $PATH
```

ستشاهد `$GOPATH/bin` الذي سيظهر في مجلد `home`، فإذا سجلت الدخول على أساس مستخدِم عادي وليكن sammy، فسترى `/Users/sammy/go/bin` في المسار:

```
/Users/sammy/go/bin:/usr/local/sbin:/usr/local/bin:/usr/bin:/bin:/
usr/sbin:/sbin
```

الآن بعد أن أنشأت المجلد الجذر لمساحة العمل وضبطت مجموعة متغيرات البيئة `$GOPATH` الخاصة بك، أصبح بإمكانك إنشاء مشاريعك باستخدام بنية المجلد التالية، كما يفترض هذا المثال أنك تستخدِم موقع Github لاستضافة مشروعك:

```
$GOPATH/src/github.com/username/project
```

إذا كنت تعمل على مشروع https://github.com/digitalocean/godo مثلًا، فسوف تخزنه في المجلد التالي:

```
$GOPATH/src/github.com/digitalocean/godo
```

ستجعل بنية المشاريع هذه الوصول إلى هذه المشاريع متاحًا باستخدام أداة `go get`، كما أنها ستساعدك أيضًا في عمليات القراءة، إذ يمكنك الوصول مثلًا إلى مكتبة `godo` من خلال الأمر السابق كما يلي:

```
go get github.com/digitalocean/godo
```

يمكنك أيضًا التحقق من نجاح عملية التحميل من خلال عرض محتويات المجلد:

```
ls -l $GOPATH/src/github.com/digitalocean/godo
```

يجب أن تشاهد خرجًا يشبه الخرج التالي:

```
Output
-rw-r--r-- 1 sammy staff  2892 Apr 5 15:56 CHANGELOG.md
-rw-r--r-- 1 sammy staff  1851 Apr 5 15:56 CONTRIBUTING.md
.
.
.
-rw-r--r-- 1 sammy staff  4893 Apr 5 15:56 vpcs.go
-rw-r--r-- 1 sammy staff  4091 Apr 5 15:56 vpcs_test.go
```

بذلك تكون أنشأت مساحة عمل خاصة بك وضبطت متغيرات البيئة اللازمة، وفي الخطوة التالية سنختبر مساحة العمل هذه من خلال كتابة شيفرة برنامج بسيط.

1.2.7 الخطوة 6: إنشاء برنامج بسيط

ستنشئ برنامج "Hello, World!" بغية اختبار مساحة العمل والتعرف أكثر على جو، إذ ستنشئ هنا ملفًا مصدريًا واحد لجو وليس مشروعًا متكاملًا، لذا لا داعي لأن تكون ضمن مساحة العمل الخاصة بك لإنجاز ذلك.

افتح محرر نصوص سطر الأوامر nano من المجلد الرئيسي وأنشئ ملفًا جديدًا:

```
$ nano hello.go
```

اكتب برنامجك في الملف الجديد:

```
package main
import "fmt"
func main() {
  fmt.Println("Hello, World!")
}
```

ستستخدِم هذه الشيفرة حزمة `fmt` وستستدعي الدالة `Println` لطباعة عبارة "Hello, World!" التي مُرِّرت إلى الدالة.

اخرج الآن من المحرر nano بالضغط على مفتاحي Ctrl+X وعند مطالبتك بحفظ الملف اضغط على Y ثم ENTER، وبمجرد الخروج من nano والعودة إلى الصدَفة شغّل برنامج hello.go الذي أنشأته:

```
go run hello.go
```

سترى الخرج التالي:

```
Hello, World!
```

يكون بذلك اختبار مساحة العمل الخاصة بك قد أكتمل.

تهانينا، لقد أنشأت مساحة عمل خاصة بك لكتابة الشيفرات البرمجية وإنشاء المشاريع بلغة جو على جهاز يعمل بنظام تشغيل ماك.

تثبيت لغة جو وإعداد بيئة برمجة محلية على ويندوز

سنرشدك في هذه الفقرة إلى كيفية تثبيت جو على جهاز محلي بنظام تشغيل ويندوز وإعداد بيئة برمجة باستخدام سطر الأوامر.

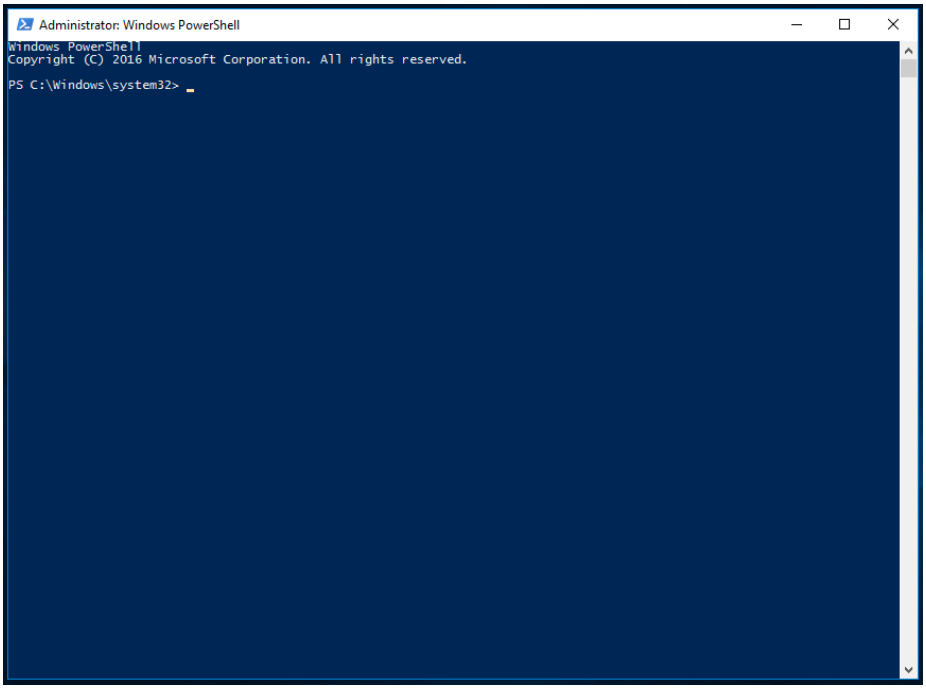
1.2.8 المتطلبات

جهاز مثبت عليه نظام ويندوز مع إمكانية الوصول على أساس مدير والاتصال بالإنترنت.

1.2.9 الخطوة 1: فتح وتهيئة PowerShell

سننجز عملية التثبيت والإعداد من خلال واجهة سطر الأوامر command line، ويُعَدّ PowerShell برنامجًا من مايكروسوفت يوفر واجهةً لصدفة سطر الأوامر، إذ تُنفَّذ المهام الإدارية عن طريق تشغيل أوامر `cmdlets` أي command-lets، وهي أصناف Classes متخصصة من إطار العمل .NET. الذي يمكنه تنفيذ العمليات، وقد أصبح PowerShell مفتوح المصدر في عام 2016، وهو متوفر الآن على أطر العمل لكل من نظامَي ويندوز ويونكس UNIX بما في ذلك ماك Mac ولينكس Linux.

يمكنك النقر بزر الفأرة الأيمن فوق رمز قائمة ابدأ في الزاوية اليسرى السفلية من شاشتك للعثور على PowerShell على جهازك، ثم انقر فوق بحث واكتب PowerShell في شريط البحث، ثم انقر بزر الفأرة الأيمن فوق Windows PowerShell، ولأسباب تتعلق بهذا الكتاب يُرجى فتحه على أساس مسؤول -أو مدير- من خلال اختيار Run as Administrator، وعندما ينبثق لك مربع حوار سيسألك هل تريد السماح لهذا التطبيق بإجراء تغييرات على جهاز الكمبيوتر الخاص بك؟ انقر فوق نعم، وبعد إجراء ما ذكرناه ستظهر لك النافذة التالية:



انتقل الآن إلى مجلد النظام من خلال الأمر التالي:

```
$ cd ~
```

سينقلك تنفيذ هذا الأمر إلى المجلد PS C:\Users\sammy.

يجب عليك الآن إعداد الأذونات من خلال PowerShell لمتابعة عملية التثبيت، ففي الحالة الافتراضية ستكون مهيأةً للتشغيل في الوضع الأكثر أمانًا، كما تجدر الإشارة إلى أنه هناك مستويات قليلة من الأذونات التي يمكنك إعدادها على أساس مسؤول:

- **المُقيّد Restricted**: هي سياسة التنفيذ الافتراضية، وفي هذا الوضع لن تتمكن من تشغيل البرامج النصية scripts، وسيعمل PowerShell فقط على أساس صدفة تفاعلية مع هذه البرامج.
- **الموقّع AllSigned**: سيمكنك هذا الوضع من تشغيل جميع البرامج النصية وملفات الضبط الموقَّعة عبر التوقيع الرقمي بواسطة ناشر موثوق، مما يعني أنه من المحتمل أن تُعرِّض جهازك لخطر تشغيل البرامج النصية الضارة التي وقّع عليها ناشر موثوق به.
- **الموقّعة عن بعد RemoteSigned**: يتيح لك تشغيل البرامج النصية وملفات الضبط المُحمّلة من الإنترنت والموقَّعة من قِبل ناشرين موثوقين، أي أنه يعرّض جهازك للخطر أيضًا إذا كانت هذه البرامج النصية الموثوقة ضارة.
- **بدون قيود Unrestricted**: ستُشغَّل جميع البرامج النصية وملفات الضبط التي نُزِّلت من الإنترنت عند تأكيدك أنّ الملف قد نُزِّل من الإنترنت، وهنا لا يلزم وجود توقيع رقمي، وبالتالي فإنك تُعرّض جهازك لخطر تشغيل البرامج النصية غير الموقّعة والتي من المحتمل أن تكون ضارةً.

سنعتمد في فقراتنا التالية على مستوى RemoteSigned في سياسة التنفيذ للمستخدِم الحالي، وبالتـالي سيمنح PowerShell إمكانية قبول البرامج النصية الموثوقة دون الحاجة إلى جعـل الأذونـات واسـعةً أكـثر مثـل المستوى بدون قيود.

نفّذ ما يلي ضمن PowerShell:

```
$ Set-ExecutionPolicy -Scope CurrentUser
```

سيطالبك PowerShell بعد ذلك بتقديم سياسة تنفيذ، لذا أدخل ما يلي لاستخدام RemoteSigned:

```
$ RemoteSigned
```

سيُطلب منك تأكيد التغيير في سياسة التنفيـذ بعـد الضـغط على ENTER، لـذا اكتب المحـرف Y للسـماح بتنفيذ التغييرات، كما يمكنك التأكد من نجاح ذلك من خلال الاستعلام عن الأذونات الحالية كما يلي:

```
$ Get-ExecutionPolicy -List
```

سيكون الخرج كما يلي:

```
    Scope ExecutionPolicy
    ----- ---------------
    MachinePolicy    Undefined
    UserPolicy    Undefined
    Process    Undefined
    CurrentUser  RemoteSigned
    LocalMachine    Undefined
```

هذا يؤكد أنّ المستخدِم الحالي يمكنه تشغيل نصـوص برمجيـة موثوقـة مُنزَّلـة من الإنـترنت، ويمكنـك الآن الانتقال إلى تنزيل الملفات التي سنحتاجها لإعداد بيئة برمجة جو.

1.2.10 الخطوة 2: تثبيت مدير الحزم شوكولاتي Chocolatey

يُعَدّ مدير الحـزم مجموعـةً من أدوات الـبرامج الـتي تعمـل على أتمتـة عمليـات التثـبيت، إذ يتضـمن ذلـك التثبيت الأولي للبرنامج وترقيته وتهيئته وإزالة البرامج حسب الحاجة، إذ يحتفظ مُدير الحزم بالبرامج المُثبّتـة في موقع مركزي ويمكنه صيانة جميع حزم البرامج في النظام وغيرها.

يُعَدّ شوكولاتي Chocolatey مدير حزم يمكن التفاعل معه من خلال سـطر أوامـر ومُصـمَّم لنظـام التشـغيل ويندوز ويعمل مثل `apt-get` على لينكس، كما يُعَدّ برمجية مفتوحة المصدر يساعدك على تثـبيت التطبيقـات والأدوات بسرعة، وستتعلم كيفية استخدامه لتنزيل ما تحتاجه لبيئة التطوير الخاصة بك.

اقرأ البرنامج النصي أو السكربت قبل تثبيته لتؤكّد على أنك موافق على التغييرات التي سيجريها على جهازك، ولإنجاز ذلك استخدم إطار عمل NET. لتنزيل وعرض البرنامج شوكولاتي ضمن نافذة الطرفية.

أنشئ كائن WebClient اسمه `$script` يشارك إعدادات الاتصال بالإنترنت مع Internet Explorer:

```
$ $script = New-Object Net.WebClient
```

ألق نظرةً على الخيارات المتاحة عن طريق تمرير كائن `$script` مع المحرف | إلى الصنف `Get-Member`:

```
$ $script | Get-Member
```

سيؤدي ذلك إلى عرض جميع الأعضاء -أي الخصائص والدوال- الخاصة بالكائن `WebClient`:

```
   . . .
[secondary_label Snippet of Output]
DownloadFileAsync         Method      void DownloadFileAsync(uri
address, string fileName), void DownloadFileAsync(ur...
DownloadFileTaskAsync     Method       System.Threading.Tasks.Task
DownloadFileTaskAsync(string address, string fileNa...
DownloadString      Method      string DownloadString(string address),
string DownloadString(uri address) #method we will use
DownloadStringAsync       Method      void DownloadStringAsync(uri
address), void DownloadStringAsync(uri address, Sy...
DownloadStringTaskAsync  Method  System.Threading.Tasks.Task[string]
DownloadStringTaskAsync(string address), Sy…
   . . .
```

بالنظر إلى الخرج، يمكنك تحديد طريقة `DownloadString` المستخدَمة لعرض البرنامج النصي أو السكربت والتوقيع في نافذة PowerShell، واستخدِم هذه الطريقة لفحص البرنامج النصي:

```
$ $script.DownloadString("https://chocolatey.org/install.ps1")
```

ثبّت شوكولاتي بعد فحص البرنامج النصي بكتابة ما يلي في PowerShell:

```
$ iwr https://chocolatey.org/install.ps1 -UseBasicParsing | iex
```

يسمح لك أمر `iwr` -وهو أحد أوامر `cmdlet`- أو `Invoke-WebRequest` باستخراج البيانات من الويب، إذ سيؤدي ذلك إلى تمرير البرنامج النصي إلى `iex`، أو `Invoke-Expression cmdlet`، والذي سينفذ محتويات البرنامج النصي ويشغّل عملية تثبيت مدير حزم شوكولاتي.

سيصبح بإمكانك بعد تثبيت شوكولاتي من خلال PowerShell البدء في تثبيت أدوات إضافية باستخدام أمر choco، وإذا كنت بحاجة إلى تحديث شوكولاتي في أيّ وقت في المستقبل، فنفّذ الأمر التالي:

```
$ choco upgrade chocolatey
```

يمكنك تثبيت ما تبقى من أدوات لتجهيز بيئة برمجة جو بعد تثبيت مدير الحزم.

1.2.11 الخطوة 3: تثبيت محرر النصوص Nano (خطوة اختيارية)

ستثبّت في هذه الخطوة محرر النصوص نانو nano، وهو محرر نصوص يستخدِم واجهة سطر الأوامر، كما يمكنك استخدام نانو لكتابة البرامج مباشرةً داخل PowerShell، وهذه الخطوة ليست إلزاميةً، إذ يمكنك أيضًا استخدام محرر نصوص بواجهة مستخدِم رسومية مثل Notepad، ولكن يوصي هذه الكتاب باستخدام نانو، إذ سيساعدك على اعتياد استخدام PowerShell.

استخدم شوكولاتي لتثبيت نانو:

```
$ choco install -y nano
```

تُستخدَم الراية y- للتأكيد التلقائي -أي الموافقة- على تشغيل البرنامج النصي.

يمكنك استخدام الأمر nano بعد تثبيت نانو لإنشاء ملفات نصية جديدة، كما ستستخدِمه لاحقًا في هذا الفصل لكتابة برنامج جو الأول الخاص بك.

1.2.12 الخطوة 4: تثبيت جو

ستستخدِم شوكولاتي كما في الخطوة السابقة لتثبيت جو:

```
$ choco install -y golang
```

سيثبّت PowerShell جو الآن وستشاهد الخرج التالي:

```
Environment Vars (like PATH) have changed. Close/reopen your shell to
see the changes (or in powershell/cmd.exe just type `refreshenv`).
The install of golang was successful.
 Software installed as 'msi', install location is likely default.
Chocolatey installed 1/1 packages.
See the log for details (C:\ProgramData\chocolatey\logs\
chocolatey.log).
```

يمكنك الآن التحقق من نجاح عملية التثبيت عن طريـق إغلاق PowerShell وإعـادة فتحـه مـرةً أخـرى على أساس مسؤول ثم تنفيذ الأمر التالي للتحقق من إصدار لغة جو المُثبَّت:

```
$ go version
```

سترى خرجًا مُشابهًا للخرج التالي:

```
go version go1.12.1 windows/amd643.7.0
```

حيث يعرض لك تفاصيل النسخة الحالية المثبتة على حاسـبك من جـو. يمكنـك الآن إعـداد مسـاحة عمـل لمشاريع التطوير الخاصة بك.

1.2.13 الخطوة 5: إنشاء مساحة العمل الخاصة بك لبناء مشاريع جو

يمكنك إنشاء مساحة عمل البرمجة الخاصة بك بعد تثبيت شوكولاتي ونانو ولغـة جـو، إذ سـتحتوي مسـاحة العمل على مجلدَين في جذرها وهما:

- `src`: مجلد ستوضَع فيه ملفات جو المصدرية وهي الملفات التي المكتوبة والمنشأة باستخدام لغة جو، إذ يستخدِمها مُصرِّف جو لإنشاء ملفات قابلة للتنفيذ وهي ملفـات ثنائيـة يمكن تشـغيلها على نظامـك لتنفيذ المهام التي تتضمنها).
- `bin`: مجلد ستوضَع فيه الملفات الثنائية التي أُنشِـئت وثبِّتَت بواسـطة أدوات جـو؛ وبعبـارة أخـرى هي الـبرامج الـتي المصـرَّفة بواسـطة الشـيفرة المصـدرية الخاصـة بـك أو غيرهـا من التعليمـات البرمجيـة المصدرية المرتبطة بجو والتي قد نزِّلت.

من المحتمـل أن يحتـوي المجلـد الفـرعي `src` على عـدة مسـتودعات للتحكم في الإصـدارات مثـل Git و Mercurial و Bazaar، وهذا يسمح لك باستيراد أساسي import canonical للشيفرة في مشـروعك وهـو عملية استيراد تشير إلى حزمة مؤهلة وجاهزة بالكامل مثل github.com/digitalocean/godo.

سترى مجلدات مثل `github.com` أو `golang.org` عندما يستورد برنامجك مكتبات خارجية، فـإذا كنت تستخدِم -أو لديك- شيفرات برمجية على إحدى المواقع مثل GitHub، فستضع أيضًا هذه المشاريع أو الملفـات المصدرية ضمن هذا المجلد وستتعرّف على ذلك بعد قليل.

ستبدو مساحة العمل النموذجية هكذا:

```
.
├── bin
│   ├── buffalo                    # أمر تنفيذي
│   ├── dlv                        # أمر تنفيذي
```

```
|   └── packr                    # أمر تنفيذي
└── src
      └── github.com
      └── digitalocean
      └── godo
            ├── .git             # البيانات الوصفية لمستودع جيت
            ├── account.go       # ملف الحزمة
            ├── account_test.go  # اختبار ملف الحزمة
            ├── ...
            ├── timestamp.go
            ├── timestamp_test.go
            └── util
            ├── droplet.go
            └── droplet_test.go
```

يُعَدّ المجلد الافتراضي لمساحة العمل في جو بدءًا من الإصدار 1.8 المجلد الرئيسي `home` للمستخدِم الذي يحتوي على مجلد فرعي باسم `go` أي `$HOME/go`، فإذا كنت تستخدِم إصدارًا أقدم من 1.8، فمن الأفضل الاستمرار في استخدام الموقع `$HOME/go` لمساحة عملك.

نفّذ الأمر التالي للانتقال إلى المجلد `$HOME`:

```
$ cd $HOME
```

أنشئ بعدها مجلدًا لبيئة العمل بتنفيذ الأمر التالي:

```
$ mkdir go/bin, go/src
```

سيؤدي ذلك إلى إنشاء بنية المجلد التالية:

```
└── $HOME
  └── go
    ├── bin
    └── src
```

سابقًا أي قبل الإصدار 1.8 كان يجب عليك استخدام متغير بيئة محلي يسمى `$GOPATH` من أجل تحديد المكان الذي يمكن للمُصرِّف فيه العثور على الشيفرة المصدرية المطلوب استخدامها في مشروعك، سواءً كانت الشيفرة محليةً أو على موقع استضافة خارجي (عمومًا، تعتبر هذه الطريقة جيدة أكثر كما أن بعض الأدوات ما زالت تعتمد على استخدامه) لكن لا يطلب ضبط متغير البيئة ذاك صراحة.

يجب أن يكون متغير البيئة `$GOPATH` قد حُدّدَ فعلًا نظرًا لأنك استخدَمت شـوكولاتي في عمليـة التثـبيت، ويمكنك التحقق من ذلك بالأمر التالي:

```
$ $env:GOPATH
```

يجب أن ترى الخرج التالي مع اسم المستخدِم الخاص بك بدلًا من sammy:

```
C:\Users\sammy\go
```

عنـدما يُصـرَّف ويثبّت جـو الأدوات، فسيضـعها في المجلـد `$GOPATH/bin`، وللسـهولة فإنـه من الشـائع إضافة المجلـد الفـرعي `bin` الخـاص بمسـاحة العمـل إلى `$PATH`، ويمكنـك إنجـاز ذلـك من خلال الأمـر `setx` في PowerShell:

```
$ setx PATH "$($env:path);$GOPATH\bin"
```

سيسمح لك ذلك بتشغيل أيّ برامج تصرّفها أو تنزّلها عبر أدوات جو في أيّ مكان على نظامك.

الآن وبعد أن أنشأت المجلد الجذر لمساحة العمل وضبطت مجموعـة متغـيرات البيئـة `$GOPATH` الخاصـة بك، فقد أصبح بإمكانك إنشاء مشاريعك المستقبلية باستخدام بنية المجلد التاليـة، ويفـترض هـذا المثـال أنـك تستخدِم موقع GitHub لاستضافة مشروعك:

```
$GOPATH/src/github.com/username/project
```

إذا كنت تعمل على مشروع https://github.com/digitalocean/godo، فسيُخزَّن في المجلد التالي:

```
$GOPATH/src/github.com/digitalocean/godo
```

إن هذه البنية للمشاريع ستجعل المشاريع متاحةً باستخدام الأمر `go get`، كما أنهـا ستسـاعدك أيضًـا في عمليات القراءة، إذ يمكنك الوصول إلى مكتبة `godo` مثلًا من خلال الأمر السابق كما يلي:

```
$ go get github.com/digitalocean/godo
```

> إذا لم يكن جيت git مُثبّتًا لديك، فسيفتح ويندوز مربع حوار يسألك عما إذا كنت تريد تثبيته، وعندها انقر فوق نعم yes للمتابعة واتبع إرشادات التثبيت.

يمكنك أيضًا التحقق من نجاح عملية التنزيل من خلال سرد محتويات المجلد:

```
$ ls -l $GOPATH/src/github.com/digitalocean/godo
```

يجب أن تشاهد خرجًا يشبه الخرج التالي:

```
    Directory: C:\Users\sammy\go\src\github.com\digitalocean\godo
Mode        LastWriteTime       Length Name
----          -------------     ------ ----
d-----        4/10/2019  2:59 PM        util
-a----        4/10/2019  2:59 PM        9 .gitignore
-a----        4/10/2019  2:59 PM         69 .travis.yml
-a----        4/10/2019  2:59 PM        1592 account.go
-a----        4/10/2019  2:59 PM        1679 account_test.go
-rw-r--r-- 1 sammy staff  2892 Apr 5 15:56 CHANGELOG.md
-rw-r--r-- 1 sammy staff  1851 Apr 5 15:56 CONTRIBUTING.md
.
.
.
-a----        4/10/2019  2:59 PM        5076 vpcs.go
-a----        4/10/2019  2:59 PM        4309 vpcs_test.go
```

تكون بذلك قد أنشأت مساحة عمل خاصة بك وضبطت متغيرات البيئة اللازمة، وفي الخطوة الثالثة سنختبر مساحة العمل هذه من خلال كتابة شيفرة برنامج بسيط.

1.2.14 الخطوة 6: إنشاء برنامج بسيط

ستنشئ برنامجًا بسيطًا بلغة جو Go يطبع العبارة "Hello, World!" بغية اختبار مساحة العمل والتعرف أكثر على جو، كما ستنشئ هنا ملفًا مصدريًا واحدًا لجو وليس مشروعًا متكاملًا، لذا لا داعي لأن تكون ضمن مساحة العمل الخاصة بك لإنجاز ذلك.

افتح محرر نصوص سطر الأوامر نانو من المجلد الرئيسي وأنشئ ملفًا جديدًا:

```
$ nano hello.go
```

اكتب برنامجك في الملف الجديد:

```
package main
import "fmt"
func main() {
      fmt.Println("Hello, World!")
}
```

ستستخدم هذه الشيفرة حزمة `fmt` وستستدعي الدالة `Println` لطباعة "Hello, World!" الممـررة على أساس وسيط للدالة.

اخرج الآن من المحرر nano بالضغط على مفتاحَي Ctrl+ X، وعند مطالبتك بحفظ الملـف، اضـغط على Y ثم ENTER، ثم شغّل برنامج hello.go الذي أنشأته عند الخروج من nano والعودة إلى الصدَفة:

```
$ go run hello.go
```

سترى الخرج التالي:

```
Hello, World!
```

بذلك يكون اختبار مساحة العمل الخاصة بك قد أكتمل.

1.3 الخاتمة

تهانينا، لقد انتهيت من الفصل الأول من كتاب البرمجة بلغة جو، والـذي تعلمت فيـه كيـف تثبت لغـة جـو على حاسوب محلي، وكيف تنشئ مساحة عمل خاصة بك لكتابة الشيفرات البرمجية وإنشاء المشاريع بلغـة جـو على جهاز يعمل بنظام تشغيل أبونتو أو ماك أو ويندوز، وأنت جاهز للانتقال إلى الفصل التالي من الكتاب وكتابة برامج مختلفة حول مفاهيم مختلفة حول لغة البرمجة جو.

دورة علوم الحاسوب

مميزات الدورة

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

اشترك الآن

2. كتابة برنامجك الأول

ستتعلم في هذا الفصل كيفية كتابة برنامج بسيط باستخدام لغة جـو، كمـا سـتتعرف على المجلـد GOPATH الذي يحتوي على الشيفرة المصدرية الخاصة بك وكيفية التعامل معه، وأخـيرًا سـتتعرف على مفهـوم التعليقـات وطريقة كتابتها في برامجك بشكل صحيح.

2.1 برنامجك الأول في جو

سنحاول اتباع العادة التي جرت عليها دروس تعلم لغات البرمجـة، وهي ببسـاطة كتابـة السلسـلة النصـية "Hello, World!" أي "أهلًا بالعالم!"، لكن سنجعل الأمور أكثر متعةً من خلال جعل البرنـامج يسـأل المسـتخدِم عن اسمه لكي يطبع الاسم بجانب عبارة الترحيب، وبعد الانتهاء من كتابة البرنامج سيكون خرج البرنامج مشـابهًا للخرج التالي:

```
Please enter your name.
Sammy
Hello, Sammy! I'm Go!
```

2.1.1 المتطلبات

يجب أن تكون قد أعددت بيئة تطوير جو على حاسبك، فإذا لم يكن لديك بيئة تطوير جاهزة، فيمكنك العودة إلى ما شرحناه في الفصل السابق واتباع تعليمات التثبيت حسب نظام التشغيل الذي تستخدمه.

2.1.2 الخطوة 1: كتابة برنامج "Hello, World!" الأساسي

افتح محرر نصوص سطر الأوامر مثل نانو nano وأنشئ ملفًا جديدًا:

```
$ nano hello.go
```

اكتب البرنامج التالي بعد فتح الملف النصي في الطرفية:

```
package main
import "fmt"
func main() {
 fmt.Println("Hello, World!")
}
```

لنشرح مكونات الشيفرة بالتفصيل:

- الحزمة `package`: هي كلمة مفتاحية keyword في جو تحدِّد اسم الحزمة التي ينتمي لها هذا الملف، أي الملف الذي تكتب فيه البرنامج، إذ لا بدّ أن يكون لكل ملف حزمة ما، ولا بدّ أن تنتمي كل حزمة في جو إلى مجلد ما، كما لا يمكن لحزمتَين التواجد على مستوى نفس المجلد، لكن يمكن لعدة ملفات أن تنتمي إلى نفس الحزمة (يعني مجلد به عدة ملفات)، كما يمكن أن تتواجد حزمة داخل حزمة أخرى لكن كلٌ في مجلد فرعي على حِدى. في مثالنا السابق، أعطينا `main` كاسم لحزمتنا، وهو اسم خاص، حيث يُعامل مُصرّف compiler لغة جو هذه الحزمة على أنها مدخل البرنامج، أي أن التعليمات الموجودة في هذه الحزمة يتم تشغيلها أولًا. أسبقنا اسم الحزمة بالكلمة المفتاحية `package`.
- `import` كلمة مفتاحية أخرى وتعني استرِد أو اجلِب المكتبة الفلانية أو الحزمة الفلانية.
- `fmt` اختصار format أو formatting وهي مكتبة قياسية standard library تأتي مع جو، وهي خاصة ببناء وطباعة النص. لاحظ أن اسم المكتبات أو الحزم المُراد استيرادها دائما ما يتم إحاطتها بعلامة اقتباس "".
- في التعبير `fmt.Println` استعملنا دالة `Println` من الحزمة `fmt` التي استوردناها، ولاستعمال أيّ حزمة في جو يكفي كتابة اسمها ثم نقطة ثم اسم الدالة التي تريد استعمالها، وهنا أردنا طباعة نص "Hello, World!"، لذا لاحظ أننا مررنا القيمة بين علامتَي اقتباس " " لأنها سلسلة نصية string.

احفظ الملف البرمجي الذي أنشأته واخرج من المحرر عن طريق كتابة Ctrl+X وتأكيد الخروج بالضغط على المفتاح Y عندما يطلب منك، ويمكنك الآن تجربة برنامجك.

2.1.3 الخطوة 2: تشغيل البرنامج

يمكنك تشغيل أيّ برنامج مكتوب بلغة جو من خلال كتابة الكلمة المفتاحية `go` متبوعة باسم الملف، وبالتالي لتشغيل البرنامج السابقة سنكتب:

```
$ go run hello.go
```

الخرج:

```
Hello, World!
```

بدايةً يُصرِّف البرنامج قبل تشغيله، أي إنشاء برنامج قابل للتنفيذ يمكن للحاسوب فهمه وتنفيذه، إذ يحوَّل إلى ملف ثنائي، وبالتالي عند استدعاء `go run` سيُصرَّف الملف البرمجي ثم يُشغل الملف التنفيذي الناتج.

غياب حزمة `main` من برنامج جو يجعل منه مكتبة فقط وليس برنامجًا بحد ذاته (حزمة تنفيذية)، لذا تتطلب برامج جو وجود هذه الحزمة، كما تتطلب وجود دالة `main()` واحدة تعمل على أساس نقطة دخول للبرنامج بحيث لا تأخذ هذه الدالة أيّ وسائط ولا تُعيد أيّ قيم.

تُنفَّذ الشيفرة أو البرنامج بعد انتهاء عملية التصريف من خلال وضع الدالة `main()` في الحزمة `main`، والتي تتضمن طباعة السلسلة النصية "Hello, World!" عن طريق استدعاء دالة `fmt.Println` أي `fmt.Println("Hello, World!")`، وتُدعى السلسلة النصية وسيطًا لتلك الدالة بما أنها تُمرَّر إليها.

> لا تُطبَع علامتَا الاقتباس الموجودتان على جانبي Hello, World! على الشاشة لأنك تستخدمهما لإخبار جو من أين تبدأ السلسلة الخاصة بك وأين تنتهي.

ستستكشف في الخطوة التالية كيفية جعل البرنامج تفاعليًّا أكثر.

2.1.4 الخطوة 3: إدخال معلومات من المستخدم لاستخدامها في البرنامج

سيطبع البرنامج السابق الخرج نفسه في كل مرة تستدعيه بها، كما يمكنك إضافة بعض التعليمات إلى البرنامج بحيث تجعله يطلب من المستخدِم إدخال اسمه لكي يُطبع بجانب عبارة الترحيب.

بإمكانك تعديل برنامجك السابق، ولكن يُفضَّل أن تُنشئ برنامجًا جديدًا يسمى greeting.go بالمحرِّر نانو:

```
nano greeting.go
```

ثم تضيف له التعليمات التالية:

```
package main
import (
  "fmt"
)
func main() {
 fmt.Println("Please enter your name.")
}
```

استخدامنا دالة `fmt.Println` لطباعة النص على الشاشة كما في المرة السابقة.

أضف الآن السطر `var name string` لتخزين مُدخلات المستخدِم:

```
package main
import (
  "fmt"
)
func main() {
 fmt.Println("Please enter your name.")
 var name string
}
```

سيُنشئ السطر السابق متغيرًا جديدًا اسمه `name` باستخدام الكلمة المفتاحية `var`، وحددنا نوع هذا المتغير على أنه `string`، أي سلسلة نصية.

أضف الآن السطر `fmt.Scanln(&name)` إلى الشيفرة:

```
package main
import (
  "fmt"
)
func main() {
 fmt.Println("Please enter your name.")
 var name string
 fmt.Scanln(&name)
}
```

يخبر التابع `fmt.Scanln` الحاسوب بأن ينتظر إدخالًا من لوحة المفاتيح ينتهي بسطر جديد (الضغط على Enter) أو المحرف `\n`، إذ توقف عملية الإدخال هذه البرنامج مؤقتًا إلى حين انتهاء المستخدِم من إدخال أيّ نص يريده، ثم يستمر البرنامج عندما يضغط المستخدم على مفتاح الإدخال ENTER من لوحة المفاتيح، إذ تُلتقط بعد ذلك كافة ضغطات المفاتيح بما في ذلك ضغطات المفتاح ENTER وتُحوّل إلى سلسلة من المحارف.

بما أن الهدف هو استخدام النص (اسمه) الذي يدخله المستخدِم لوضعه بجانب العبارة الترحيبية التي تظهر في خرج البرنامج، فسينبغي عليك حفظ هذه المحارف عن طريق تخزينها في متغير من النوع `string` أي في المتغير `name`، إذ يُخزّن جو هذه السلسلة في ذاكرة الحاسب إلى حين انتهاء تشغيل البرنامج.

أخيرًا، أضف السطر `fmt.Printf("Hi, %s! I'm Go!", name)` إلى برنامجك لطباعة الخرج:

```
package main
import (
  "fmt"
)
func main() {
 fmt.Println("Please enter your name.")
 var name string
 fmt.Scanln(&name)
 fmt.Printf("Hi, %s! I'm Go!", name)
}
```

استخدمنا هذه المرة الدالة `fmt.Printf` في عملية الطباعة لأنها ستسمح لنا بتنسيق عملية الطباعة بالطريقة التي نريدها، إذ تأخذ هذه الدالة وسيطَين الأول هو سلسلة نصية تتضمن العنصر النائب `%s` (العنصر النائب Placeholder هو متغير يأخذ قيمة في وقتٍ لاحق أي ينوب عن القيمة الحقيقة في البداية)، والثاني هو المُتغير الذي سيُحقن في هذا العنصر النائب، وهذه الدالة تُسمى دالة مرنة، أي تأخذ عددًا غير مُحدد من المعطيات، كما أن المُعطى الأول قد يتضمن أكثر من عنصر نائب لكن هذا لا يهمنا الآن، ونفعل ذلك لأن لغة جو لا تدعم تضمين المتغيرات ضمن السلاسل النصية مباشرة كما تفعل لغات أخرى مثل لغة جافا سكريبت.

احفظ الملف البرمجي الذي أنشأته واخرج من المحرِّر عن طريق الضغط على Ctrl+X وتأكيد الخروج بالضغط على المفتاح Y عندما يطلب منك.

شغّل البرنامج الآن وستُطالب بإدخال اسمك، لذا أدخله واضغط على ENTER، وقد لا يكون الخرج هو بالضبط ما تتوقعه:

```
Please enter your name.
Sammy
Hi, Sammy
! I'm Go!
```

بدلًا من طباعة Hi, Sammy! I'm Go!، هناك سطر فاصل بعد الاسم مباشرةً، فقد التقط البرنامج جميع ضغطات المفاتيح بما في ذلك المفتاح ENTER الذي ضغطنا عليه لإخبار البرنامج بالمتابعة، لذا تكمن المشكلة في السلسلة، إذ يؤدي الضغط على المفتاح الإدخال ENTER إلى إنشاء رمز خاص يُنشئ لنا سطرًا جديدًا وهو الرمز `\n`.

افتح الملف لإصلاح المشكلة:

```
$ nano greeting.go
```

أضف السطر التالي:

```
...
fmt.Scanln(&name)
...
```

ثم السطر التالي:

```
name = strings.TrimSpace(name)
```

استخدمنا هنا الدالة `TrimSpace` من الحزمة `strings` في مكتبة جو القياسية على السلسلة التي التقطتها باستخدام `fmt.Scanln`، إذ تزيل الدالة `strings.TrimSpace` محارف المسافة space characters بما في ذلك الأسطر الجديدة من بداية السلسلة ونهايتها، إذًا ستُزيل محرف السطر الجديد في نهاية السلسلة التي أُنشِئت عند الضغط على ENTER.

ستحتاج من أجل استخدام الحزمة `strings` إلى استيرادها في الجزء العلوي من البرنامج كما يلي:

```
import (
  "fmt"
  "strings"
)
```

سيكون البرنامج الآن كما يلي:

```
package main
import (
  "fmt"
  "strings"
)
func main() {
  fmt.Println("Please enter your name.")
  var name string
  fmt.Scanln(&name)
  name = strings.TrimSpace(name)
  fmt.Printf("Hi, %s! I'm Go!", name)
}
```

احفظ الملف البرمجي الذي أنشأته، واخرج من المحرِّر عن طريق الضغط على الاختصار Ctrl+X وتأكيد الخروج بالضغط على الزر Y عندما يطلب منك.

شغّل البرنامج:

```
$ go run greeting.go
```

ستحصل على الناتج المتوقع هذه المرة بعد إدخال اسمك والضغط على مفتاح ENTER:

```
Please enter your name.
Sammy
Hi, Sammy! I'm Go!
```

أصبح لديك الآن برنامج جو يأخذ مدخلات من المستخدِم ويطبعها على الشاشة.

تعلمت في هذه الفقرة كيفية كتابة برنامج بسيط يطبع رسالة ترحيبية، كما تعلمت كيفية إنشاء برنامج يأخذ مُدخلات من المستخدِم ويعالجها ويطبع رسالةً تفاعليةً، كما يمكنك الآن التلاعب بهذا البرنامج وتعديل بعض الأمور بداخله مثل أن تجعله يطلب اسم اللون المُفضّل للمستخدِم أو اسم فريقه المُفضل ويطبع تلك المعلومات على الشاشة.

2.2 التعرف على GOPATH

سنوضح لك في الفقرات التالية ماذا يعني GOPATH وكيف يعمل وكيفية إعداده لأنها خطوة هامة عند إعداد بيئة التطوير، بالإضافة إلى فهم كيفية عثور جو على الملفات المصدرية وتثبيتها وإنشائها. وفي هذا الكتاب سوف نستخدم GOPATH عند الإشارة إلى مفهوم بنية المجلد التي سنناقشها، كما سنستخدِم $GOPATH للإشارة إلى متغير البيئة الذي سيستخدِمه جو للعثور على بنية المجلد.

تُعَدّ مساحة عمل جو Go Workspace الطريقة التي يدير بها جو ملفات المصدر والملفات الثنائية والكائنات المخزنة مؤقتًا التي ستُستخدَم في وقت لاحق لعمليات التصريف السريع، وعادةً ما تكون لديك مساحة عمل واحدة فقط، إذ من الممكن أن تكون لديك أكثر لكن يُستحسن واحدة فقط، كما يكون GOPATH هو المجلد الجذر لها.

انتبه إلى أنه بدءًا من الإصدار 1.13 أضاف مطورو لغة Go آلية جديدة لإدارة المكتبات التي يعتمد عليها مشروع مكتوب بلغة جو تدعى وحدات جو Go Modules -سنتحدث عنها لاحقًا- وأصبحت هذه الآلية الطريقة المعتمد حاليًا في إضافة اعتماديات المشروع وإدارة هيكلة مجلده الرئيسي، وصحيح أن هذه الآلية قد حلت مكان GOPATH إلا أنه من الجيد فهم الآلية القديمة هذه التي قد تكون مستعملة في مشاريع قديمة أو تمر معك في شيفرة ما، ولمزيد من التفاصيل ومعرفة أحدث المستجدات عد إلى توثيق workspaces الرسمي من لغة جو.

2.2.1 ضبط متغير البيئة $GOPATH

يحمل متغير البيئة `$GOPATH` قائمةً تتضمن أماكن ليبحث فيها جو عن مساحات العمل.

يتوقع جو في الحالة الافتراضية أنّ موقع `GOPATH` الخاص بك هو `$HOME/go`، حيث `$HOME` هو المجلد الجذر لحساب المستخدِم الخاص بك على الحاسب، ويمكنك تغيير ذلك من خلال تعديل متغير البيئة `$GOPATH` حسب توثيق جو.

2.2.2 الفرق بين $GOPATH و $GOROOT

يُعَدّ `$GOROOT` المكان الذي تتواجد فيه شيفرة جو والمصرّف compiler والأدوات الأخرى التابعة له، أي أنه لا يتضمن الشيفرة المصدرية الخاصة بك، وعادةً ما يكون مسار `$GOROOT` هو `/usr/local/go` ومسار `$GOPATH` هو `$HOME/go`، كما تجدر الإشارة إلى أنك لست بحاجة إلى إعداد المتغير `$GOROOT` بعد الآن، إذ كان ذلك مطلبًا في السابق.

ا. مكونات مساحة العمل

توجد ثلاثة مجلدات داخل مساحة عمل جو أو المجلد `GOPATH` وهي `bin` و `pkg` و `src`، كما يملك كل مجلد من هذه المجلدات وظيفةً خاصةً في جو.

```
.
├── bin
├── pkg
└── src
    └── github.com/foo/bar
        └── bar.go
```

يمثّل المجلد `$GOPATH/bin` المكان الذي توضع فيه الملفات الثنائية (التنفيذيّة) التي تم تثبيتها بواسطة الأمر `go install`.

يستخدِم نظام التشغيل متغير البيئة `$PATH` للعثور على الملفات الثنائية التي يمكن تنفيذها بدون كتابة المسار الكامل، لذا يوصى بإضافة المجلد `bin` إلى المتغير `$PATH` العام، فإذا لم تضف `$GOPATH/bin` مثلًا إلى متغير البيئة `$PATH`، فسنحتاج إلى كتابة ما يلي:

```
$ $GOPATH/bin/myapp
```

أما إذا كان مُضافًا إليه لكان بالإمكان تشغيله بكتابة ما يلي:

```
$ myapp
```

توضع في المجلد `$GOPATH/pkg` ملفات الكائنات المُصرّفة مسبقًا والتي تُستخدَم لتسريع عملية التصريف في وقت لاحق، ومن الجدير بالذكر أنّ معظم المطورين لن يحتاجوا إلى التعامل مع هذا المجلد، فإذا كنت تواجه مشكلات في التصريف، فيمكنك حذف هذا المجلد بأمان وسيُعيد جو بناءه.

توضع في المجلد `src` شيفرة جو المصدرية وهي الملفات التي يتم كتابتها وإنشاؤها باستخدام لغة جو والتي تكون لاحقتها `.go`، الملفات المصدرية يستخدمها مُصرّف جو لإنشاء ملفات قابلة للتنفيذ (ملفات ثنائية يمكن تشغيلها على نظامك لتنفيذ المهام التي تتضمنها). وانتبه جيدًا مرةً أخرى إلى أنّ هذه الملفات ليست الملفات الموجودة في `$GOROOT`، كما ستوضع هذه الملفات في المسار `$GOPATH/src/path/to/code` أثناء كتابة برامج وحزم ومكتبات جو.

ب. ما هي الحزم؟

تُعَدّ الحزم طريقةً بسيطةً لتجزئة برنامجك إلى أجزاء أصغر مُقسّمة حسب الغرض أو الوظيفة، كما يُمكن الإشارة للحزم على أنها مكتبات libraries أو وحدات modules، تركيب الحزم مع بعضها يشكل مُجمل برنامجك. الشيفرات التي تُكتب في جو تُنظّم ضمن حزم، إذ تمثل الحزمة جميع الملفات الموجودة في مجلد واحد على القرص.

تتبع جو القواعد التالية في كيفية تقسيم برنامج إلى حزم:

- تُخزّن الحزم مع جميع ملفات جو المصدرية المكتوبة من قبل المستخدِم ضمن المجلد `$GOPATH/src`.
- الحزمة غالبًا عبارة عن مجلد داخل مشروعك به الملفات التي تحمل اسم الحزمة.
- لا بُدَّ من وجود حزمة واحدة على الأقل في أي برنامج أو مكتبة.
- إذا كان البرنامج عبارة عن برنامج تنفيذي (وليس مكتبة - library) فإنه يجب وجود حزمة باسم `main` (أي `package main`) تكون هي مدخل البرنامج كما رأينا سابقًا.

وكتذكير لما ذكرناه سابقًا:

- لا بُدَّ لكل ملف شفرة برمجية في Go أن ينتمي إلى حزمة (package) ما.
- لا بُدَّ لكل حزمة في Go أن تنتمي إلى مجلد ما.
- لا يمكن لحزمتين التواجد على مستوى نفس المجلد، لكن يمكن لعدة ملفات أن تنتمي إلى نفس الحزمة (يعني مجلد به عدة ملفات).
- يمكن أن تتواجد حزمة داخل حزمة أخرى لكن كلٌ في مجلد فرعي على حِدة.

- يمكن للمجلد الرئيسي لمشروعك أن يحتوي على حزمة ما، واحدة فقط، باقي الحزم الخاصة به يمكنها أن تتواجد في مجلدات فرعية.
- غياب حزمة `main` من برنامج ما، يجعل منه مكتبة فقط وليس برنامجا تشغيليا قائما بحد ذاته.

إذا كانت الشيفرة الخاصة بك موجودةً في المسار `$GOPATH/src/blue/red` فيجب أن يكون اسم الحزمة التي تنتمي إليها الشيفرة هو red، ويمكنك استيراد هذه الحزمة كما يلي:

```
import "blue/red"
```

في حالة الحزم المتواجدة ضمن مستودعات على مواقع استضافة الشيفرات مثل GitHub و BitBucket، فيجب تضمين المسار الكامل في عملية الاستيراد، فإذا أردنا استيراد الشيفرة المصدرية من الرابط https://github.com/gobuffalo/buffalo، فسنكتب ما يلي:

```
import "github.com/gobuffalo/buffalo"
```

وستُخزَّن في الموقع التالي على القرص: `$GOPATH/src/github.com/gobuffalo/buffalo`

بهذا تكون قد تعرّفت على المجلد `GOPATH` ومحتوياته، وطريقة تغيير الموقع الافتراضي `$HOME/go` له، وتعرّفت على الآلية التي يبحث بها جو عن الحزم داخل بنية المجلد.

وتذكر أن Go Modules قُدّمت في الإصدار 1.11 من جو، وهي تهدف إلى استبدال مساحات عمل جو و `GOPATH`، وعلى الرغم من أنه يُوصى ببدء استخدامها إلا أنّ بعض البيئات قد لا تكون جاهزةً لاستخدامها مثل بيئات الشركة corporate environments.

قد يُعَدّ `GOPATH` أكثر الجوانب تعقيدًا في إعداد جو ولكن بمجرد إعداده يمكنك نسيانه بعدها على الغالب.

2.3 كتابة التعليقات في لغة جو Go

التعليقات هي عبارات دخيلة على الشيفرات البرمجية وليست جزءًا منها، إذ يتجاهلها المُصرّف compiler والمُفسّر interpreter، كما يُسهِّل تضمين التعليقات في الشيفرات من قراءتها وفهمها ومعرفة وظيفة كل جزء من أجزائها، لأنها توفر معلومات وشروحات حول ما يفعله كل جزء من البرنامج.

يمكن أن تكون التعليقات بمثابة مُذكِّرات لك بناءً على الغرض من البرنامج، أو يمكنك كتابتها لمساعدة المبرمجين الآخرين على فهم الشيفرة، ويُستحسَن كتابة التعليقات أثناء كتابة البرامج أو تحديثها لأنّك قد تنسى السياق وتسلسل الأفكار لاحقًا، و قد تكون التعليقات المكتوبة لاحقًا أقل فائدةً على المدى الطويل.

2.3.1 صياغة التعليقات

تبدأ التعليقات بالمحرفَين // وتنتهي بنهاية السطر الحالي، وغالبًا ما نضع فراغًا بعد هذين المحرفين لجعل التعليق مُرتبًا أكثر مثل:

```
// This is a comment
// هذا تعليق على الشيفرة
```

يتم تجاهل التعليقات كما ذكرنا أي وكأنها غير موجودة، وبالتالي عند تشغيل البرنامج لن يطرأ أي حدث يُشير إلى وجودها، فالتعليقات ضمن التعليمات البرمجية مُخصصة ليقرأها البشر وليس للحاسب، ففي برنامج "Hello، World!" مثلًا يمكنك إضافة التعليقات كما يلي:

```
package main
import (
  "fmt"
)
func main() {
  // اطبع العبارة "Hello, World!" إلى الطرفية
  fmt.Println("Hello, World!")
}
```

وإليك مثالًا آخر لوضع التعليقات ضمن حلقة for التي تُكرِّر تنفيذ كتلة برمجية أو مجموعة من التعليمات البرمجية:

```
package main
import (
  "fmt"
)
func main() {
  // عرف المتغير التالي على أنه شريحة من سلاسل نصية
  sharks := []string{"hammerhead", "great white", "dogfish",
"frilled", "bullhead", "requiem"}

  // حلقة تكرار للمرور على كل عنصر من عناصر القائمة وطباعته
  for _, shark := range sharks {
   fmt.Println(shark)
  }
}
```

نستنتج من المثالين السابقين أن التعليقات يجب أن تكون على سويّة التعليمات التي توضحها، أي بالمسافة البادئة نفسها للشيفرة التي تُعلّق عليها، وبالتالي يجب أن يكون تعليق بدون مسافة بادئة تسبُقه من أجل تعريف دالة بدون مسافة بادئة، وكل مقطع برمجي مسبوق بمسافة بادئة سيكون له تعليقات تتماشى مع هذه المسافة.

لاحظ على سبيل المثال كيف سنُعلّق على الدالة `main`، ولاحظ كيف سنضع التعليقات بما يتماشى مع كل مقطع مسبوق بمسافة بادئة:

```
package main
import "fmt"
const favColor string = "blue"

func main() {
  var guess string

  // إنشاء حلقة تكرار
  for {
   // اطلب من المستخدم تخمين لوني المفضل
   fmt.Println("Guess my favorite color:")
   // اقرأ ما أدخله المستخدم واطبع النتيجة
   if _, err := fmt.Scanln(&guess); err != nil {
   fmt.Printf("%s\n", err)
   return
   }
   // تأكد إن خمن اللون الصحيح
   if favColor == guess {
   // اللون الذي أدخله صحيح
   fmt.Printf("%q is my favorite color!\n", favColor)
   return
   }
   // اللون الذي أدخله خطأ ونطلب منه تكرار الإجابة
   fmt.Printf("Sorry, %q is not my favorite color. Guess again.\n",
guess)
  }
}
```

تهدف التعليقات كما ذكرنا إلى مساعدة المبرمجين سواءً كان المبرمج الذي كتب الشيفرة أو أي شخص آخر يستخدِم المشروع أو يساعد فيه، كما يجب أن تُحفظ التعليقات وتحدّث بطريقة صحيحة بما يتماشى مع مستوى التعليمات البرمجية ومحتواها وإلا فمن الأفضل عدم كتابته.

يجب أن يكون التعليق على شيفرة ما هو توضيح لمضمونها أو للفكرة التي أنتجت هذه الشيفرة وليس كتابة أسئلة أو شيء آخر، أي يجب أن تُجيب التعليقات عن السبب الذي أنتج الشيفرة.

2.3.2 التعليقات الكتلية

يمكن استخدام التعليقات الكتليّة Block Comments -أي عدة أسطر من التعليقات- لتوضيح الشيفرات البرمجية المعقدة التي تتوقع بأن لا يكون القارئ على دراية بها، إذ تنطبق هذه التعليقات الطويلة على جزء من الشيفرة أو جميعها، كما توضع في نفس مستوى المسافة البادئة للشيفرة، كما يمكن كتابة التعليقات الكتلية بطريقتين؛ الأولى من خلال المحرفَين السابقَين `//` وتكرارهما من أجل كل سطر مثل:

```
// First line of a block comment
// Second line of a block comment
```

أما الطريقة الثانية، فمن خلال علامة الفتح `/*` التي نضعها في بداية التعليق وعلامة الإغلاق `*/` التي نضعها في نهاية التعليق، إذ يُستخدَم `//` عادةً مع عمليات التوثيق؛ أما `/* ... */` فيستخدم من أجل تصحيح الأخطاء debugging.

```
/*
تعليق طويل
يمتد على
أكثر من سطر
*/
```

سنستخدم التعليقات الكتليّة في المثال التالي لتوضيح الدالة `MustGet()`:

```
// تأخذ الدالة MustGet رابطًا وتعيد الصفحة الرئيسية له
// وإن حصل أي خطأ ستظهره
func MustGet(url string) string {
  resp, err := http.Get(url)
  if err != nil {
   panic(err)
  }
  defer resp.Body.Close()
  var body []byte
```

```
    if body, err = ioutil.ReadAll(resp.Body); err != nil {
     panic(err)
    }
    return string(body)
}
```

من الشائع رؤية التعليقات الكتلية في بداية الدوال التي تُقدّمها جو، وهذه التعليقات هي التي تُنشئ أيضًا توثيق الشيفرة البرمجية الخاصة بك، كما تُستخدَم التعليقات الكتلية أيضًا عندما تتطلب الشيفرة شرحًا كاملًا.

> تجنب قدر الإمكان استخدام التعليقات الكتليّة باستثناء الأهداف التوثيقية، وثِق بقدرة المبرمجين الآخرين في فهم التعليقات المختصرة إلا إذا كان لديك هدف من كتابة هذه التعليقات الطويلة لأغراض تعليمية مثلًا.

2.3.3 التعليقات السطرية

توضَع التعليقات السطرية Inline comments على السطر نفسه الذي توجد فيه التعليمة البرمجية، وهي مثل التعليقات الأخرى أي تبدأ بالمحرفَين // ومسافة بيضاء واحدة، إذ تبدو التعليقات الضمنيّة عمومًا كما يلي:

```
[code] // تعليق سطري يشرح سطر الشيفرة
```

لا ينبغي الإكثار من استخدام التعليقات السطرية، ولكن يمكن أن تكون فعالة لشرح الأجزاء الصعبة من الشيفرة عند استخدامها في محلها، وقد تكون مفيدةً أيضًا إذا ظننت أنَّك قد لا تتذكر سطرًا من الشيفرة في المستقبل أو إذا كنت تتعاون مع شخص قد لا يكون على دراية بجميع جوانب الشيفرة.

فإذا كنت لا تستخدم الكثير من الحسابات في برامجك في جو Go ولم يكن هناك توضيح مسبق على سبيل المثال، فقد لا تعلم أنت أو المتعاونون معك أنّ الشيفرة التالية ستنشئ عددًا عقديًا، لذلك قد ترغب في إضافة تعليق سطري كما يلي:

```
z := x % 2 // احصل على باقي قسمة العدد x
```

يمكن أيضًا استخدام التعليقات السطرية لشرح السبب وراء فعل شيء ما أو لتعطي بعض المعلومات الإضافية كما في المثال التالي:

```
x := 8 // إسناد قيمة للعدد
```

يجب استخدام التعليقات السطرية عند الضرورة وعندما توفِّر إرشادات مفيدةً للشخص الذي يقرأ البرنامج.

2.3.4 تعليق جزء من الشيفرة بدواعي الاختبار والتنقيح

يمكن أيضًا استخدام المحرفَين // أو رمزي الفتح والإغلاق /*...*/ لتعليق جزء من الشيفرة وتعطيله أثناء اختبار أو تنقيح البرنامج الذي تعمل عليه، بالإضافة إلى استخدام التعليقات على أساس وسيلة لتوثيق

الشيفرة، أي عندما تواجه أخطاء بعد إضافة أسطر جديدة إلى الشيفرة، فسترغب في تعليق بعضها لمعرفة موضع الخلل.

يمكن أن يتيح لك استخدام الرمزين `/*...*/` تجربة بدائل أخرى أثناء إعداد الشيفرة أو للتعليق على التعليمات البرمجية التي تفشل أثناء استمرار العمل على أجزاء أخرى من التعليمات البرمجية الخاصة بك.

```
// دالة تضيف عددين
func addTwoNumbers(x, y int) int {
  sum := x + y
  return sum
}

// دالة تضرب عددين
func multiplyTwoNumbers(x, y int) int {
  product := x * y
  return product
}

func main() {
  /*
   addTwoNumbers علقنا في هذا الموضع استدعاء الدالة
   لأنه يولد خطأ لم نكتشفه فلا نريد إيقاف الشيفرة
   a := addTwoNumbers(3, 5)
   fmt.Println(a)
  */
  m := multiplyTwoNumbers(5, 9)
  fmt.Println(m)
}
```

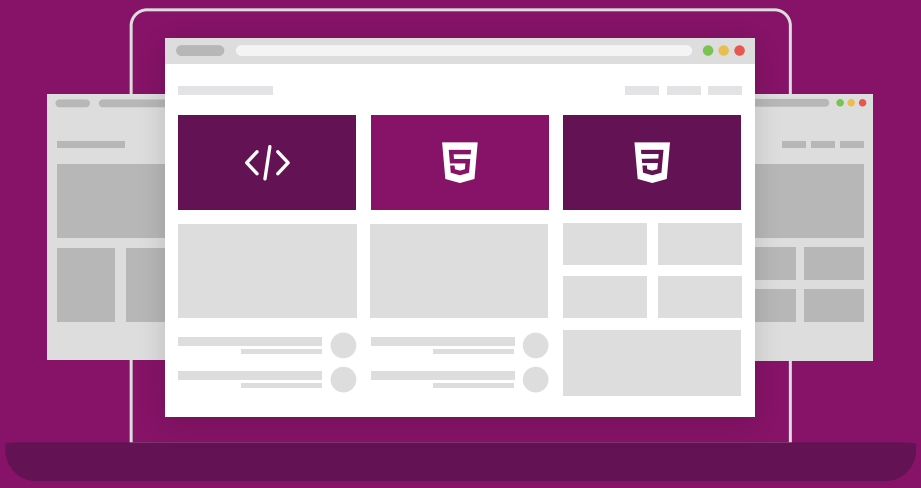
يتيح لك تعليق الشيفرة البرمجية تجربة عدة طرق ومقاربات برمجية، بالإضافة إلى مساعدتك على العثور على مصدر الخطأ من خلال التعليق المنهجي لبعض أجزاء البرنامج وتشغيلها، وتذكر أنك تستخدمه لأغراض الاختبار والتنقيح.

يمكنك بعد التعليق على الشيفرة في جو بشكل صحيح، استخدام الأداة Godoc، وهي أداة تستخرج التعليقات من التعليمات البرمجية الخاصة بك وتُنشئ وثائق لبرنامج جو الخاص بك.

2.4 الخاتمة

بهذا تكون قد وصلت لنهاية الفصل الثاني من كتاب تعلم البرمجة بلغة Go والذي تعرفت فيـه على طريقـة كتابة برنامجك الأول بلغة جو والذي يطبع عبارة "Hello, World!" على شاشة الخرج، وتعرفت كذلك على مفهوم المجلد GOPATH الذي يحتوي على مجموعة من المجلدات التي يتوقع جو وجود الشيفرة المصدرية الخاصة بك بداخلها، كما تعرفت في نهاية الفصل على مفهوم التعليقات وماهي حالات استخدامها المختلفة في بـرامج جـو ودورها في جعل برامجك أكثر قابلية للقراءة سواءً لك أو لغيرك.

دورة تطوير واجهات المستخدم



ابدأ مسارك المهني كمطور واجهات المواقع والمتاجر الإلكترونية
فور انتهائك من الدورة

التحق بالدورة الآن

3. تعرف على أنواع البيانات

تُعَدّ المتغيرات مفهومًا برمجيًا مهمًّا يشير إلى القيم ونوع القيم التي تستخدمها في برنامجك، إذ يحدد نوع المتغير -أو نوع البيانات- نوع القيم التي يمكن تخزينها فيه، وما هي العمليات التي يمكن إجراؤها عليه. من الناحية الفنية، يُخصَّص للمتغير مساحة تخزينية في الذاكرة توضع القيمة المرتبطة به فيها، ويُستخدم اسم المتغير للإشارة إلى تلك القيمة المُخزَّنة في ذاكرة البرنامج التي هي جزء من ذاكرة الحاسوب.

يغطي هذا الفصل أنواع المتغيرات المهمة والأساسية التي تحتاجها مبدأيًا، ويشرح المتغيرات في جو لتكون قادرًا على فهمها والتعامل معها، وتمتلك القدرة على كتابة تعليمات برمجية أكثر وضوحًا وتعمل بكفاءة.

يمكن التفكير في أنواع المتغيرات كما في العالم الحقيقي، فنحن نتعامل مثلًا مع الأعداد من مجموعات عددية مختلفة مثل مجموعة الأعداد الطبيعية (0 ، 1 ، 2 ، ...) ومجموعة الأعداد الصحيحة (... ، -1 ، 0 ، 1 ، ...) والأعداد غير الكسرية مثل π.

عند التعامل مع المسائل الرياضية قد نجمع بين الأعداد التي تنتمي إلى مجموعات عددية مختلفة لنحصل على نتيجة ما مثلًا إضافة العدد 5 إلى π:

```
+ π
```

يمكننا الآن الاحتفاظ بالمعادلة كما هي لتكون إجابة، أو يمكننا تقريب العدد π إلى منزلة عشرية مناسبة ثم جمعه مع العدد 5، كما يلي:

```
+ π = 5 + 3.14 = 8.14
```

لكن إذا حاولت جمع أيّ عدد مع نوع بيانات آخر وليكن كلمة مثلًا، فسيكون هذا بدون معنى مثل:

```
shark + 8
```

هناك فصل بين أنواع البيانات في الحاسوب، إذ يختلف كل نوع عن الآخر بحيث يوجد نوع مخصص للنصوص والكلمات ونوع آخر للأعداد، ويجب الانتباه عند استخدام أو تطبيق العمليات على هذه المتغيرات.

3.1 الأعداد الصحيحة

كما هو الحال في الرياضيات، فإن الأعداد الصحيحة Integers في الحاسوب هي أعداد موجبة أو سالبة إضافةً إلى الصفر، أي (... ، 1 ، 0 ، 1- ، ...).

يُعبَّر عن الأعداد الصحيحة في لغة جو من خلال النوع `int`، كما يجب أن تكتب العدد الصحيح كما هو بدون فواصل بين الأعداد كما في اللغات البرمجية الأخرى، فعادةً تُفصَل الأعداد الكبيرة بفواصل لتسهيل قراءتها، فقد يُكتب المليون مثلًا بالصورة 1,000,000 وفي لغات البرمجة هذا غير مسموح، ويمكن طباعة العدد الصحيح ببساطة كما يلي:

```
fmt.Println(-459)
```

والنتيجة هي:

```
-459
```

أو يمكنك تعريف متغير من النوع الصحيح ليمثل هذا العدد كما يلي:

```
var absoluteZero int = -459
fmt.Println(absoluteZero)
```

والنتيجة هي:

```
-459
```

يمكنك أيضًا إجراء العمليات الحسابية على الأعداد الصحيحة في جو، ففي الشيفرة التالية سنستخدِم معامل الإسناد `:=` لتعريف وتقييم المتغير `sum`:

```
sum := 116 - 68
fmt.Println(sum)
```

والنتيجة هي:

```
48
```

نلاحظ أن العدد 68 طُرح من العدد 116 ثم خُزّن الناتج 48 في المتغيّر `sum`.

ستتعرف على المزيد من التفاصيل المتعلقة بالتصريح عن المتغيرات لاحقًا، وهناك العديد من الطرق لاستخدام الأعداد الصحيحة في جو ستتعرف على معظمها خلال رحلتك التعليمية هذه.

3.2 الأعداد العشرية

يُعَدّ العدد ذو الفاصلة العشرية Floating-Point عددًا عشريًا ويمكن كتابته على صورة حاصل ضرب في على النحو التالي: 1012.5 = 101.25^2 = 125.

وتُستخدَم الأعداد العشرية لتمثيل الأعداد الحقيقية التي لا يمكن التعبير عنها بالأعداد الصحيحة، إذ تتضمن الأعداد الحقيقية جميع الأعداد الكسرية وغير الكسرية، ولهذا السبب يمكن أن تحتوي أرقام عشرية على جزء كسري مثل 9.0 أو 116.42- في جو، فالعدد العشري في جو هو أيّ عدد يحتوي على فاصلة عشرية ببساطة:

```
fmt.Println(-459.67)
```

والنتيجة هي:

```
-459.67
```

يمكنك أيضًا تعريف متغير يمثّله كما يلي:

```
absoluteZero := -459.67
fmt.Println(absoluteZero)
```

والنتيجة هي:

```
-459.67
```

يمكنك أيضًا إجراء العمليات الحسابية عليه كما فعلنا مع الأعداد الصحيحة:

```
var sum = 564.0 + 365.24
fmt.Println(sum)
```

والنتيجة هي:

```
929.24
```

يجب أن تأخذ في الحسبان أنّ العدد 3 لا يساوي العدد 3.0، إذ يشير العدد 3 إلى عدد صحيح في حين يشير العدد 3.0 إلى عدد عشري.

3.3 حجم الأنواع العددية

يحتوي جو على نوعين من البيانات العددية بالإضافة إلى التمييز بين الأعداد الصحيحة والعشرية، وهما الساكنة static والديناميكية dynamic، فالنوع الأول مستقل عن المعمارية architecture-independent، أي أنّ حجم البيانات بواحدة البِتّ bit لا يتغير بغض النظر عن الجهاز الذي تعمل عليه الشيفرة.

معظم معماريات أجهزة الحاسب في هذه الأيام هي إما 32 بِتّ أو 64 بِتّ، فإذا فرضنا مثلًا أنك تريد تطوير برنامج لحاسبك المحمول، فسترغب بجعله يعمل بنظام ويندوز حديث معماريته 64 بِتّ، لكن إذا كنت تعمل على تطوير برنامج لجهاز مثل ساعة اللياقة البدنية fitness watch، فربما ستُجبر على العمل بمعمارية 32 بِتّ.

إذا استخدمت نوع المتغيرات المستقلة مثل `int32`، فسيكون حجم التخزين للمتغير ثابتًا بغض النظر عن المعمارية التي تُجري عملية التصريف عليها، أي باختصار، سيجعل استخدامك للمتغيرات من النوع الأول هذا المتغير ذا حجم ثابت في أيّ معمارية أو بيئة تستخدمها.

يُعَدّ النوع الثاني نوعًا خاصًا بالتنفيذ implementation-specific، إذ يمكن هنا أن يختلف حجم البِتّ بناءً على المعمارية التي بُنيَ البرنامج عليها، فإذا استخدمت النوع `int`، فسيكون حجم المتغير هو 32 بِتّ عند تصريف برنامج جو لمعمارية 32 بِتّ؛ أما إذا صُرِّف البرنامج بمعمارية 64 بِتّ، فسيكون حجم المتغير 64 بِتّ، لذا يُسمى ديناميكيًا.

هناك أنواع لأنواع البيانات نفسها إضافةً إلى أنواع البيانات ذات الأحجام المختلفة، فالأعداد الصحيحة مثلًا لها نوعان أساسيان هما أعداد ذات إشارة signed وأخرى عديمة الإشارة unsigned، فالنوع `int8` هو عدد صحيح له إشارة سالبة أو موجبة ويمكن أن يكون له قيمة من 128- إلى 127. أما النوع `uint8` فهو عدد صحيح عديم الإشارة أي موجب دومًا، ويمكن أن يكون له فقط قيمة موجبة من 0 إلى 255.

يمتلك جو الأنواع التالية المستقلة عن نوع المعمارية architecture-independent للأعداد الصحيحة:

```
uint8   unsigned 8-bit integers (0 to 255)
uint16  unsigned 16-bit integers (0 to 65535)
uint32  unsigned 32-bit integers (0 to 4294967295)
uint64  unsigned 64-bit integers (0 to 18446744073709551615)
int8    signed 8-bit integers (-128 to 127)
int16   signed 16-bit integers (-32768 to 32767)
int32   signed 32-bit integers (-2147483648 to 2147483647)
int64   signed 64-bit integers (-9223372036854775808 to
9223372036854775807)
```

تأتي أعداد الفاصلة العشرية والأعداد المركبة أو العقدية complex numbers أيضًا بأحجام مختلفة:

```
float32 IEEE-754 32-bit floating-point numbers
float64 IEEE-754 64-bit floating-point numbers
complex64  complex numbers with float32 real and imaginary parts
complex128 complex numbers with float64 real and imaginary parts
```

يوجد أيضًا نوعان بديلان alias للأعداد بغية إعطائها اسمًا هادفًا:

```
byte    alias for uint8
rune    alias for int32
```

الهدف من الأسماء البديلة مثل البايت byte هو تحسين مقروئية الشيفرة وتوضيحها مثل حالة استعمال الاسم byte للإشارة إلى تخزين السلاسل النصية لكونه مقياسًا شائعًا في الحوسبة المتعلقة بالمحارف والنصوص، خلاف حالة استعمال الاسم uint8 الذي يشيع استعماله مع الأعداد رغم أن كلاهما يصيران إلى نوع واحد عند تصريف البرنامج، فبذلك مثلًا أينما رأيت التسمية byte في برنامج ستعرف أنك تتعامل مع بيانات نصية، وأينما رأيت الاسم uint8 ستعرف أنك تتعامل مع بيانات عددية.

الاسم البديل رون rune مختلف قليلًا، فعلى عكس البايت byte و uint8 اللذان يمثلان البيانات نفسها، يمكن أن يكون الرون بايتًا واحدًا أو أربعة بايتات، أي مجالًا من 8 إلى 32 بت وهو مايحدده النوع int32، كما يُستخدم الرون لتمثيل محرف Unicode (وهو معيار يُمكّن الحواسيب من تمثيل النصوص المكتوبة بأغلب نُظم الكتابة ومعالجتها بصورة متناسقة).

تحتوي لغة جو إضافةً إلى ذلك الأنواع التالية الخاصة بالتنفيذ:

```
uint    unsigned, either 32 or 64 bits
int     signed, either 32 or 64 bits
uintptr unsigned integer large enough to store the uninterpreted bits
of a pointer value
```

يُحدَّد حجم الأنواع الخاصة بالتنفيذ من خلال المعمارية التي صُرّف البرنامج من أجلها.

3.4 اختيار حجم الأنواع العددية في برنامجك

عادةً ما يكون اختيار الحجم الصحيح للنوع مرتبطًا بأداء المعمارية المستهدفة التي تبرمج عليها أكثر من ارتباطه بحجم البيانات التي تعمل عليها، وهناك إرشادات أساسية عامة يمكنك اتباعها عند بداية أي عملية تطوير.

تحدثنا سابقًا أنّ هناك أنواع بيانات مستقلة عن المعمارية، وأنواع خاصة بالتنفيذ، فبالنسبة لبيانات الأعداد الصحيحة من الشائع في جو استخدام أنواع تنفيذ مثل int أو uint بدلًا من int64 أو uint64، إذ ينتج عن ذلك عادةً سرعة معالجة أكبر على المعمارية المستهدفة، فإذا كنت تستخدم int64 مثلًا وأنجزت عملية

التصريف على معمارية 32 بِتّ، فسيستغرق الأمر ضعف الوقت على الأقل لمعالجة هذه القيم بحيث تستغرق دورات معالجة إضافية لنقل البيانات من معمارية لأخرى؛ أما إذا استخدمت `int` بدلًا من ذلك، فسيعرّفه البرنامج على أنه حجم 32 بِتّ لمعمارية 32 بِتّ وسيكون أسرع في المعالجة.

إذا كنت تعرف أنّ بياناتك لن تتجاوز مجالًا محددًا من القيم، فإن اختيار نوع مستقل عن المعمارية يمكن أن يزيد السرعة ويقلل من استخدام الذاكرة، فإذا كنت تعلم مثلًا أنّ بياناتك لن تتجاوز القيمة 100 وستكون موجبةً فقط، فسيجعل اختيار `uint8` برنامجك أكثر كفاءةً لأنه يتطلب ذاكرةً أقل.

بعد أن تعرّفت على بعض المجالات المحتملة لأنواع البيانات العددية، لنلقِ نظرةً على ما سيحدث إذا تجاوزت هذه النطاقات في البرنامج.

3.5 الفرق الطفحان والالتفاف Overflow vs Wraparound

تتعامل لغة جو مع حالة الطفحان overflow أو بلوغ الحد الأعظمي wraparound عند محاولة تخزين قيمة أكبر من نوع البيانات المصمم لتخزينه اعتمادًا على ما إذا كانت هذه القيمة محسوبة في وقت التصريف compile time أو في وقت التشغيل runtime، إذ يحدث خطأ في وقت التصريف عندما يعثر البرنامج على خطأ أثناء محاولته إنشاء البرنامج؛ أما الخطأ في وقت التشغيل، فيحدث بعد تصريف البرنامج أثناء تنفيذه بالفعل.

ضبطنا قيمة المتغير `maxUint32` في المثال التالي إلى أعلى قيمة ممكنة:

```
package main
import "fmt"
func main() {
  var maxUint32 uint32 = 4294967295 // Max uint32 size
  fmt.Println(maxUint32)
}
```

سيكون الخرج:

```
Output4294967295
```

الآن إذا أضفنا العدد `1` إلى المتغير `maxUint32` في وقت التشغيل، فسيحدث التفاف إلى القيمة `0`، وبالتالي سيكون الخرج:

```
0
```

سنعدّل الآن البرنامج ونضيف العدد `1` إلى المتغير `maxUint32` قبل وقت التصريف:

```
package main
import "fmt"
func main() {
  var maxUint32 uint32 = 4294967295 + 1
  fmt.Println(maxUint32)
}
```

إذا عثر المُصرّف compiler في وقت التصريف على قيمة كبيرة جدًا بحيث لا يمكن الاحتفاظ بها في نوع البيانات المحدد، فسيؤدي ذلك إلى حدوث خطأ طفحان overflow error، وهذا يعني أنّ القيمة المحسوبة كبيرة جدًا بالنسبة لنوع البيانات الذي حددته.

إذًا سيظهر الخطأ التالي في الشيفرة السابقة لأنّ المصرف قد عثر على قيمة كبيرة جدًا لا تُلائم مجال النوع المحدَّد:

```
prog.go:6:36: constant 4294967296 overflows uint32
```

فمعرفة أحجام بياناتك وما تتعامل معه يساعدك على تجنب أخطاء الطفحان تلك في برنامجك مستقبلًا لذا يجب الانتباه.

ستتعرّف الآن على كيفية تخزين القيم المنطقية أو البوليانية في لغة جو.

3.6 القيم المنطقية Boolean

يمتلك نوع البيانات المنطقية قيمتين؛ إما صح `true` أو خطأ `false`، وتعرَّف على أنها من النوع `bool` عند التصريح عنها، كما تُستخدَم القيم المنطقية لتمثيل قيم الحقيقة المرتبطة بالجبر المنطقي في الرياضيات والذي يُعد أساس الخوارزميات في علوم الحاسوب، كما يُعبَّر عن القيمتَين `true` و `false` في جو بمحرفين صغيرين دائمًا وهما `t` و `f` على التوالي.

تُعطينا العديد من العمليات الحسابية إجابات بوليانية تُقيَّم إما `true` أو `false` مثل:

- **أكبر من**: 500 > 100 إجابتها `true` أو 1 > 5 إجابتها `false`.
- **أقل من**: 200 > 400 إجابتها `true` أو 4 > 2 إجابتها `false`.
- **يساوي**: 5 = 5 إجابتها `true` أو 500 = 400 إجابتها `false`.

وبالتالي يمكنك تخزين قيمة منطقية في متغير بالصورة التالية:

```
myBool := 5 > 8
```

ثم يمكنك طباعتها من خلال الدالة `fmt.Println()` كما يلي:

```
fmt.Println(myBool)
```

بما أن العدد 5 ليس أكبر من 8، فسنحصل على الخرج التالي:

```
false
```

عندما تكتب المزيد من البرامج في جـو، ستصـبح أكـثر درايـةً بكيفيـة عمـل القيم المنطقيـة، وكيـف يمكن للوظائف والعمليات المختلفة التي تُقيَّم إلى true أو false أن تغيِّر مسار البرنامج.

3.7 السلاسل النصية Strings

تُعَدّ السلسلة النصية سلسلةً مكونة من محرف واحد أو أكثر (والمحـرف هـو حـرف أبجـدي أو عـدد أو رمـز)، ويمكن أن تكون سلسلة ثابتة أو متغيرة.

توجد صيغتان لتعريف السلاسل النصية في جو، فإذا اسـتخدمت علامتَي الاقتبـاس الخلفيـة ``` لتمثيـل السلسلة بداخلها، فهـذا يعـني أنـك ستُنشـئ سلسـلة أوليّـة raw string literal؛ أمـا إذا اسـتخدمت علامتَي الاقتباس المزدوجة "، فهذا يعني أنك ستُنشئ سلسلة مُفسّرة interpreted string literal.

3.7.1 السلسلة النصية الأولية

هي تسلسـل من المحـارف الموضـوعة بين علامتَي اقتبـاس خلفيـة ` والمسـمّاة أيضًـا علامتَي التجزئـة الخلفية back ticks.

سيُعرض كل شيء داخل هاتين العلامتين كما هو، وهذا لا يشـمل علامتَي الاقتبـاس الخلفيـة، فهمـا مجـرد دليل لنقطة بداية ونهاية السلسلة مثل:

```
a := `Say "hello" to Go!`
fmt.Println(a)
```

سيكون الخرج كما يلي:

```
Say "hello" to Go!
```

يُستخدَم رمز الشرطة المائلة للخلف \ (و نقصد بها الرمز \ وليس الرمز /) لتمثيل المحارف الخاصـة ضـمن السلسلة النصية، فإذا عُـثر في سلسـلة نصـيّة مُفسّـرة على الرمـز \n فهـذا يعـني الانتقـال لسـطر جديـد، ولكن الشرطة المائلة الخليفة ليس لها معنى ضمن السلاسل الأولية كما في المثال التالي:

```
a := `Say "hello" to Go!\n`
fmt.Println(a)
```

سيكون الخرج كما يلي:

```
Say "hello" to Go!\n
```

لاحظ أنّ السلسلة النصية السابقة هي سلسلة أوليّة، أي غير مُفسّـرة، وبالتـالي تُعـرَض كمـا هي ولن يكـون للرمز \n أيّ معنى خاص.

تُستخدَم السلاسل الأولية أيضًا لإنشاء سلاسل متعددة الأسطر:

```
a := `This string is on
multiple lines
within a single back
quote on either side.`
fmt.Println(a)
```

سيكون الخرج كما يلي:

```
This string is on
multiple lines
within a single back
quote on either side.
```

لاحظ هنا تأثير السلاسل الأولية؛ إذ ذكرنا سابقًا أنها تطبع السلسلة كما هي.

3.7.2 السلسلة النصية المفسرة

هي سلسلة نصية موضوعة بين علامتَي اقتبـاس مزدوجـتين " "، إذ سـيعرض كـل شـيء موجـود بـداخل علامتي الاقتباس باستثناء الأسطر الجديدة وعلامات الاقتباس المزدوجة نفسـها إلا مـا جـرى تهريبـه منهـا، فـإذا أردت إظهار هذه الرموز في هذه السلسلة، فيمكنك استخدام محرف الهروب \ قبلها مثل:

```
a := "Say \"hello\" to Go!"
fmt.Println(a)
```

سيكون الخرج كما يلي:

```
Say "hello" to Go!
```

ستستخدِم غالبًا السلاسل المُفسرة لأنها تتعامل مع المحارف الخاصة وتسمح بتخطيها أيضًا وبالتالي تملـك تحكمًا أكبر.

3.8 سلاسل صيغة التحويل الموحد UTF-8

تُعَدّ صيغة التحويل الموحد UTF-8 أو UTF اختصارًا للمصطلح الذي يترجم إلى صيغة تحويل نظام الحروف الدولي الموحد، وهذا الترميز وضع من قبل كل من روب بايك وكين تومسن لتمثيل معيار نظام الحروف الدولي الموحد للحروف الأبجدية لأغلب لغات العالم، وتُمثَّل فيه الرموز ذات العرض المتغير بحجم يتراوح بين بايت واحد وأربعة بايتات للرمز الواحد.

يدعم جو صيغة التحويل هذه دون أي إعداد خاص أو مكتبات أو حزم، ويمكن تمثيل الأحرف الرومانية مثل الحرف A بقيمة ASCII مثل الرقم 65، لكن سيكون ترميز UTF-8 للمحارف الخاصة مثل المحرف 世،مطلوبًا، كما يستخدِم جو النوع رون لبيانات UTF-8.

```
a := "Hello, 世界"
```

يمكنك استخدام الكلمة المفتاحية `range` داخل الحلقة `for` للتكرار على فهارس أيّ سلسلة نصية، إذ ستُغطى الحلقات لاحقًا في الفصول القادمة من الكتاب، كما تُستخدم الحلقة `for` في المثال التالي لحساب عدد البايتات ضمن سلسلة محددة:

```
package main
import "fmt"
func main() {
  a := "Hello, 世界"
  for i, c := range a {
   fmt.Printf("%d: %s\n", i, string(c))
  }
  fmt.Println("length of 'Hello, 世界': ", len(a))
}
```

عرّفنا المتغير `a` في المثال السابق، وأسندنا إليه السلسة النصية "Hello, 世界" التي تحتوي على محارف UTF-8، ثم استخدمنا حلقة `for` مع الكلمة المفتاحية `range` للتكرار على محارف السلسلة، إذ تُفهرس الكلمة المفتاحية `range` عناصر السلسلة المحددة وتعيد قيمتين الأولى هي فهرس البايت والثانية هي قيمة البايت -التي تمثِّل هنا محرف السلسلة- وبترتيب ورودها نفسه في السلسلة.

تُستخدَم الدالة `fmt.Printf` كما هي العادة من أجل الطباعة على الشاشة، حيث حددنا لها التنسيق بالصورة `%d: %s\n`، إذ تشير `%d` إلى وجود عدد صحيح مكانها وهو الفهرس `i` الذي مثّل المُعطى الثاني للدالة، والرمز `%s` إلى وجود سلسلة نصية أو محرف وهو المحرف `c` المُمثّل بالمُعطى الثالث والرمز الأخير للانتقال إلى سطر جديد، ولاحظ أيضًا أننا وضعنا `c` ضمن الدالة `string` وذلك كي تعامل معاملة سلسلة، ثم طبعنا أخيرًا طول المتغير `a` من خلال استخدام الدالة `len`.

كما ذكرنا سابقًا فإن الرون `rune` هو اسم بديل للنوع `int32` ويمكن أن يتكون من بـايت واحـد إلى أربعـة بايتات، فهو يستخدم 3 بايتات لتمثيل المحرف 世 والكلمة المفتاحية `range` تُدرك ذلك، وبالتالي عنـدما تصـل إليه تقرأ 3 بايتات متتالية على أنها فهرس لمحرف واحد وهذا واضح من الخرج التالي.

```
0: H
1: e
2: l
3: l
4: o
5: ,
6:
7: 世
10: 界
length of 'Hello, 世界': 13
```

لاحظ أننا حصلنا على القيمة 13 عند طباعة الطول، بينما حصلنا على عدد فهارس يساوي 8 وهذا يُفسّـر مـا ذكرناه في الأعلى.

لن تتعامل دائمًا مع سلاسل UTF-8 في برامج جو، ولكن عندما تعمل معها ستكون مُدركًا لسـبب تمثيلهـا باستخدام النوع رون `rune` الذي يمكن أن يكون طوله من 8 إلى 32 بت وليس `int32` واحدة فقط.

3.9 التصريح عن أنواع البيانات للمتغيرات

بعد أن تعرفت على أنواع البيانات الأولية المختلفة، سـننتقل الآن إلى شـرح كيفيـة إسـناد هـذه الأنـواع إلى المتغيرات في جو.

يمكننا تحديد متغير باستخدام الكلمة المفتاحيـة `var` متبوعـة باسـم المتغـير ونـوع البيانـات المطلـوب، إذ سنصرّح في المثال التالي عن متغـير يسـمى `pi` من النـوع `float64`، أي الكلمـة المفتاحيـة `var` هي أول مـا يُكتب عند التصريح عن متغير، ثم اسم المتغير ثم نوع البيانات:

```
var pi float64
```

يمكن إسناد قيمة ابتدائية للمتغير اختياريًا:

```
var pi float64 = 3.14
```

تُعَدّ جو لغةً ثابتة الأنـواع statically-typed language مثـل لغـة C و Java و C++، وهـذا يعـني أن كـل تعليمة في البرنامج تُفحَص في وقت التصريف، كما يعني أيضًا أنّ نوع البيانات مرتبط بالمتغير؛ أما في اللغات

ديناميكية الأنواع مثل بايثون أو PHP، فيرتبط نوع البيانات بالقيمة، فقد يُصرَّح مثلًا عن نوع البيانات في جو عند التصريح عن المتغير كما يلي:

```
var pi float64 = 3.14
var week int = 7
```

يمكن أن يكون كل من هذه المتغيرات نوع بيانات مختلف إذا عرّفته بطريقة مختلفة، لكن بعد التصريح عنه لأول مرة لن تكون قادرًا على تغيير ذلك؛ أما في لغة PHP فالأمر مختلف، إذ يرتبط نوع البيانات بالقيمة كما يلي:

```
$s = "sammy";        // يأخذ المتغير ديناميكًا نوع سلسلة نصية
$s = 123;            // يأخذ المتغير ديناميكيًا نوع عدد صحيح
```

كان المتغير s سلسلةً نصيةً ثم تغيَّر إلى عدد صحيح أوتوماتيكيًا تبعًا للقيمة المُسندة إليه.

سنتعرّف الآن على أنواع البيانات الأكثر تعقيدًا التي تخزن أكثر من قيمة بذات الوقت مثل المصفوفات.

3.10 المصفوفات Arrays

تُعَدّ المصفوفة متغيرًا واحدًا يتألف من تسلسل مرتب من العناصر ذات النوع نفسه، وكل عنصر في المصفوفة يمكن تخزين قيمة واحدة فيه. عناصر المصفوفة تتميز عن بعضها من خلال رقم محدد يعطى لكل عنصر يسمى فهرس index، وأول عنصر في المصفوفة يكون فهرسه 0 دائمًا.

تُحدَّد سعة المصفوفة لحظة إنشائها، وبمجرد تعريف حجمها لا يمكن تغييره، وبالتالي بما أنّ حجم المصفوفة ثابت، فهذا يعني أنه يخصص الذاكرة مرةً واحدةً فقط، وهذا يجعل المصفوفات غير مرنة نوعًا ما للعمل معها، لكنه يزيد من أداء برنامجك، ولهذا السبب تُستخدم المصفوفات عادةً عند تحسين البرامج.

تُعَدّ الشرائح Slices التي ستتعرف عليها بعد قليل أكثر مرونةً، إذ تُعَدّ نوعًا من البيانات القابلة للتغيير وتكوّن ما قد تعتقد أنه مصفوفات بلغات أخرى.

تُعرّف المصفوفات من خلال التصريح عن حجمها ثم نوع البيانات ثم القيم المحددة بين الأقواس المعقوصة {} كما يلي:

```
[capacity]data_type{element_values}
```

مثال:

```
[3]string{"blue coral", "staghorn coral", "pillar coral"}
```

يمكنك إسناد المصفوفة إلى متغير ثم طباعتها كما يلي:

```
coral := [3]string{"blue coral", "staghorn coral", "pillar coral"}
fmt.Println(coral)
```

سيكون الخرج كما يلي:

```
[blue coral staghorn coral pillar coral]
```

3.11 الشرائح Slices

تُعَدّ الشريحة تسلسلًا مرتبًا من العناصر وطولها قابلًا للتغيير، إذ يمكن للشرائح أن تزيد حجمها ديناميكيًا.

إذا لم يكن للشريحة حجم ذاكرة كافي لتخزين العناصر الجديدة عند إضافة عناصر جديدة إلى شريحة، فستطلب ذاكرةً أكبر من النظام حسب الحاجة، ولهذا السبب هي شائعة الاستخدام أكثر من المصفوفات.

يُصرّح عن الشرائح من خلال تحديد نوع البيانات مسبوقًا بقوس فتح وإغلاق مربع `[]` والقيم بين أقواس معقوصة `{}`، وإليك مثالًا عن شريحة من الأعداد الصحيحة:

```
[]int{-3, -2, -1, 0, 1, 2, 3}
```

شريحة من الأعداد الحقيقية:

```
[]float64{3.14, 9.23, 111.11, 312.12, 1.05}
```

شريحة من السلاسل النصية:

```
[]string{"shark", "cuttlefish", "squid", "mantis shrimp"}
```

يمكنك أيضًا إسنادها إلى متغير كما يلي:

```
seaCreatures := []string{"shark", "cuttlefish", "squid", "mantis
shrimp"}
```

ثم طباعة هذا المتغير:

```
fmt.Println(seaCreatures)
```

سيكون الخرج كالتالي:

```
[shark cuttlefish squid mantis shrimp]
```

يمكنك استخدام الكلمة المفتاحية `append` لإضافة عنصر جديد إلى الشريحة، وسنضيف السلسلة `seahorse` إلى الشريحة السابقة على سبيل المثال كما يلي:

```
seaCreatures = append(seaCreatures, "seahorse")
```

سنطبع الشريحة الآن للتأكد من نجاح العملية:

```
fmt.Println(seaCreatures)
```

سيكون الخرج كما يلي:

```
[shark cuttlefish squid mantis shrimp seahorse]
```

كما تلاحظ، إذا كنت بحاجة إلى إدارة حجم غير معروف من العناصر، فستكون الشريحة الخيار الأفضل.

3.12 الخرائط Maps

تُعَدّ الخرائط نوع بيانات مختلف يُمثّل قاموسًا أو رابطًا يربط قيمة مع مفاتحها، إذ تستخدِم الخرائط أزواج المفاتيح والقيم لتخزين البيانات، كما تُعَدّ مفيدةً جدًا عندما يتطلب الأمر البحث عن القيم بسرعة بواسطة الفهارس أو المفاتيح هنا، فمثلًا في حالة كان لديك عدة مستخدمين وتريد تسجيلهم في بنية بيانات تربط معلومات كل مستخدم -أي القيمة- بعدد فريد لهذا المستخدِم -أي مفتاح-، فالخرائط أُنشئت لهذا الأمر.

تُنشأ الخرائط في جو باستخدام الكلمة المفتاحية `map` متبوعة بنوع بيانات المفاتيح بين قوسين مربعين `[]` متبوعًا بنوع بيانات القيم، ثم أزواج القيم والمفاتيح في أقواس معقوصة `{}`.

```
map[key]value{}
```

عادةً ما تُستخدَم الخرائط للاحتفاظ بالبيانات المرتبطة ببعضها كما عرضنا في المثال السابق أو كما في المثال التالي:

```
map[string]string{"name": "Sammy", "animal": "shark", "color": "blue",
"location": "ocean"}
```

يمكنك أن تلاحظ أنه بالإضافة إلى الأقواس المعقوصة `{}`، توجد أيضًا نقطتان `:` في جميع أنحاء الخريطة، إذ تمثل الكلمات الموجودة على يسار النقطتين المفاتيح وعلى اليمين القيم، كما تجدر الملاحظة أيضًا إلى أنّ المفاتيح يمكن أن تكون من أيّ نوع بيانات في جو، فالمفاتيح الموجودة في الخريطة أعلاه هي الاسم والحيوان واللون والموقع، ويفضل أن تكون من الأنواع القابلة للمقارنة وهي الأنواع الأولية primitive types مثل السلاسل النصية `string` والأعداد الصحيحة `ints` وما إلى ذلك، إذ يُحدد النوع الأساسي من خلال اللغة ولا ينشأ من دمج أيّ أنواع أخرى، كما يمكن للمستخدِم أيضًا تحديد أنواع جديدة، إلا أنه يُفضّل إبقاءها بسيطةً لتجنب أخطاء البرمجة.

سنخزّن الخريطة أعلاه ضمن متغير ثم سنطبعها:

```
sammy := map[string]string{"name": "Sammy", "animal": "shark",
"color": "blue", "location": "ocean"}
fmt.Println(sammy)
```

سيكون الخرج كما يلي:

```
map[animal:shark color:blue location:ocean name:Sammy]
```

إذا أردت الوصول إلى لون سامي sammy وطباعته فيمكنك كتابة ما يلي:

```
fmt.Println(sammy["color"])
```

سيكون الخرج كما يلي:

```
blue
```

تُعَدّ الخرائط عناصر مهمة في كثير من البرامج التي قد تُنشأ في جو كونها توفِّر إمكانية فهرسة عملية الوصول إلى البيانات.

3.13 الخاتمة

إلى هنا تكون قد تعرّفت على أنواع البيانات الأساسية في جو إضافةً إلى بُنى البيانات المهمة مثل المصفوفات والشرائح والخرائط، إذ ستكوّن الفروق بين هذه الأنواع أهميةً كبيرةً لك أثناء تطوير مشاريعك باستخدام جو، كما أنّ فهمك الجيد لأنواع البيانات سيجعلك قادرًا على معرفة متى وكيف وفي أيّ وقت ستستخدِم أو ستغيّر نوع بيانات متغير حسب الحاجة.

دورة تطوير التطبيقات
باستخدام لغة بايثون

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

4. التعامل مع السلاسل النصية

تُعَدّ السلسلة النصية تسلسلًا من محرف واحد -أي حرف أبجدي أو عدد أو رمز- أو أكثر ويمكن أن تكـون ثابتًـا constant أو متغيرًا variable، كما تتبـع السلاسـل النصـية الترمـيز الموحـد يونيكـود Unicode (معيـار يُمكّن الحواسيب من تمثيل النصوص المكتوبة بأغلب نظم الكتابة ومعالجتها، بصورة متناسقة).

تُعَدّ السلاسل النصية في جو غير قابلة للتغيير Immutable، وهو المصطلح المعاكس لمصطلح mutable الذي يشير إلى أنواع البيانات التي يمكن التعديل على القيم التي تحتويها حتى بعد إنشاءها. أي أنـه وبعـد إنشـاء نوع البيانات وتخزينه في الذاكرة، يمكن التعديل على بياناته.

تدخل النصوص في كل شيء، لذا فإن نوع البيانات الـذي هـو السلسـلة النصـية string مهم جـدًا في كـل لغـات البرمجـة، وسـتتعلم في هـذا الفصـل كيفيـة إنشـاء السلاسـل وطباعتهـا وربطهـا وتكرارهـا وتخزينها في متغيرات.

4.1 صياغة السلاسل النصية String Literals

غالبًا ما تكون السلاسل عبارة عن نصوص عادية مكتوبة، لذا قد تـرغب في العديـد من الحـالات أن تحصـل على تحكم أكبر في كيفية ظهور هذه السلاسل بغية جعلها مقروءة أكثر للآخرين، وذلـك بإضـافة علامـات الـترقيم وفواصل الأسطر والمسافات البادئة ...إلخ.

وستتعرف فيما يلي على بعض الطرق التي يمكنك من خلالها تمثيل السلاسل النصـية بالصـياغة المطلوبـة في جو، ولكن يجب عليك في البداية فهم الفرق بين صياغة السلسلة النصية وقيمتهـا؛ فالسلسـلة النصـية مـع صياغتها هي ما نراه في الشيفرة المصدرية لبرنامج الحاسـوب بمـا في ذلـك علامتَي الاقتبـاس مثلًا؛ أمـا قيمـة السلسلة، فهي ما نراه عند التعامل مع السلسـلة عنـد تشـغيل البرنـامج مثـل اسـتدعاء الدالـة `fmt.Println` معها، ففي برنامج طباعة النص "Hello, World!" مثلًا، تكون صياغة السلسلة هي "Hello, World!" وقيمتها

هي Hello, World!، أي بدون علامتَي الاقتباس. إذًا قيمة السلسلة هي ما تراه خرجًا على نافذة الطرفية عند تشغيل برنامج جو وسيكون كالتالي:

```
Hello, World!
```

وقد تحتاج في بعض الأحيان إلى تضمين علامتَي الاقتباس بحيث تظهر في قيمة السلسلة عندما تقتبس من مصدر مثلًا، ولكن بما أنّ صياغة السلسلة وقيمتها غير متكافئين، فستحتاج غالبًا لإضافة تنسيق محدد إلى السلسلة لضمان ظهور علامة الاقتباس في قيمة السلسلة كما هو متوقع.

4.1.1 علامات الاقتباس وتفسيرها

تتيح لغة جو Go استخدام علامات الاقتباس الخلفية `` ` `` أو المزدوجة " " من أجل تعريف السلاسل النصية، فبالتالي عندما ترغب باستخدام علامات الاقتباس المزدوجة، فيمكنك تضمينها ضمن سلسلة تستخدِم علامات اقتباس خلفية والعكس بالعكس كما في المثال التالي:

```
Sammy says, "Hello!"
```

أو الحالة المعاكسة:

```
"Sammy likes the `fmt` package for formatting strings.."
```

يمكنك بهذه الطريقة الجمع بين علامات الاقتباس الخلفية والمزدوجة والتحكم في عرض علامات الاقتباس داخل السلسلة النصية، لكن تذكّر أنّ استخدام علامات الاقتباس الخلفية في جو سيؤدي إلى إنشاء سلسلة نصية أولية raw string literal واستخدام علامات الاقتباس المزدوجة سيؤدي إلى إنشاء سلسلة نصية مُفسّرة interpreted string literal.

4.1.2 محارف الهروب Escape Characters

يمكنك استخدام محارف خاصة داخل السلاسل النصية لها معنى خاص بالنسبة للغة جو مثل " ولكن ماذا لو أردنا استعمال علامة الاقتباس " نفسها ضمن سلسلة نصية؟ نلجأ هنا إلى مفهوم الهروب والذي يعني إخبار المُصرِّف أن لا يفسر المحرف التالي له بمعناه الخاص مثل تهريب علامة الاقتباس " والعكس أيضًا أي إعطاء معنى خاص مع محارف محددة إن وجدت ضمن السلسلة النصية مثل محرف السطر الجديد \n، إذ تبدأ جميع محارف الهروب بمفتاح الشرطة المائلة للخلف \ يليها مباشرةً المحرف المراد تهريبه داخل السلسلة، ومن أبرز حالات استعمال محرف التهريب:

- \: تهريب تفسير الشرطة المائلة \ بمعناها الخاص أي محرف التهريب نفسه.
- \": تهريب علامة اقتباس مزدوجة.

- `\n`: إعطاء الحرف n معنى خاص وهو تفسيره إلى الانتقال إلى السطر التالي (أول حـرف من كلمـة new line أي سطر جديد) أي كأنك تضغط على حرف الإدخال Enter.
- `\t`: إعطاء الحرف t معنى خاص وهو مفتاح الجدولة Tab، أي تفسيره بمسافة بادئة أفقية.

سنستخدِم في المثال التالي محرف الهروب `\"` لإضافة علامات الاقتباس:

```
fmt.Println("Sammy says, \"Hello!\"")
```

سيكون الخرج كما يلي:

```
Sammy says, "Hello!"
```

إذًا باستخدام محرف الهروب `\"` يمكنك استخدام علامات الاقتباس المزدوجة لإحاطة سلسلة تتضمن نصًـا بين علامتَي اقتباس مزدوجتين، وسنستخدم الآن محرف هروب آخر وهو `\n` للانتقال إلى السطر التالي:

```
fmt.Println("This string\nspans multiple\nlines.")
```

سيكون الخرج كما يلي:

```
This string
spans multiple
lines.
```

يمكنك أيضًا استخدام عدة محارف هروب ضمن السلسلة في الوقت نفسه كما في المثال التالي:

```
fmt.Println("1.\tShark\n2.\tShrimp\n10.\tSquid")
```

سيكون الخرج كما يلي:

```
\1.     Shark
\2.     Shrimp
\10.    Squid
```

يساعدك محرف الهروب `\t` في عرض البيانـات المُجدولـة بمحـاذاة أفقيـة واحـدة، إذ يسـتبدلها المُصـرِّف بمسافة بادئة أفقية بطول ثابت مما يساعدك في جعل البيانات المعروضة قابلة للقراءة بوضوح.

تُستخدَم محارف الهروب لإضافة تنسيق إضافي إلى السلاسل الـتي قـد تكـون من الصـعب أو المسـتحيل تحقيقها، إذ لن تتمكن بدون محارف الهروب من إنشاء سلسلة مثل:

```
 Sammy says, "I like to use the fmt package".
```

4.1.3 السلسلة النصية الأولية Raw String Literal

لنفرض مثلًا أنك لا تريد استخدام تنسيق خاص داخل السلاسل النصية، فقد ترغب بمقارنة أو تقييم سلاسل شيفرة الحاسوب التي تستخدِم الشرطة المائلة للخلف، لذلك لا تريد أن يستخدمها جو على أساس محرف هروب، وبالتالي تدل السلسلة النصية الأولية على أن لغة جو ستتجاهل جميع التنسيقات والتفسيرات داخل السلسلة بما في ذلك محارف الهروب.

سننشئ هنا سلسلةً أوليةً باستخدام علامتَي الاقتباس الخلفية:

```
fmt.Println(`Sammy says,\"The balloon\'s color is red.\"`)
```

سيكون الخرج كما يلي:

```
Sammy says,\"The balloon\'s color is red.\"
```

نستنتج أنّ استخدام سلسلة نصية أولية يؤدي إلى تجاهل علامات الاقتباس وأية محارف وتنسيقات خاصة يمكن للغة جو أن تفسرها بمعنى مختلف، وبالتالي يمكنك الاحتفاظ بالشرطة المائلة العكسية والمحارف الأخرى التي تُستخدم كمحارف هروب.

4.2 السلاسل النصية المفسرة

هي سلسلة نصية موضوعة بين علامتي اقتباس مزدوجتين " ". كل شيء موجود بداخل علامتي الاقتباس سيُعرض باستثناء الأسطر الجديدة (المُعبر عنها بالرمز \n) وعلامات الاقتباس المزدوجة. وفي حال أردت أن تظهر هذه الرموز أيضًا، يمكنك استخدام مفتاح الهروب / قبلها.

على سبيل المثال:

```
fmt.Println("Say \"hello\" to Go!")
```

سيكون الخرج كما يلي:

```
Say "hello" to Go!
```

غالبًا ستستخدم السلاسل المُفسرة لأنها تتعامل مع المحارف الخاصة وتسمح بتخطيها أيضًا (أي أنها تمنحك تحكمًا أكبر).

4.3 طباعة السلاسل النصية على عدة أسطر

تسهل طباعة السلاسل النصية على أسطر متعددة قابليةً قراءة النص، وتمكّنك أيضًا من تجميع السلاسل لإنشاء نص مُنظّم ونظيف أو استخدامها للاحتفاظ بفواصل الأسطر في حال كانت تعبر عن قصيدة مثلًا.

يمكنك كتابة السلسلة بالطريقة التي تريد أن تظهر بها تمامًا وإحاطتها بعلامتَي الاقتباس الخلفية لإنشاء سلاسل تمتد على عدة أسطر، لكن لا تنسَ أنّ ذلك سيجعل منها سلسلةً نصية أوليةً.

```
`
This string is on
multiple lines
within three single
quotes on either side.
`
```

سيكون الخرج على النحو التالي:

```
This string is on
multiple lines
within three single
quotes on either side.
```

لاحظ وجود سطر بداية ونهاية فارغين، ولتجنب ذلك ضع السطر الأول فورًا بعد علامة الاقتباس الخلفية وفي السطر الأخير ضع علامة الاقتباس الخلفي مباشرةً بعد آخر كلمة.

```
`This string is on
multiple lines
within three single
quotes on either side.`
```

وإذا كنت بحاجة إلى إنشاء سلسلة مفسرة متعددة الأسطر، فيمكنك إنجاز ذلك بعلامتَي اقتباس مزدوجة واستخدام العامِل +، ولكن ستحتاج إلى إدخال فواصل الأسطر من خلال محرف الهروب في الأماكن التي تريدها كما في المثال التالي:

```
"This string is on\n" +
"multiple lines\n" +
"within three single\n" +
"quotes on either side."
```

بالرغم من أنّ علامتَي الاقتباس الخلفية تُسهّل طباعة النص الطويل وقراءته، لكن إذا كنت بحاجة إلى سلسلة نصية مفسرة، فستحتاج إلى استخدام علامتَي الاقتباس المزدوجة.

4.4 طباعة السلاسل

يمكنك طباعة السلاسل باستخدام حزمة `fmt` من مكتبة النظام، ثم استدعاء الدالة `Println()` منها:

```
fmt.Println("Let's print out this string.")
```

سيكون الخرج كما يلي:

```
Let's print out this string.
```

يجب عليك استيراد حزم النظام عند استخدامها، وبالتالي سيكون البرنامج اللازم لطباعة السلسلة كما يلي:

```
package main
import "fmt"
func main() {
  fmt.Println("Let's print out this string.")
}
```

4.5 ربط السلاسل

يُقصَد بربط السلاسل Concatenation ضم أو وصل السلاسل معًا لإنشاء سلسلة جديدة، إذ يمكنك ربط السلاسل باستخدام العامِل `+`، وضَع في الحسبان أنه عند التعامل مع الأرقام سيكون العامل `+` عاملًا للجمع، ولكن عند استخدامه مع السلاسل فهو عامل وصل أو دمج.

سنربط في المثال التالي السلسلة "Sammy" والسلسلة "Shark" لإنشاء سلسلة نصية جديدة، وسنطبع الناتج مباشرةً من خلال دالة الطباعة كما يلي:

```
fmt.Println("Sammy" + "Shark")
```

سيكون الخرج:

```
SammyShark
```

يمكنك إضافة مسافة فارغة للفصل بين الكلمتَين من خلال إضافتها بين علامتي الاقتباس بعد الكلمة الأولى كما يلي:

```
fmt.Println("Sammy " + "Shark")
```

سيكون الخرج:

```
Sammy Shark
```

لا يمكنك استخدام العامل + بين نوعين مختلفين من البيانات، فلا يمكنك مثلًا ربط السلاسل والأعداد الصحيحة معًا، وإذا حاولت ذلك:

```
fmt.Println("Sammy" + 27)
```

فسيظهر الخطأ التالي:

```
cannot convert "Sammy" (type untyped string) to type int
invalid operation: "Sammy" + 27 (mismatched types string and int)
```

فإذا أردت إنشاء السلسلة النصية "Sammy27"، فيمكنك ذلك بوضع العدد 27 بين علامتَي اقتباس "27" بحيث لا يكون عددًا صحيحًا وإنما سلسلةً إذ يكون استخدام الأعداد ضمن السلاسل مفيدًا في كثير من الحالات مثل الرموز البريدية أو أرقام الهواتف، فقد ترغب مثلًا بجعل رمز البلد ورمز المنطقة معًا بدون فاصل بينهما.

عندما تربط سلسلتين فهذا يعني أنك ستُنشئ سلسلةً جديدةً يمكنك استخدامها في جميع أنحاء برنامجك.

4.6 تخزين السلاسل في المتغيرات

تطرقنا في الفصل السابق إلى مفهوم المتغيرات وأنها مفهوم برمجي مهم يشير إلى القيم ونوع القيم التي تستخدمها في برنامجك، وتعرفنا كيف يحدد نوع المتغير نوع البيانات التي يمكن تخزينها فيه، وما هي العمليات التي يمكن إجراؤها عليه، وبأن بإمكانك التفكير في المتغيرات على أساس صندوق فارغ يمكنك ملؤه بقيمة أو ببيانات، وتُعَدّ السلاسل النصية بيانات، لذا يمكنك استخدامها لملء قيم المتغيرات، كما يسهّل استخدام المتغيرات لتخزين السلاسل العمل مع السلاسل في أي برنامج.

لتخزين سلسلة داخل متغير، ما عليك سوى إسناد السلسلة إلى متغير من النوع `string`، وسنخزّن هنا مثلًا سلسلةً داخل المتغير الذي أسميناه `s`، كما يمكنك استخدام أيّ اسم تراه مُعبّرًا، لكن يُفضل استخدام الأسماء القصيرة:

```
s := "Sammy likes declaring strings."
```

يمكنك الآن طباعة هذا المتغير الذي يُمثّل السلسلة السابقة:

```
fmt.Println(s)
```

سيكون الخرج:

```
Sammy likes declaring strings.
```

يُسهّل عليك استخدام المتغيرات مع السلاسل معالجة السلاسل والتعامل معها كثيرًا، فكل ما عليك فعله هو تخزينها في متغير مرةً واحدة لتتمكن من استخدام هذا المتغير الذي يحملها في أيّ مكان في برنامجك.

4.7 التعامل مع السلاسل النصية ومعالجتها

تحتوي حزمة `strings` في لغة جو على العديد من الدوال المتاحة للعمل مع نوع بيانات السلسلة النصية `string`، تمكّنك هذه الدوال من تعديل السلاسل والتعامل معها بسهولة، إذ تُعَدّ هذه الدوال إجراءات تتمثل بمجموعة من التعليمات البرمجية التي تُطبق على السلاسل لأهداف مختلفة.

حزمة السلاسل في جو هي حزمة مدمجة في لغة جو، فعند تثبيت جو تُثبّت معها، وبما أنها حزمة مُدمجة فكل الدوال التابعة لها مدمجة أيضًا ومتاحة للعمل مباشرةً، وستتعرف فيما يلي على هذه الدوال واستخداماتها.

4.7.1 تبديل حالة الأحرف من صغير إلى كبير والعكس

تُستخدَم الدالتان `strings.ToUpper` و `strings.ToLower` للتبديل بين حالتَي الأحرف الكبيرة والصغيرة، حيث تحوّل الدالة الأولى كل محارف السلسلة الأولى المُعطاة إلى الحالة الكبيرة وتعيدها على أساس سلسلة جديدة وبالطبع إذا كان الحرف كبير أصلًا، فلن يتغير، في حين تحوّل الدالة الثانية المحارف إلى الحالة الصغيرة وتعيدها على أساس سلسلة جديدة.

سنحوّل في المثال التالي كل محارف السلسلة "Sammy Shark" إلى حالتها الكبيرة:

```
ss := "Sammy Shark"
fmt.Println(strings.ToUpper(ss))
```

سيكون الخرج:

```
SAMMY SHARK
```

يمكنك أيضًا تحويلها إلى الحالة الصغيرة:

```
fmt.Println(strings.ToLower(ss))
```

سيكون الخرج:

```
sammy shark
```

بما أنك تستخدم دوال السلاسل، فلابد أن ستستورد الحزمة أولًا، وإلا فلن يتعرّف المُصرِّف على تلك الدوال:

```
package main
import (
  "fmt"
  "strings"
)
```

```
func main() {
  ss := "Sammy Shark"
  fmt.Println(strings.ToUpper(ss))
  fmt.Println(strings.ToLower(ss))
}
```

تفيد الدالتان `strings.ToUpper` و `strings.ToLower` كثيرًا في حالة مقارنة السلاسل، لأنه يجعل حالة المحارف بحالة متسقة، فإذا قارنت مثلًا بين كلمتَي Sammy و sammy، فستكون النتيجة أنهما غير متماثلتان لأن جو حساسة لحالة الأحرف سواءً بالنسبة للقيم أو المتغيرات، لكن إذا طبقت الدالة `strings.ToLower` على الكلمة الأولى قبل المقارنة، سيكون الناتج أنهما متماثلتان، ومن هنا تأتي أهميتهما.

4.7.2 دوال البحث في السلاسل

تحتوي حزمة السلاسل النصية في جو على العديد من الدوال التي تمكّنك من إنجاز عمليات البحث في السلاسل مثل البحث عن تسلسل محدد من المحارف داخل سلسلة ما، ومن أشهر هذه الدوال:

- `strings.HasPrefix` تبحث في السلسلة من بدايتها.
- `strings.HasSuffix` تبحث في السلسلة من نهايتها.
- `strings.Contains` تبحث في أيّ مكان في السلسلة.
- `strings.Count` تعيد عدد المرات التي ظهرت بها السلسلة ضمن السلسلة المعطاة.

تسمح لك الدالة الأولى والثانية بالبحث عن محرف أو سلسلة من المحارف المحددة في بداية أو نهاية السلسلة، أي اختبار فيما إذا كانت تنتهي أو تبدأ بمجموعة من المحارف، فلاختبار فيما إذا كانت السلسلة "Sammy Shark" تبدأ بكلمة Sammy وتنتهي بكلمة Shark مثلًا، سنكتب ما يلي:

```
ss := "Sammy Shark"
fmt.Println(strings.HasPrefix(ss, "Sammy"))
fmt.Println(strings.HasSuffix(ss, "Shark"))
```

سيكون الخرج:

```
true
true
```

يمكنك استخدام الدالة `strings.Contains` مثلًا لاختبار احتوائها على السلسلة "Sh":

```
fmt.Println(strings.Contains(ss, "Sh"))
```

سيكون الخرج:

```
true
```

يمكنك أيضًا حساب عدد مرات ظهور المحرف S في السلسلة Sammy Shark:

```
fmt.Println(strings.Count(ss, "S"))
```

سيكون الخرج:

```
2
```

سنجرب الآن حساب عدد مرات ظهور الحرف s بالحالة الصغيرة:

```
fmt.Println(strings.Count(ss, "s"))
```

سيكون الخرج:

```
0
```

> تذكّر ما أشرنا إليه سابقًا بخصوص حساسية لغة جو لحالة الأحرف وتمييزها بين الأحرف الكبيرة والصغيرة، فهذا سيُفسّر نتيجة الخرجَين السابقَين.

4.7.3 تحديد طول السلسلة

تُستخدَم الدالة len لحساب عدد المحارف الموجودة ضمن سلسلة نصية ما -أي طولها-، وهذه الدالة مفيدة في الكثير من الحالات مثل معرفة طول السلسلة، أو قص السلاسل الأطول من طول محدد، أو إجبار المستخدِم على إدخال طول محدد لكلمة المرور، وفي المثال التالي سنحسب طول سلسلة محددة:

```
import (
  "fmt"
  "strings"
)
func main() {
   openSource := "Sammy contributes to open source."
   fmt.Println(len(openSource))
}
```

سيكون الخرج:

```
33
```

في المثال أعلاه أسندنا السلسلة "Sammy contributes to open source." إلى المتغير openSource ثم مررنا المتغير openSource إلى الدالة len لحساب طولها ووضعناها ضمن الدالة fmt.Println لطباعة النتيجة مباشرةً، ويجب أن تضع في ذهنك أنّ الدالة len ستحسب طول السلسلة مع احتساب أي شيء ضمنها مثل الرموز والأرقام والفراغات ...إلخ.

4.7.4 دوال معالجة السلاسل النصية وتعديلها

توجد دوال إضافية للتعامل مع السلاسل مثل strings.Join و strings.Split و strings.ReplaceAll، إذ تستخدَم الدالة strings.Join لربط شريحة من السلاسل مع بعضها مع إمكانية فصلها بفاصل محدد كما في المثال التالي:

```
fmt.Println(strings.Join([]string{"sharks", "crustaceans",
"plankton"}, ","))
```

سيكون الخرج:

```
sharks,crustaceans,plankton
```

كان الفاصل هنا هو فاصلة ",", كما يمكنك مثلًا إضافة فراغ بعد الفاصلة أيضًا كما يلي:

```
strings.Join([]string{"sharks", "crustaceans", "plankton"}, ", ")
```

يمكنك استخدام الدالة strings.Split أيضًا لتقسيم السلاسل اعتمادًا على فاصل محدد كما في المثال التالي:

```
balloon := "Sammy has a balloon."
s := strings.Split(balloon, " ")
fmt.Println(s)
```

سيكون الخرج شريحةً من السلاسل كما يلي:

```
[Sammy has a balloon]
```

قد يكون من الصعب تحديد محتوى هذه الشريحة بمجرد النظر إلى خرج الدالة strings.Println، لذا يمكنك استخدام الدالة fmt.Printf مع الرمز %q لطباعة السلاسل مع علامتَي الاقتباس كما يلي:

```
fmt.Printf("%q", s)
```

سيكون الخرج:

```
["Sammy" "has" "a" "balloon."]
```

توجد دالة أخرى مشابهة للدالة السابقة هي strings.Fields، لكن هذه الدالة تتجاهـل أيّـة فراغـات في السلسلة وبالتالي لا تقسم إلى الحقول الفعلية:

```
data := " username password     email date"
fields := strings.Fields(data)
fmt.Printf("%q", fields)
```

سيكون الخرج:

```
["username" "password" "email" "date"]
```

تستخدَم الدالة strings.ReplaceAll لاستبدال محرف أو عدة محارف في السلسـلة بمحـرف جديـد أو أكثر، إذ سنمرر لهذه الدالة السلسلة المطلوب تعديلها على أساس وسيط أول، ففي المثال التالي سنمرر المتغير balloon الذي يحمل القيمة Sammy has a balloon ثم سنمرر لها السلسلة المطلوب استبدالها وهي has ثم السلسة البديلة وهي had كما يلي:

```
fmt.Println(strings.ReplaceAll(balloon, "has", "had"))
```

سيكون الخرج :

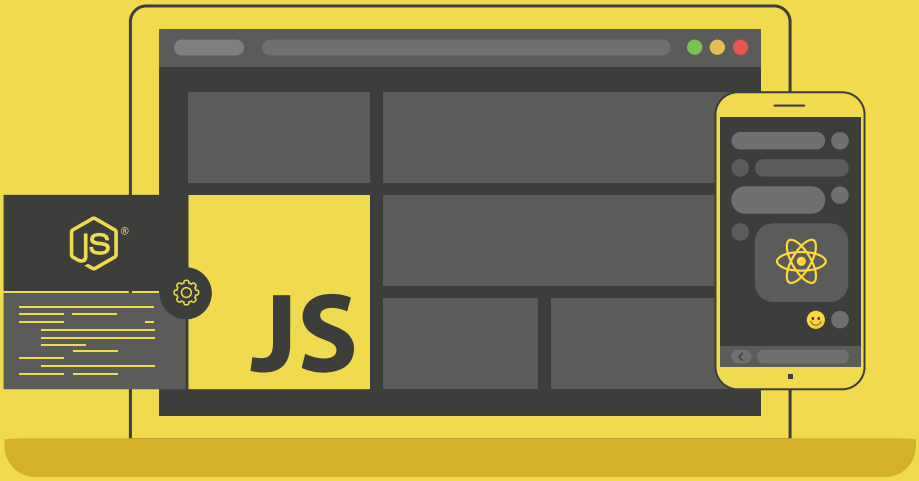
```
Sammy had a balloon.
```

سيكون لديك تحكم كبير في السلاسل من خلال الدوال السابقة.

4.8 الخاتمة

تعلمت في هذا الفصـل أساسـيات العمـل مـع السلاسـل النصـية في لغـة جـو بـدءًا من إنشـاء السلاسـل وطباعتها ودمجها والتعامل معها كمتغيرات ضمن برنامجك، كما تعلمت عدة طـرق لصـياغة السلاسـل النصـية وتنسيقها بالإضافة إلى تقنيات مثل محارف الهروب التي تمكّنك من عرض السلاسل النصية في برنامجـك على الشاشة كما تريد لكي يتمكن المستخدِم النهائي من قراءة كل المخرجات بسـهولة، ثم تعـرفت أخـيرًا على أشـهر الدوال المستخدَمة في معالجة السلاسل النصية وتعديلها.

دورة تطوير التطبيقات
باستخدام لغة JavaScript



مميزات الدورة

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

اشترك الآن

5. استخدام المتغيرات والثوابت

سنشرح في هذا الفصل المزيد حول المتغيرات التي تعد مفهومًا برمجيًا مهمًا يتوجّب عليك فهمه وإتقانه جيدًا للتعامل مع القيم التي تستخدِمها في برنامجك بشكل صحيح، كما سنشرح أفضل الممارسات عند التعامل معها، وسنتطرق لتوضيح مفهوم الثوابت والفرق بينها وبين المتغيرات ومتى نستخدم كل منها.

5.1 فهم المتغيرات

تُخصَّص للمتغير من الناحية الفنية مساحة تخزينية في الذاكرة توضَع القيمة المرتبطة به فيها، ويُستخدَم اسم المتغير للإشارة إلى تلك القيمة المُخزَّنة في ذاكرة البرنامج والتي هي جزء من ذاكرة الحاسوب، وسمكن تشبيه المُتغيِّر بأنه عنوان تُعلِّقه على قيمة مُعيَّنة كما في الصورة التالية، إذ تشير variable name هنا إلى اسم المتغير وvalue إلى قيمته:

لنفترض أنه لدينا عددًا صحيحًا يساوي 1032049348 ونريد تخزينه في متغيِّر بدلًا من إعادة كتابة هذا العدد الطويل كل مرة، لذلك سنستخدِم شيئًا يُسهِّل تذكّره مثل المتغير `i`،

```
i := 1032049348
```

إذا نظرنا إليه على أنَّه عنوانٌ مرتبط بقيمة، فسيبدو على النحو التالي:

يمثّل `i` اسم المتغير وتُربَط به القيمة `1032049348` التي هي من نوع عدد صحيح `int`، كما تُعَدّ العبـارة `i := 1032049348` أنها تعليمة إسناد وتتألف من الأجزاء التالية:

- اسم المتغير `i`.
- معامِل تعريف المتغير المختصر `:=`.
- القيمة التي أُسنِدَت إلى المتغير `1032049348`.
- وبالنسبة لنوع البيانات فتكتشفه لغة جو تلقائيًا.

تُسند القيمة السابقة من خلال هذه الأجزاء إلى المتغير `i`، ونكون عند تحديد قيمة المتغير قد هيّئنا أو أنشأنا ذلك المتغير، وبعد ذلك يمكنك استخدام ذلك المتغير بدلًا من القيمة كما في المثال التالي:

```
package main
import "fmt"
func main() {
  i := 1032049348
  fmt.Println(i)
}
```

ويكون الخرج كما يلي:

```
1032049348
```

يسهل استخدام المتغيرات علينا إجـراء العمليـات الحسـابية، إذ سـنعتمد في المثـال التـالي على التعليمـة السابقة `i = 1032049348` وسنطرح من المتغير `i` القيمة `813` كما يلي:

```
fmt.Println(i - 813)
```

ويكون الخرج:

```
1032048535
```

تجري لغة جو Go العملية الحسابية وتطرح 813 من المتغير i وتعيد القيمة 1032048535.

يمكن ضبط المتغيرات وجعلها تساوي ناتج عملية حسابية ما، إذ سنجمع الآن عددين معًا ونخزِّن قيمة المجموع في المتغير x:

```
x := 76 + 145
```

يشبه المثال أعلاه إحدى المعادلات التي تراها في كتب الجبر، إذ تُستخدَم المحارف والرموز لتمثيل الأعداد والكميات داخل الصيغ والمعادلات، وبصورة مماثلة تُعَدّ المتغيرات أسماءً رمزيةً تمثِّل قيمةً من نوع بيانات معيّن، والفرق في لغة جو أنه عليك التأكد دائمًا من كتابة المتغير على الجانب الأيسر من المعادلة، ولنطبع الآن قيمة x كما يلي:

```
package main
import "fmt"
func main() {
  x := 76 + 145
  fmt.Println(x)
}
```

يكون الخرج:

```
221
```

أعادت جو القيمة 221 فقد أُسنِد إلى المتغير x مجموع العددين 76 و 145، كما يمكن أن تمثل المتغيرات أيّ نوع بيانات وليس الأعداد الصحيحة فقط كما يلي:

```
s := "Hello, World!"
f := 45.06
b := 5 > 9 // ستُرجع قيمة منطقية، إما صواب أو خطأ
array := [4]string{"item_1", "item_2", "item_3", "item_4"}
slice := []string{"one", "two", "three"}
m := map[string]string{"letter": "g", "number": "seven", "symbol":
"&"}
```

فإذا طبعت أيًّا من المتغيرات المذكورة أعلاه، فستعيد جو قيمة المتغير، ففي الشيفرة التالية مثلًا سنطبع متغيرًا يمثّل شريحة slice من سلسلة نصية:

```
package main
import "fmt"
```

```
func main() {
  slice := []string{"one", "two", "three"}
  fmt.Println(slice)
}
```

يكون الخرج:

```
[one two three]
```

لقد أسندنا الشريحة `[]string{"one", "two", "three"}` إلى المتغير `slice` ثم استخدمنا دالة الطباعة `fmt.Println` لطباعتها.

تأخذ المتغيرات مساحةً صغيرةً من ذاكرة الحاسوب لتسمح لك بوضع القيم في تلك المساحة.

5.2 التصريح عن المتغيرات

هناك عدة طرق للتصريح عن متغير في جـو، فيمكن التصـريح عن متغـير يسـمى `i` من نـوع البيانـات `int` بدون تهيئة، أي بدون قيمة أوليّة كما يلي:

```
var i int
```

يمكن تهيئة المتغير من خلال استخدام المعامِل `=` كما يلي:

```
var i int = 1
```

يُطلق على كل من هذين النموذجين للتصريح بالتصريح الطويل للمتغيرات long variable declaration، في حين يمثِّل النموذج التالي الأسلوب القصير أو المختصر short variable declaration:

```
i := 1
```

في هـذه الحالـة نحن لا نحـدد نـوع البيانـات ولا نسـتخدِم الكلمـة المفتاحيـة var، حيث يسـتنتج جـو الصنف تلقائيًّا.

تبنّى مجتمع جو المصطلحات التالية من خلال الطرق الثلاث للتصريح عن المتغيرات:

- استخدِم النموذج الطويل `var i int` عندما لا تُهيئ المتغير فقط.
- استخدِم النموذج المختصر `i := 1`، عند التصريح والتهيئة.
- إذا لم تكن ترغب بأن تستنتج لغة جو نـوع البيانـات الخاصـة بـك، ولكنـك لا تـزال تـرغب في اسـتخدام تصريح قصير للمتغير، فيمكنك تمرير القيمة إلى باني نوع البيانات الذي تريده كما يلي:

```
i := int64(1)
```

في حين لا يُعَدّ استخدام النموذج الطويل التالي من المصطلحات الشائعة في جو:

```
var i int = 1
```

> من الأفضل اتباع الطريقة التي يحدد فيها مجتمع جو عادات التصريح عن المتغيرات من الممارسات الجيدة، وذلك حتى يتمكن الآخرون من قراءة برامجك بسلاسة.

5.3 القيم الصفرية Zero Values

تأخذ أنواع البيانات المُعرّفة ضمن جو قيمًا صفرية على أساس قيمة أوليّة في حال لم تحدَّد لها قيمـة أوليـة كما في المثال التالي:

```
package main
import "fmt"
func main() {
  var a int
  var b string
  var c float64
  var d bool
  fmt.Printf("var a %T = %+v\n", a, a)
  fmt.Printf("var b %T = %q\n", b, b)
  fmt.Printf("var c %T = %+v\n", c, c)
  fmt.Printf("var d %T = %+v\n\n", d, d)
}
```

يكون الخرج كما يلي:

```
var a int = 0
var b string = ""
var c float64 = 0
var d bool = false
```

يُستخدم الرمز `%T` داخل الدالة `fmt.Printf` لطباعة نوع بيانات المتغير. لاحظ أن القيمة الصفرية المقابلة للسلاسل النصية تمثّلها السلسلة `""` والقيمة الصفرية للبيانات المنطقيـة `bool` تمثّلهـا القيمـة `false`. هـذا مهم ففي بعض اللغات توضع قيم عشوائية للمتغيرات في حال لم تُهيئ بقيمة أولية، وبالتالي قد نرى أن متغيرًا

منطقيًا يأخذ القيمة None أو شيء آخر، وهذا يُنتج عدم اتساقية للبيانات، فالصنف bool لا يمكن أن يكون إلا false أو true.

5.4 قواعد تسمية المتغيرات

تتميز تسمية المتغيرات بمرونة عالية، ولكن هناك بعض القواعد التي عليك أخذها في الحسبان كما يلي:

- يجب أن تكون أسماء المتغيرات من كلمة واحدة فقط بدون مسافات.
- يجب أن تتكوّن أسماء المتغيرات من المحارف والأعداد والشرطة السفلية _ فقط.
- لا ينبغي أن تبدأ أسماء المتغيرات بعدد.

دعنا نلقي نظرةً على بعض الأمثلة باتباع القواعد المذكورة أعلاه:

صالح	غير صالح	التفسير
user-Name	userName	غير مسموح استخدام الواصلات Hyphens
4i	i4	لا يمكن البدء برقم
user	$user	لا يمكن استخدام أيّ رمز غير الشرطة السفلية
user Name	userName	لا ينبغي للمتغير أن يتكون من أكثر من كلمة واحدة

من الأمور التي يجب أخذها في الحسبان عند تسمية المتغيرات هو أنَّها حساسة لحالة المحارف، وهذا يعني أنَّ my_int و MY_INT و My_Int و mY_iNt كلها مختلفة، أي ينبغي أن تتجنب استخدام أسماء متغيرات متماثلة لضمان ألا يحدث خلط عندك أو عند المتعاونين معك حاليًا أو في المستقبل، كما أنّ حالة المحرف الأول من المتغير لها معنى خاص في جو، فإذا بدأ المتغير بمحرف كبير، فيمكن الوصول إلى هذا المتغير من خارج الحزمة التي أُعلن عنه فيها؛ أما إذا بدأ المتغير بمحرف صغير، فهو متاح فقط داخل الحزمة التي أُعلن عنه فيها كما في المثال التالي:

```
var Email string
var password string
```

يبدأ اسم المتغير Email هنا بحرف كبير، وبالتالي يمكن الوصول إليه بواسطة حزم أخرى، في حين يبدأ المتغير password بحرف صغير ولا يمكن الوصول إليه إلا داخل الحزمة المُعلن عنه فيها.

من الشائع في جو استخدام أسماء متغيرات مختَصرة جدًا أو قصيرة، إذ يُفضّل كتابة user بدلًا من كتابة userName، كما يلعب النطاق Scope دورًا في اختصار اسم المتغير، فكلما كان نطاق المتغير أصغر، كلما كان اسم المتغير أصغر:

```
names := []string{"Mary", "John", "Bob", "Anna"}
```

```
for i, n := range names {
  fmt.Printf("index: %d = %q\n", i, n)
}
```

عند استخدام المتغيرات نفسها في أماكن مختلفة من البرنامج، يُفضّل إعطاءها أسماء واضحة لكي لا يربك ذلك الأشخاص الآخرين الذين يقرأون شيفرة البرنامج، لكن في المثال أعلاه استخدمنا المتغيرين `i` و `n` لأننا نستخدِمهما مرةً واحدةً فقط ومباشرةً، لذا لن يسبب ذلك أيّ مشكلة في قراءة أو فهم الشيفرة.

يُفضّل عند تسمية المتغيرات استخدام تنسيق سنام الجَمل camelCase أو سنام الجمل المرتفع CamelCase بدلًا من استعمال الشرطة السفلية multi_word أو العادية multi-word.

يتضمن الجدول التالي بعض الملاحظات المفيدة:

شائع	غير شائع	التفسير
`user_name`	`userName`	الشرطات السفلية غير شائعة الاستخدام
`index`	`i`	لا يُفضّل استخدام الاسم الكامل وإنما الاسم المختَصر
`serveHttp`	`serveHTTP`	يجب كتابة الاختصارات بمحارف كبيرة

الخيار الأهم الذي عليك التمسك به هو الاتساق في أسلوب التسمية، فإذا بدأت العمل على مشروع يستخدِم تنسيق سنام الجَمل في تسمية المتغيرات، فمن الأفضل الاستمرار في استخدام ذلك التنسيق حتى نهاية المشروع.

5.5 إعادة إسناد قيم للمتغيرات Reassigning

يمكن تغيير قيم المتغيرات في جو بسهولة كما تشير كلمة "متغير"، وهذا يعني أنه يمكنك إسناد قيمة مختلفة إلى متغير قد أُسنِدَت له قيمة مسبقًا بسهولة بالغة، إذ تُعَدّ القدرة على إعادة الإسناد مفيدةً للغاية، فقد تحتاج مثلًا خلال أطوار برنامجك إلى استلام قيم جديدة من المستخدِم وإسنادها إلى متغير قد حُـدّدَت قيمته سابقًا أو قد تحتاج إلى تغيير قيمة متغير اعتمادًا على أحداث معينة، كما أنّ سهولة إعادة إسناد قيم المتغيرات مفيدة أيضًا في البرامج الكبيرة التي قد تحتاج خلالها إلى تغيير القيم باستمرار.

سنُسّند إلى المتغير `i` الذي نوعه `int` العدد `76` ثم نعيد إسناده بالقيمة `42`:

```
package main
import "fmt"
func main() {
  i := 76
  fmt.Println(i)
```

```
    i = 42
    fmt.Println(i)
}
```

يكون الخرج:

```
76
42
```

يوضِّح المثال أنه يمكننا تعريف متغير وإسناد قيمة له ثم تغيير قيمته بإعادة إسناد قيمة أخرى له ما بسهولة من خلال استخدام معامل الإسناد =.

> لاحظ أننا نستخدِم المعامِل = : عند التهيئة والتصريح عن متغير والمعامِل = عند إعادة الإسناد (تعديل قيمة المتغير)، وبالتالي لا يمكنك استخدام المعامل = : بدلًا من = عند تغيير قيمة المتغير.

بما أنّ لغة جو هي لغة تحتاج إلى تحديد النوع typed language، فعند إسناد قيمة جديدة إلى متغير يجب أن تكون هذه القيمة من نوع بيانات المتغير، فلا يمكن مثلًا إسناد قيمة تُمثّل عددًا صحيحًا إلى متغير من نوع سلسلة نصيّة أو العكس، فإذا حاولت فعل ذلك:

```
i := 72
i = "Sammy"
```

فستحصل على الخرج التالي الذي يُمثّل خطأً، والذي يخبرك أن هذه العملية غير مسموحة كما أوضحنا:

```
cannot use "Sammy" (type string) as type int in assignment
```

لا يمكنك أيضًا تعريف أكثر من متغير واحد بالاسم نفسه، فعندما تكتب:

```
var s string
var s string
```

ستحصل على خطأ كما يلي:

```
s redeclared in this block
```

لا يمكنك استخدام المعامل := بدلًا من = عند محاولة تعديل قيمة متغير كما أشرنا سابقًا كما يلي:

```
i := 5
i := 10
```

فالخرج التالي يُظهر خطأً مفاده أنه لا يوجد متغير جديد على الطرف الأيسر، إذ تُفسّر جو وجـود المعامِـل := بأننا نريد التصريح عن متغير جديد أو تهيئته بقيمة أولية.

```
no new variables on left side of :=
```

ستحسّن تسمية المتغيرات الخاصة بك بطريقة سليمة، واتباع قواعد البرمجة المختلفـة الأخـرى من قابليـة قراءة برنامجك سواءً لك أو للآخرين ولاسيما عند العودة إليه مستقبلًا.

5.6 الإسناد المتعدد

يمكنك في لغة جو إسناد عدة قيم إلى عدة متغيرات في الوقت نفسه، إذ يتيح لك هذا تهيئة عدة متغـيرات دفعةً واحدةً، ويمكن أن تكون كل من هذه المتغيرات من أنواع بيانات مختلفة كما يلي:

```
j, k, l := "shark", 2.05, 15
fmt.Println(j)
fmt.Println(k)
fmt.Println(l)
```

يكون الخرج:

```
shark
2.05
15
```

هنا أُسندت السلسلة النصية "shark" إلى المتغير j، والقيمة 2.05 للمتغير k، والقيمة 15 للمتغير l.

إن إسناد قيم لعدة متغيرات في سطر واحد يمكن أن يجعل الشيفرة مختصَرةً ويقلل من عدد الأسطر، ولكن تأكد من أنّ ذلك ليس على حساب قابلية القراءة.

5.7 المتغيرات العامة والمحلية

عند استخدام المتغيرات داخل البرنامج، من المهم أن تضع نطاق المتغير variable scope في حساباتك، والذي يشير إلى المواضع التي يمكن الوصول منها إلى المتغير داخل الشيفرة، حيث لا يمكن الوصول إلى جميـع المتغيرات من جميع أجزاء البرنامج، فبعض المتغيرات عامة وبعضها محلي، إذ تُعـرّف المتغـيرات العامـة خـارج الدوال؛ أما المتغيرات المحلية، فتوجد داخل الدوال، ويعطي المثال التالي فكرةً عن المتغيرات العامة والمحلية:

```
package main
import "fmt"
var g = "global"
```

```
func printLocal() {
  l := "local"
  fmt.Println(l)
}
func main() {
  printLocal()
  fmt.Println(g)
}
```

يكون الخرج:

```
local
global
```

استخدمنا هنا `var g = "global"` للتصريح عن متغير عام خارج الدالة ثم عرّفنا الدالة `printLocal()` وبداخلها المتغير المحلي `l` المسنَد إليه قيمة، ثم تعليمة لطباعته. واستدعينا في الدالة الرئيسية الدالة `printLocal()` ثم طبعنا قيمة المتغير العام `g`، وبما أنّ `g` متغير عام، فيمكننا الوصول إليه من داخل الدالة `printLocal()` مباشرةً كما في المثال التالي:

```
package main
import "fmt"
var g = "global"
func printLocal() {
  l := "local"
  fmt.Println(l)
  fmt.Println(g)
}
func main() {
  printLocal()
  fmt.Println(g)
}
```

يكون الخرج كالتالي:

```
local
global
global
```

يختلف هذا المثال عن المثال السابق في أننا نحاول الوصول إلى المتغير العام g مباشـرةً من داخـل الدالـة printLocal وهذا ما يُفسّره خرج البرنامج، حيث تطبع الدالة printLocal قيمـة كـل من l و g هـذه المـرة وليس فقط l.

إذا حاولت الوصول إلى قيمة المتغير المحلي من خارج الدالة، فستفشل وستظهر رسالة خطأ تشير لذلك:

```
package main
import "fmt"
var g = "global"
func printLocal() {
  l := "local"
  fmt.Println(l)
}
func main() {
  fmt.Println(l)
}
```

يكون الخرج:

```
undefined: l
```

أي أن المتغير غير مُعرّف في نطـاق الدالـة الرئيسـية، إذ لا يمكنـك الوصـول إلى المتغـير المحلي إلا ضـمن النطاق الذي عُرّف ضمنه ونطاقه في هذا البرنـامج هـو الدالـة printLocal، فلنلـق الآن نظـرةً على مثـال آخـر بحيث نستخدِم اسم المتغير نفسه لمتغير عام ومتغير محلي كما يلي:

```
package main
import "fmt"
var num1 = 5
func printNumbers() {
  num1 := 10
  num2 := 7
  fmt.Println(num1)
  fmt.Println(num2)
}
func main() {
  printNumbers()
  fmt.Println(num1)
}
```

يكون الخرج:

```
10
7
5
```

صرّحنا في البرنامج السابق عن المتغير num1 مرتين؛ مرة ضمن النطاق العام var num1 = 5 والأخرى ضمن نطاق محلي num1 := 10 داخل الدالة printNumbers.

عندما نطبع num1 من البرنامج الرئيسي -أي داخل الدالة main-، سنرى قيمة 5 مطبوعةً لأن main لا ترى سوى المتغير العام؛ أما عندما نطبع num1 من داخل الدالة printNumbers، فإنه سيرى المتغير المحلي وسوف يطبع القيمة 10، وعلى الرغم من أنّ printNumbers يُنشئ متغيرًا جديدًا num1 ويُسند له قيمة 10، إلا أنه لا يؤثر على المتغير العام وستبقى قيمته 5.

عند التعامل مع المتغيرات، من المهم أن تختار بين استخدام المتغيرات العامة أو المحلية. يُفضل في العادة استخدام المتغيرات المحلية، ولكن إذا وجدت نفسك تستخدم نفس المتغير في عدة دوال، فقد ترغب في جعله عامًا. أما إن كنت تحتاج المتغير داخل دالة أو صنف واحد فقط، فقد يكون الأولى استخدام متغير محلي.

5.8 الثوابت

تشبه الثوابت المتغيرات إلا أنه لا يمكن تعديلها بعد التصريح عنها، وهي مفيدة عندما تريد استخدام قيمة محددة في برنامجك ولا تريد تعديلها أبدًا مثل قيمة pi في الرياضيات التي تأخذ القيمة الثابتة 3.14، أو إذا أردنا التصريح عن معدل الضريبة لنظام عربة التسوق، فيمكننا استخدام ثابت ثم حساب الضريبة في مناطق مختلفة من برنامجنا، فإذا تغير معدل الضريبة في وقت لاحق، فسيتعيّن علينا فقط تغيير هذه القيمة في مكان واحد في برنامجنا؛ أما إذا استخدمنا متغيرًا، فمن الممكن تغيير القيمة عن طريق الخطأ في مكان ما في برنامجنا، مما قد يؤدي إلى حساب غير صحيح.

يمكننا استخدام الصيغة التالية للتصريح عن ثابت:

```
const shark = "Sammy"
fmt.Println(shark)
```

يكون الخرج:

```
Sammy
```

إذا حاولت تعديل قيمة الثابت بعد التصريح عنه، فسنحصل على الخطأ التالي:

```
cannot assign to shark
```

يمكن أن تكون الثوابت من دون نوع untyped، وهذا مفيد عند التعامل مع أعداد مثل بيانات الأعداد الصحيحة، فإذا كان الثابت من النوع untyped فيُحوّل صراحةً، حيث لا تُحوَّل الثوابت التي لديها نوع typed.

```
package main
import "fmt"
const (
  year   = 365
  leapYear = int32(366)
)
func main() {
  hours := 24
  minutes := int32(60)
  fmt.Println(hours * year)
  fmt.Println(minutes * year)
  fmt.Println(minutes * leapYear)
}
```

يكون الخرج:

```
8760
21900
21960
```

عرّفنا المتغير year على أنه ثابت من دون نوع untyped، وبالتالي يمكن استخدامه مع أيّ نوع بيانات آخر؛ أما لو حددنا له نوعًا وليكن int32 كما فعلنا مع الثابت leapYear، فلن نستطيع التعامل معه إلا مع بيانات من النوع نفسه لأنه typed constant.

عندما عرّفنا المتغير hours بدون تحديد نوعه، استنتجت جو تلقائيًا أنه من النوع int لأننا أسندنا القيمة 24 له؛ أما عندما عرّفنا المتغير minutes، فقد حددنا له صراحةً نوع البيانات int32 من خلال التعليمة minutes := int32(60).

دعنا الآن نتصفح كل عملية حسابية وسبب نجاحها:

```
hours * year
```

هنا المتغير hours من النوع الصحيح والثابت year من دون نوع، وبالتالي لإجراء العملية تحوّل جو الثابت year إلى النوع المطلوب؛ أي تحوّله إلى النوع int.

```
minutes * year
```

هنـا المتغـير `minutes` من النـوع `int32` والثـابت `year` دون نـوع، لـذا تحـوّل جـو الثـابت `year` إلى النوع `int32`.

```
minutes * leapYear
```

كل من المتغير `minutes` والثـابت `leapYear` من النـوع `int32`، لـذا لن تحتـاج جـو لأيّ عمليـة تحويـل فكلاهما متوافقان.

إذا حاولت إجراء عملية حسابية مثل الضرب بين نوعين مختلفين أي typed، فلن ينجح الأمر:

```
fmt.Println(hours * leapYear)
```

يكون الخرج كما يلي:

```
invalid operation: hours * leapYear (mismatched types int and int32)
```

هنا `hours` هي من النوع `int` كما تحدثنا والثابت `leapYear` من النـوع `int32`، بمـا أنّ جـو لغـة تعتمـد على تحديد النوع typed language، فهذا يعني أنّ النـوعين `int` و `int32` غـير متـوافقين لإجـراء العمليـات الحسابية، ولحل المشكلة عليك تحويل أحدهما إلى نوع الآخر.

5.9 الخاتمة

لقد مررنا في هذا الفصل على بعض حالات الاستخدام الشـائعة للمتغـيرات في جـو، فـالمتغيرات هي لبنـة مهمة في البرمجة، إذ تُمثِّل حاضـنةً لمختلـف أنـواع البيانـات في جـو، و نسـتطيع من خلالهـا إجـراء العديـد من العمليات وتخزين القيم التي نحتاجها، كما تعرفنا على مفهوم الثوابت التي تستخدم عند الحاجة لاستخدام قيمـة لا يجب أن تتغير خلال تنفيذ البرنامج واستعرضنا عدة أمثلة على التعامل معها.

هل تطمح لبيع منتجاتك الرقمية عبر الإنترنت؟

استثمر مهاراتك التقنية وأطلق منتجًا رقميًا
يحقق لك دخلًا عبر بيعه على متجر بيكاليكا

أطلق منتجك الآن

6. تحويل أنواع البيانات

تُستخدَم أنواع البيانات في لغة جو للإشارة إلى نوع معيَّن من البيانات وتحديد القيم الـتي يمكنـك إسـنادها لذلك النوع والعمليات التي يمكنك إجراؤها عليها، لكن هناك أوقات نحتـاج فيهـا إلى تحويـل القيم من نـوع إلى آخر لمعالجتها بطريقة مختلفة، فقد نحتاج مثلًا إلى وصـل القيم العدديـة بالسلاسـل النصـية أو إظهـار الفواصـل العشرية لأعداد صحيحة، وغالبًا ما تُعامَل البيانات التي يُدخلها المستخدِم في جو على أنها سلاسل نصـية، حـتى إذا تضمنت أعدادًا، وبالتالي يتوجب عليك تحويلها إلى بيانات عددية إذا أردت إدخالها في عمليات حسابية.

تُعَدّ جو لغةً ثابتةً تتطّلب تحديد النـوع statically typed language، لـذا تكـون أنـواع البيانـات مرتبطـةً بالمتغيرات وليس بالقيم، أي في حال كان لديك متغير من النوع `int`، فلا يمكن أن يُسـنَد إليـه متغـير من نـوع آخـر، وسنرشـدك في هـذا الفصـل إلى كيفيـة تحويـل الأعـداد والسلاسـل النصـية، بالإضـافة إلى تقـديم بعض الأمثلة التوضيحية.

6.1 تحويل الأنواع العددية

هناك نوعان رئيسيان من البيانات العددية في جو وهما: الأعداد الصحيحة والأعـداد العشـرية. سـتعمل في بعض الأحيان على شيفرة برمجية كتبها شـخص آخـر، وقـد تحتـاج إلى تحويـل عـدد صـحيح إلى عـدد عشـري أو العكس، أو قد تجد أنك تستخدم عددًا صحيحًا في الوقت الـذي تحتـاج إلى أعـداد عشـرية، لـذلك تتـوفر في جـو توابع مضمّنة تُسهِّل عليك التحويل بين الأنواع العددية.

ا. التحويل بين أنواع الأعداد الصحيحة

تتضمن جو العديد من أنواع البيانات الصحيحة التي يمكنك التبديل بينها حسب الحاجة ومتطلبات الأداء أو لأسباب أخرى، فيُنشئ جو في بعض الأحيان مثلًا قيمًا عدديةً من النوع `int` تلقائيًـا، وبالتـالي في حـال أدخلت

أعدادًا من النوع int64، فلن تتمكن استخدام العدد السابق الذي ولدته جو مع العدد الذي أدخلتـه ضـمن تعبـير رياضي واحد لأنهما من نوعين مختلفين.

افترض أنه لديك متغير من النوع int8 وتحتـاج إلى تحويلـه إلى int32، فيمكنـك إجـراء ذلـك عن طريـق استخدام الدالة int32():

```
var index int8 = 15
var bigIndex int32
bigIndex = int32(index)
fmt.Println(bigIndex)
```

يكون الخرج كما يلي:

```
15
```

عرّفنا متغير index من النوع int8 ومتغير bigIndex من النـوع int32 ثم حوّلنـا نـوع المتغـير index إلى النوع int32 وخزّنا النتيجة في المتغير bigIndex ثم طبعنا قيمته، وللتأكـد من عمليـة التحويـل أو لتفقّـد نوع البيانات لأحد المتغيرات، استخدم العنصر النائب %T مع دالة الطباعة fmt.Printf كما يلي:

```
fmt.Printf("index data type:    %T\n", index)
fmt.Printf("bigIndex data type: %T\n", bigIndex)
```

يكون الخرج كما يلي:

```
index data type: int8
bigIndex data type: int32
```

لاحظ أن الكود يطبع هنا نوع بيانات المتغير لا قيمته.

يمكنك أيضًا التحويل من نوع بيانات كبير إلى نوع أصغر مثل تحويل من int64 إلى int8 كما يلي:

```
var big int64 = 64
var little int8
little = int8(big)
fmt.Println(little)
```

يكون الخرج:

```
64
```

لكن انتبه إلى أنه عند التحويل من نوع أكبر إلى نوع أصغر قد تتجاوز القيمة الأكبر بالقيمة المطلقة التي يمكن لهذا النوع تخزينها، ففي حال تجاوزت القيمة تلك، فستحدث عملية التفاف wraparound كما في المثال التالي:

```
var big int64 = 129
var little = int8(big)
fmt.Println(little)
```

يكون الخرج:

```
-127
```

أكبر قيمة يمكن أن يخزنها متغير من النوع `int8` هي 127 وأصغر قيمة 127-، فإذا تجاوزتها، فسيحدث التفاف، وبما أنّ 129 في المثال السابق أكبر من القيمة العظمى التي يمكن للنوع المحدد تخزينه، فقد حدث التفاف إلى القيمة الأصغر، أي حوّله إلى أصغر قيمة ممكنة فيه.

6.1.2 تحويل الأعداد الصحيحة إلى أعداد عشرية

الأمر مشابه لعمليات التحويل التي أجريناها منذ قليل، إذ سنستخدِم هنا الدالة `float64()` أو `float32()` لإجراء عمليات التحويل:

```
var x int64 = 57
var y float64 = float64(x)
fmt.Printf("%.2f\n", y)
```

يكون الخرج:

```
57.00
```

لاحظ أننا حوّلنا نوع المتغير `x` ذو القيمة الابتدائية `57` من `int64` إلى `float64`، وبالتالي أصبحت `57.00`، أخيرًا استخدمنا العنصر النائب `%.2f` ضمن دالة الطباعة للإشارة إلى عدد عشري مع الإبقاء على رقمين بعد الفاصلة.

وبذلك يمكنك استخدام `float32()` أو `float64()` لتحويل الأعداد العشرية إلى أعداد صحيحة.

6.1.3 تحويل الأعداد العشرية إلى أعداد صحيحة

يمكنك تحويل الأعداد العشرية إلى أعداد صحيحة باستخدام الدوال نفسها التي استخدمناها سابقًا عند التحويل بين أنواع الأعداد الصحيحة، ولكن انتبه إلى أنّ تغير النوع من عشري إلى صحيح سيفقدك الأعداد الموجودة بعد الفاصلة العشرية، وبالتالي قد تؤثر على دقة الحالة أو العملية كما في المثال التالي:

```
var f float64 = 390.8
var i int = int(f)
fmt.Printf("f = %.2f\n", f)
fmt.Printf("i = %d\n", i)
```

يكون الخرج:

```
f = 390.80
i = 390
```

عرّفنا في المثال السابق متغير f من النوع العشري وهيأناه بقيمة 390.8 ثم عرّفنا متغير i من النوع الصحيح وهيّأناه بقيمة المتغيّر f بعد تحويّله إلى النوع الصحيح من خلال الدالة int ثم طبعنا قيمتهما، ولاحظ أنه عندما حوّلنا القيمة 390.8 إلى 390، أُسقطت القيمة الموجودة بعد الفاصلة، وتجدر الملاحظة إلى أنّ لغة جو لا تُقرّب العدد؛ وإنما تهمل كل الأعداد بعد الفاصلة فقط.

ولنأخذ المثال التالي الذي يُعرِّف المتغير b ويعطيه القيمة 125.0 والمتغير c مع القيمة 390.8 ثم يطبعهما على شكل عدد صحيح:

```
b := 125.0
c := 390.8
fmt.Println(int(b))
fmt.Println(int(c))
```

يكون الخرج:

```
125
390
```

عند تحويل عدد عشري إلى عدد صحيح باستعمال int()، تقتص لغة جو الأعداد العشرية وتبقي على العدد الصحيح فقط، وانتبه أننا قلنا "تقتص" أي أنها تزيل الفواصل العشرية دون إجراء عملية تقريب، فلن يُقرَّب العدد 390.8 إلى 391 بل سيصبح بعد التحويل 390.

6.1.4 تحويل الأعداد عبر القسمة

تُسقط جو الفواصل إذا وُجِدت أيضًا عند تقسيم على آخر صحيح:

```
a := 5 / 2
fmt.Println(a)
```

يكون الخرج:

```
2
```

إذا كان أحد الأعداد في عملية القسمة من النوع العشري `float`، فستُعَدّ كل الأعداد عشريةً وكـذلك النـاتج كما يلي:

```
a := 5.0 / 2
fmt.Println(a)
```

وسيكون الخرج عددًا عشريًا:

```
2.5
```

كما تلاحظ لم تُسقَط الأعداد بعد الفاصلة هذه المرة.

6.2 تحويل السلاسل النصية

تُعَدّ السلسلة النصية تسلسلًا مؤلفًا من محرف واحد أو أكثر، ويمكن أن يكون المحرف حرفًا أبجديًا أو عددًا أو رمزًا، وتُعَدّ السلاسل النصية إحـدى الأشـكال الشـائعة من البيانـات في عـالم البرمجـة، وقـد نحتـاج إلى تحويـل السلاسل النصية إلى أعداد أو أعداد إلى سلاسل نصية في كثير من الأحيـان خاصـةً عنـدما نعمـل على البيانـات التي ينشئها المستخدمون.

6.2.1 تحويل الأعداد إلى سلاسل نصية

يمكن إنجاز ذلك من خلال استخدام التابع `strconv.Itoa` من إحدى حزم لغة جو القياسية `strconv` في حزمة جو القياسية. وذلك بتمرير العدد الصحيح أو العشري إلى هذا التابع والذي سيتكفل بإجراء عمليـة التحويـل إلى سلسلة نصية، إذ سنحوّل العدد الصحيح 12 إلى سلسلة نصية في المثال التالي:

```
package main
import (
  "fmt"
  "strconv"
)
func main() {
  a := strconv.Itoa(12)
  fmt.Printf("%q\n", a)
}
```

عرّفنا متغير a يُمثّل سلسلة نصية وأسندنا إليه القيمة المُعـادة من الدالـة strconv.Itoa والـتي سـتعبّر عن العدد 12 بوصفه سلسلةً نصيةً، ثم استخدمنا دالة الطباعة لإظهار نتيجة التحويل، كمـا أننـا اسـتخدمنا معهـا العنصر النائب %q لإظهار السلسلة النصية مع علامتي الاقتباس ويكون الخرج كما يلي:

```
"12"
```

تشير علامات الاقتباس حول العدد 12 إلى أنّ العدد الصحيح قد أصبح سلسلةً نصيةً.

يمكنك البدء في معرفة مدى إمكانيـة تحويـل الأعـداد الصـحيحة إلى سلاسـل نصـية من خلال المتغـيرات، ولنفترض أنك تريد تتبع التقدم اليومي في البرمجة للمستخدِم وتريد إدخال عدد أسطر التعليمات البرمجية الـتي يكتبها في كل مرة، كما تريد عرض هذه الملاحظات للمستخدِم مع طباعة قيم السلسـلة والأعـداد الصـحيحة في الوقت نفسه:

```
package main
import (
  "fmt"
)
func main() {
  user := "Sammy"
  lines := 50
  fmt.Println("Congratulations, " + user + "! You just wrote " + lines
+ " lines of code.")
}
```

سيعطي الخرج الخطأ التالي:

```
invalid operation: ("Congratulations, " + user + "! You just wrote ")
+lines (mismatched types string and int)
```

لا يمكنك وصل السلاسل النصية مع الأعداد الصحيحة في جو، لـذلك يجب عليـك تحويـل المتغـير lines إلى سلسلة:

```
package main
import (
  "fmt"
  "strconv"
)
func main() {
  user := "Sammy"
```

```
    lines := 50
        fmt.Println("Congratulations, " + user + "! You just wrote " +
strconv.Itoa(lines) + " lines of code.")
}
```

يكون الخرج كما يلي:

```
Congratulations, Sammy! You just wrote 50 lines of code.
```

إذا أردت تحويل عدد عشري إلى سلسلة نصية، فالأمر مشابه لما سبق. اسـتخدم التـابع `fmt.Sprint` من الحزمة `fmt` من مكتبة جو القياسية، إذ يعيد هذا التابع السلسلة النصية المكافئة للعدد العشري الذي مررته له:

```
package main
import (
  "fmt"
)
func main() {
  fmt.Println(fmt.Sprint(421.034))
  f := 5524.53
  fmt.Println(fmt.Sprint(f))
```

يكون الخرج كما يلي:

```
421.034
5524.53
```

يمكنك إجراء تجربة وصل عدد حقيقي مع سلسلة نصية بعد تحويله إلى سلسلة كما يلي:

```
package main
import (
  "fmt"
)
func main() {
  f := 5524.53
  fmt.Println("Sammy has " + fmt.Sprint(f) + " points.")
}
```

يكون الخرج:

```
Sammy has 5524.53 points.
```

نجحت عملية التحويل بالفعل والدليل على ذلك أن عملية الوصل نجحت.

6.2.2 تحويل السلاسل النصية إلى أعداد

يمكن استخدام الحزمة `strconv` من مكتبة جو القياسية لإنجاز عملية التحويل من سلاسل نصية إلى أعداد صحيحة وعشرية، إذ تحتاج لهكذا عملية تحويل كثيرًا لا سيما عندما تحتاج إلى إدخال البيانات من المستخدِم، فقد تحتاج مثلًا إلى أن يُدخل المستخدِم عمره والذي تُعده جو سلسلةً نصية لا عددًا، لذا أنت بحاجة إلى تحويله إلى عدد صحيح، فإذا كان العدد المدخل صحيحًا، فاستخدم الدالة `strconv.Atoi`؛ أما إذا كان عشريًا، فاستخدم الدالة `strconv.ParseFloat`.

سنكمل الآن مع المثال السابق نفسه الذي تحدثنا فيه عن المستخدِم الذي يتتبع سطور التعليمات البرمجية المكتوبة كل يوم، فإذا أردت معالجة تلك القيم رياضيًا لتقديم ملاحظات أكثر تشويقًا للمستخدِم، فستحتاج لتحويلها إلى أعداد أولًا:

```
package main
import (
  "fmt"
)
func main() {
  lines_yesterday := "50"
  lines_today := "108"
  lines_more := lines_today - lines_yesterday
  fmt.Println(lines_more) }
```

سيتولد الخطأ التالي عند تنفيذ الشيفرة السابقة:

```
invalid operation: lines_today - lines_yesterday (operator - not
defined on string)
```

المتغيران اللذان تحاول طرحهما مُمثلان على أساس سلاسل نصية، لذا لا يمكن إجراء حسابات عليهما، وبالتالي سنصلح المشكلة من خلال استخدام التابع `strconv.Atoi()` من أجل تحويل قيم المتغيرات إلى أعداد صحيحة:

```
package main
import (
  "fmt"
  "log"
  "strconv"
```

```
)
func main() {
  lines_yesterday := "50"
  lines_today := "108"
  yesterday, err := strconv.Atoi(lines_yesterday)
  if err != nil {
        log.Fatal(err)
  }
  today, err := strconv.Atoi(lines_today)
  if err != nil {
        log.Fatal(err)
  }

  lines_more := today - yesterday
      fmt.Println(lines_more)
}
```

استخدمنا `if` لتجنب توقف البرنامج بسبب حدوث خطأ غير متوقع مثل أن تكون السلسلة التي نرغب بتحويلها ليست عددًا صحيحًا أساسًا أو أنها سلسلة فارغة، ويدلنا على حدوث مثل هذه الأخطاء الدالة `strconv.Atoi` من خلال إعادة القيمة `nil` في حال كانت السلسلة الممررة غير ملائمة لعملية التحويل، وبالتالي في حال حدوث خطأ سندخل إلى تعليمة `if` ثم سنسجّل الخطأ من خلال `log.Fatal` ونخرج من البرنامج.

يكون الخرج:

```
58
```

إذا حاولت تحويل سلسلة لا تمثّل عددًا صحيحًا:

```
package main
import (
  "fmt"
  "strconv"
)
func main() {
  a := "not a number"
  b, err := strconv.Atoi(a)
```

```
  fmt.Println(b)
  fmt.Println(err)
}
```

فستحصل على الخطأ التالي:

```
0
strconv.Atoi: parsing "not a number": invalid syntax
```

لم تُسنَد قيمة إلى المتغير b نظرًا لفشل الدالة strconv.Atoi في إجراء التحويل على الرغم من أنه قد صُرِّح عن b، ولاحظ أنّ قيمة b تساوي 0، وذلك لأنّ جو لديها قيم افتراضية يُشار إليها بالقيم الصفرية في جو، كما توفر strconv.Atoi خطأً يصف سبب فشل تحويل السلسلة النصية أيضًا.

6.2.3 التحويل بين السلاسل النصية والبايتات

تُخزّن السلاسل النصية في جو على هيئة شريحة من البايتات، ويمكنك التحويل من سلسلة نصية إلى شريحة من البايتات أو العكس من خلال الدالتين []byte() و string():

```
package main
import (
  "fmt"
)
func main() {
  a := "my string"
  b := []byte(a)
  c := string(b)
  fmt.Println(a)
  fmt.Println(b)
  fmt.Println(c)
}
```

خزّنتَ هنا سلسلةً في المتغير a ثم حوّلتها إلى شريحة من البايتات b ثم حوّلتها إلى سلسلة c من جديد، وبعد ذلك طبعتَ a و b و c على الشاشة، ويكون الخرج كما يلي:

```
my string
[109 121 32 115 116 114 105 110 103]
my string
```

يمثِّل السطر الأول السلسلة النصية الأصلية، ويمثِّل السطر الثاني السلسلة النصية بعد تحويلها إلى شـريحة من البايتات؛ أما السطر الثالث، فيُظهر أمان عملية التحويل من وإلى شريحة بايتات.

6.3 الخاتمة

وضحنا في هذا الفصل كيفية تحويل العديد من أنواع البيانات الأصلية المهمة في لغة جو إلى أنواع بيانـات أخـرى باسـتخدام التوابـع المُضـمّنة، إذ يـوفِّر لـك تحويـل أنـواع البيانـات في جـو مرونـةً إضـافيةً في مشاريعك البرمجية.

دورة إدارة تطوير المنتجات

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

7. العمليات الحسابية

تُعَدّ الأعداد شائعةً في البرمجة، فهي تُستخدَم لتمثيل أشياء مثل أبعاد حجم الشاشة، والمواقع الجغرافية، وكميات الأموال، وإحداثيات النقاط، ومقدار الوقت الذي مضى في مقاطع الفيديو، وتحديد مواضع الصور ضمن الألعاب، وتمثيل القيم اللونية من خلال ربطها مع الأرقام ...إلخ.

كما يرتبط الأداء الفعال في العديد من البرامج ارتباطًا وثيقًا بكيفية إنجاز العمليات الحسابية، لذا تُعَدّ أمرًا مهمًا عليك إتقانه، كما ستجعل منك براعتك في الرياضيات مبرمجًا أفضل لكنها ليست شرطًا أساسيًا، فإذا لم يكن لديك خلفية في الرياضيات، فحاول التفكير في الرياضيات بوصفها أداةً لتحقيق ما ترغب في تحقيقه وطريقةً لتحسين تفكيرك المنطقي.

سنعمل مع أهم نوعين من الأعداد المستخَدمة في لغة جو وهما الأعداد الصحيحة والعشرية:

- الأعداد الصحيحة: هي أعداد سالبة أو موجبة مثل `1, 0, -1,` ...إلخ.
- الأعداد العشرية هي التي تحتوي فواصل عشرية مثل `9.0, -2.25,` ...إلخ.

7.1 العوامل الرياضية

العامل operator هو رمز يمثل عملية وغالبًا ما تكون عملية رياضية، على سبيل المثال يمثل العامل `+` عملية الجمع الرياضية.

نستعرض في الجدول التالي أهم العوامل المستخدَمة في لغة جو والتي سنتطرق لها في هذا الفصل، ويمكنك ملاحظة أنّ أغلب هذه العوامل مألوفة لديك من خلال دراستك للرياضيات وبعضها غير مألوف.

سنشرح أيضًا العوامل المركبة التي تكون إحدى مركباتها عامل حسابي مع إشارة المساواة مثل += و =*.

العملية	الوظيفة
x + y	جمع العددين x و y
x - y	طرح العددين x و y
x * y	ضرب العددين x و y
x / y	قسمة العددين x و y
x % y	باقي قسمة x على y
+x	ضرب قيمة المتغير بالعدد 1+
-x	ضرب قيمة المتغير بالعدد 1-

7.2 الجمع والطرح

يمكنك إنجاز عمليات الجمع والطرح في لغة جو تمامًا مثل استخدامك للآلة الحاسبة كما في المثال التالي من أجل الأعداد الصحيحة:

```
fmt.Println(1 + 5)
```

يكون الخرج كما يلي:

```
6
```

يمكنك تهيئة متغيرات تُخزّن تلك القيم، ثم تطبع نتيجة جمعها بدل تمرير الأعداد مباشرةً إلى دالة الطباعة:

```
a := 88
b := 103
fmt.Println(a + b)
```

يكون الخرج كما يلي:

```
191
```

قد يكون العدد الصحيح سالبًا، ففي هذه الحالة نضع إشارة سالب قبله بكل بساطة كما يلي:

```
c := -36
d := 25
fmt.Println(c + d)
```

يكون الخرج كما يلي:

```
-11
```

الأمر نفسه بالنسبة للأعداد العشرية، إذ يمكنك إنجاز عمليات الجمع كما في المثال التالي:

```
e := 5.5
f := 2.5
fmt.Println(e + f)
```

يكون الخرج كما يلي:

```
8
```

لاحظ أن الناتج هو عدد بدون فاصلة عشرية، فالمتوقع أنّ الخرج يحتوي على فاصلة عشرية بما أنّ عملية الجمع بين عددين عشريين أي 8.0، والسبب في ذلك هو أن دالة الطباعة تحوّل الخرج تلقائيًا إلى قيمة صحيحة من خلال إسقاط الفواصل العشرية، ولتجنب حدوث ذلك نستخدِم العنصر النائب `%.2f`، إذ تشير `f` إلى وجود عدد حقيقي و`2` إلى إبقاء عددين بعد الفاصلة.

```
fmt.Printf("%.2f", e + f)
```

يكون الخرج كما يلي:

```
8.00
```

عملية الطرح نفسها كما في الجمع مع استبدال العامل `+` بالعامل `-`:

```
g := 75.67
h := 32.0
fmt.Println(g - h)
```

يكون الخرج كما يلي:

```
43.67
```

لا يمكنك في لغة جو إنجاز عملية حسابية بين عددين من نوعي بيانات مختلفين مثل `int` و `float64` كما في المثال التالي:

```
i := 7
j := 7.0
fmt.Println(i + j)
```

سيكون الخرج إشعارًا بعدم تطابق نوعي البيانات كما يلي:

```
i + j (mismatched types int and float64)
```

7.3 العمليات الحسابية الأحادية

يتكون التعبير الرياضي الأحادي Unary Arithmetic Operation من مكوّن أو عنصر واحد فقط، إذ يمكننا في لغة جو استخدام علامتَي الجمع والطرح على أساس عنصر أحادي مُقترن بقيمة، ولتبسيط الأمر، تكافئ إشارة + أو – التي تتلوها قيمة أو متغير عملية ضرب 1- أو 1+ بهذه القيمة أو المتغير.

ستوضّح لك الأمثلة هذا الأمر بصورة أفضل:

تشير علامة الجمع -على الرغم من عدم استخدامها بصورة شائعة- إلى هوية القيمة identity، ويمكننا استخدامها مع القيم الموجبة كما يلي:

```
i := 3.3
fmt.Println(+i)
```

سيكون الخرج كما يلي:

```
3.3
```

وعند استخدام إشارة الجمع مع عدد سالب، فلن تتغير إشارة هذا العدد، أي سيبقى سالبًا كما يلي:

```
j := -19
fmt.Println(+j)
```

يكون الخرج كما يلي:

```
-19
```

هنا المتغير موجب، وقد سبقناه في دالة الطباعة بالإشارة – لذا ستتبدّل إشارته من موجب إلى سالب:

```
k := 3.3
fmt.Println(-k)
```

يكون الخرج:

```
-3.3
```

وبصورة بديلة عند استخدام العامل الأحادي سالب الإشارة مع قيمة سالبة كما يلي:

```
j := -19
```

```
fmt.Println(-j)
```

سيكون الخرج كما يلي:

```
19
```

الهـدف من اسـتخدام العمليـة الأحاديـة + والعمليـة الأحاديـة - هـو إعـادة هويـة القيمـة أو الإشـارة المعاكسة لقيمة.

7.4 الضرب والقسمة

كما فعلنا في عمليات الجمع والطرح، لكن نستخدِم هنا عوامل مختلفة هي * للضـرب و / للقسـمة، وفيمـا يلي مثال عن عملية الضرب بين عددين عشريين:

```
k := 100.2
l := 10.2
fmt.Println(k * l)
```

يكون الخرج كما يلي:

```
1022.04
```

بالنسبة للقسمة يجب عليك معرفة أنّ الناتج قد يختلف تبعًا لنـوع البيانـات المختلفـة، ففي حـال قسّـمت أعدادًا صحيحةً، فسيكون ناتج القسمة عددًا صحيحًا أيضًا، إذ ستُسقط أية فواصل عشرية إذا وجدت.

سنقسّم في المثال التالي عددين صحيحين 80/6 كما يلي:

```
package main
import (
  "fmt"
)
func main() {
  m := 80
  n := 6
  fmt.Println(m / n)
}
```

سيكون الخرج كما يلي:

```
13
```

لاحظ إسقاط الفاصلة العشرية من الخرج، فإذا أردت إبقاء الأعداد بعد الفاصلة، فعليك استخدام النوع العشري مع هذه البيانات عبر تحويلها باستخدام `float32()` أو `float64()`:

```
package main
import (
  "fmt"
)
func main() {
  s := 80
  t := 6
  r := float64(s) / float64(t)
  fmt.Println(r)
```

سيكون الخرج:

```
13.333333333333334
```

7.5 باقي القسمة

يمكنك إيجاد باقي قسمة عدد على عدد آخر من خلال العامل %، وهذا مفيد جدًا في الكثير من الحالات مثل إيجاد مضاعفات الأعداد، فلنلق نظرةً على المثال التالي:

```
o := 85
p := 15
fmt.Println(o % p)
```

سيكون الخرج:

```
10
```

إن ناتج قسمة 85 على 15 هو 55 والباقي 10، لذا سيُخرج البرنامج العدد 10، وإذا أردت إيجاد باقي القسمة في حالة الأعداد العشرية، فيمكنك استخدام الدالة `Mod()` من الحزمة `math` كما يلي:

```
package main
import (
  "fmt"
  "math"
)
func main() {
```

```
    q := 36.0
    r := 8.0
    s := math.Mod(q, r)
    fmt.Println(s)
}
```

سيكون الخرج:

```
4
```

7.6 ترتيب العمليات الحسابية

كما في الرياضيات، علينا أن نضع في الحسبان أن العوامل تُطبّـق في البرمجـة حسـب الأولويـة وليس من اليسار إلى اليمين أو من اليمين إلى اليسار، فإذا نظرت إلى التعبير الرياضي التالي

```
u = 10 + 10 * 5
```

هنا يُطبق الضرب أولًا ثم ناتج الضرب 50 يُجمع مع 10 لأن الأولويـة للضـرب على الجمـع أو الطـرح ويكـون الخرج كما يلي:

```
60
```

استخدم الأقواس إذا أردت مثلًا جمع 10 مع 10 أولًا، فما بين الأقواس له الأولوية على ما هو خارجها:

```
u := (10 + 10) * 5
fmt.Println(u)
```

سيكون الخرج كما يلي:

```
100
```

يكون ترتيب إجراء العمليات الحسابية كالتالي:

الأقـواس Parentheses >الرفـع لأس Exponent > الضـرب Multiplication > القسـمة Division > الجمع Addition > الطرح Subtraction.

7.7 عوامل الإسناد

من المألوف لديك استخدام عامل المساواة = لإسناد قيمة إلى متغير، مثل إسناد القيمة 10 إلى المتغير x من خلال كتـابـة التعليمة x = 10، كمـا يمكنـك جعـل الأمر أكثر إثارةً من خلال إدخال العمليات الحسابية إلى

هذه المعادلة، إذ يمكنك استخدام العامل + أو - أو * أو / مع العامل السابق لتطبيق عمليات حسابية محددة، فإذا كان لديك مثلًا متغير x قيمته 5 وأردت جعل قيمته 7، فيمكنك أن تكتب x += 2 وهذا يكافئ كتابة x = x+2، وبالتالي تصبح قيمة x تساوي 7، أو يمكنك ضرب قيمة x بالقيمة 3 من خلال كتابة x *= 3 وهذا يكافئ x = x*3 أو يمكنك استخدام عامل باقي القسمة أيضًا x %= 2 والذي يكافئ x = x%2 وهكذا بالنسبة لباقي العمليات.

ويوضّح المثال التالي الأمر بصورة أفضل:

```
w := 5
w += 1
fmt.Println(w)
```

يكون الخرج:

```
6
```

تُستخدَم عوامل الإسناد المركبة كثيرًا في حالة حلقات for والتي ستستخدِمها عندما تريد تكرار عملية عدة مرات كما في المثال التالي:

```
package main
import "fmt"
func main() {
  values := []int{0, 1, 2, 3, 4, 5, 6}
  for _, x := range values {
        w := x
        w *= 2
        fmt.Println(w)
  }
}
```

يكون الخرج:

```
0
2
4
6
8
10
```

```
12
```

استخدمنا حلقة `for` في المثال أعلاه للتكرار على الشريحة `values`، إذ هيّأنا المتغير `w` في كل تكرار بقيمة `x` الحالية ثم ضربناها بالقيمة 2 وطبعنا الخرج.

تمتلك لغة جو عامل إسناد مركب لكل عامل رياضي، وقد ناقشنا ذلك في هذا الفصل، فلإضافة 1 إلى المتغير نفسه ثم إسناد القيمة الجديدة للمتغير نفسه، نكتب ما يلي والتي تكافئ `y=y+1`:

```
y += 1
```

طرح 1 ثم إسناد:

```
y -= 1
```

ضرب بالعدد 2 ثم إسناد:

```
y *= 2
```

قسمة على العدد 3 ثم إسناد:

```
y /= 3
```

باقي قسمة المتغير على العدد 3 ثم إسناد:

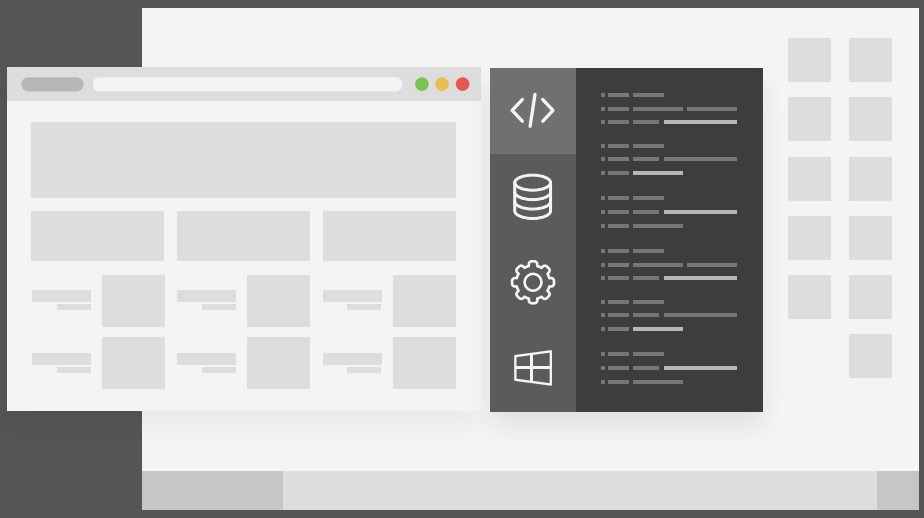
```
y %= 3
```

تُعَدّ عوامل الإسناد المركبة مفيدةً عندما تحتاج إلى زيادة أو تقليل قيمة المتغيرات تدريجيًا أو عندما تحتاج إلى أتمتة عمليات معينة في برنامجك.

7.8 الخاتمة

ناقشنا في هذا الفصل المزيد حول البيانات العددية وشرحنا وأنواعها المختلفة، كما وضحنا العديد من العوامل التي يمكنك استخدامها في لغة جو مع أنواع البيانات العددية الصحيحة والعشرية.

دورة علوم الحاسوب



دورة تدريبية متكاملة تضعك على بوابة الاحتراف
في تعلم أساسيات البرمجة وعلوم الحاسوب

التحق بالدورة الآن

8. البيانات المنطقية Boolean

سنكتشف في هذا الفصل المزيد حول نوع البيانات المنطقي أو البوليانيّ `bool` الذي سُمِّيَ بهذا الاسم نسبة إلى العالم George Boole والذي يتضمن قيمتين فقط هما صح `true` و خطأ `false` أو 1 و 0، ويُستخدم هذا النوع في البرمجة بصورة كبيرة في عمليات المقارنة والتحكم بسير عمل البرنامج.

يُمثَّل نوع البيانات هذا في لغة جو بالقيمة `false` والقيمة `true` مع ملاحظة أن أول حرف لابد أن يكون صغيرًا، وفي الفقرات التالية سنلقي الضوء على أهم الأساسيات التي ستحتاج إليها لفهم كيفية عمل نوع البيانات المنطقية بما في ذلك المقارنة المنطقية والعوامل المنطقية وجداول الحقيقة.

8.1 عوامل المقارنة

تُستخدَم عوامل المقارنة لمقارنة القيم وتقييّمها للحصول على قيمة منطقية واحدة إما صحيحة أو خاطئة، ويوضح الجدول أدناه عوامل المقارنة المنطقية:

العامل	توضيح
==	المساواة
=!	عدم المساواة
<	أصغر
>	أكبر
<=	أكبر أو يساوي
>=	أصغر أو يساوي

سنستخدِم العددين الصحيحين التاليين لتوضيح آلية عمل هذه المتغيرات:

```
x := 5
y := 8
```

من الواضح أنّ قيمة x أصغر من y، وسنستخدِم كل العوامل التي ذكرناها في الجدول السابق وسنرى خـرج البرنامج مع كل حالة في الشيفرة التالية:

```
package main
import "fmt"
func main() {
  x := 5
  y := 8
  fmt.Println("x == y:", x == y)
  fmt.Println("x != y:", x != y)
  fmt.Println("x < y:", x < y)
  fmt.Println("x > y:", x > y)
  fmt.Println("x <= y:", x <= y)
  fmt.Println("x >= y:", x >= y)
}
```

سيكون الخرج كما يلي:

```
x == y: false
x != y: true
x < y: true
x > y: false
x <= y: true
x >= y: false
```

سألنا جو في الشيفرة أعلاه ما يلي:

- هل x تساوي y؟ والإجابة كانت false طبعًا.
- هل x لا تساوي y؟ والإجابة true.
- هل x أصغر من y؟ والإجابة true.
- هل x أكبر من y؟ والإجابة false.
- هل x أصغر أو يساوي y؟ والإجابة true.
- هل x أكبر أو يساوي y؟ والإجابة false.

استخدَمنا الأعداد الصحيحة هنا، لكن يمكنك استخدام الأعداد العشرية أيضًا، كما يمكن أيضًا استخدام السلاسل النصية مع العوامل المنطقية، وتجدر الإشارة إلى أن مقارنة السلاسل حساسة لحالة المحارف، فالمحرف s لا يساوي المحرف S كما في المثال التالي:

```
Sammy := "Sammy"
sammy := "sammy"
alsoSammy := "Sammy"
fmt.Println("Sammy == sammy: ", Sammy == sammy)
fmt.Println("Sammy == alsoSammy", Sammy == alsoSammy)
```

يكون الخرج كما يلي:

```
Sammy == sammy: false
Sammy == alsoSammy true
```

لاحظ أنه عندما قارنا كلمة Sammy بكلمة sammy كان الخرج `false`؛ أما عندما قارنّا كلمة Sammy مع Sammy كان الخرج `true`، وبالتالي كما ذكرنا تكون مقارنة السلاسل النصية حساسةً لحالة المحارف.

يمكنك أيضًا استخدام عوامل المقارنة الأخرى، مثل `>` و `<` لمقارنة السلاسل، وفي هذه الحالة ستقارن جو هذه السلاسل اعتمادًا على الترتيب المُعجمي lexicographically اعتمادًا على قيم ASCII للمحارف، كما يمكنك أيضًا تقييم القيم المنطقية باستخدام عوامل المقارنة، إذ سنختبر في المثال التالي فيما إذا كان المتغير المنطقي `t` الذي يحمل القيمة `true` لا يساوي المتغير المنطقي `f` الذي يحمل القيمة `false`.

```
t := true
f := false
fmt.Println("t != f: ", t != f)
```

يكون الخرج:

```
t != f: true
```

انتبه إلى أن العامِل `=` يختلف عن العامِل `==`، إذ يُستخدَم الأول لعمليات الإسناد والثاني للاختبار المساواة، ويوضح المثال التالي الفرق، ففي الحالة الأولى نُسند قيمة `y` إلى `x` وفي السطر التالي نختبر مساواة `x` مع `y`.

```
x = y  // إسناد قيمة المتغير اليميني إلى المتغير اليساري
x == y // تقييم نتيجة المساواة بين المتغيرين
```

8.2 العوامل المنطقية

يوجد نوعان من العوامل المنطقية في الجبر المنطقي والمستخدَمان لتحديد العلاقة بين المتغيرات هما and و or، إضافةً إلى عامل النفي not، إذ يرمَز في لغة جو للعامل المنطقي and بالرمز && وللعامل or بالرمز || وللعامل not بالرمز ! وهي تعمل كما يلي:

- `x && y` تعني إذا كان كل من `x` و `y` مُحققان أي `true`، يكون الناتج `true` وإلا يكون الناتج `false`.
- `x || y` تعني أنه إذا كان أحدهما `true`، فسيكون الناتج `true`.
- `!x` تعني أنه إذا كانت x قيمتها `true`، فسيكون الخرج `false`، وإذا كانت `false` فسيكون الخرج `true`.

كثيرًا ما تُستخدَم هذه العوامل في البرمجة ولا سيما عند محاولة اختبار تعابير متعددة أو شروط متعددة، فإذا كنت مثلًا تبني برنامجًا يُحدِّد فيما إذا كان الطالب ناجحًا أم لا في إحدى المواد الجامعية، فيجب أن يكون الطالب قد حضر 4 جلسات عمليّة على الأقل `x`، ويجب أن يحصل على 40% من درجة الامتحان العملي على الأقل `y` و60% من درجة الامتحان (النظري + العملي) على الأقل `z`، أي لدينا 3 متغيرات وهي `x y z` ويجب أن تتحقق كلها، أي `x && y && z`، ولنرى مثالًا عن استخدام هذه العوامل كما يلي:

```
fmt.Println((9 > 7) && (2 < 4))  // التعبيران محققان
fmt.Println((8 == 8) || (6 != 6)) // التعبير الأول صحيح فقط
fmt.Println(!(3 <= 1))           // التعبير خاطئ، لكن مع النفي يصبح صحيح
```

يكون الخرج كما يلي:

```
true
true
true
```

لدينا في السطر الأول `fmt.Println((9 > 7) && (2 < 4))`، لاحظ أنّ 9 أكبر من 7 وأنّ 2 أصغر من 4، وبالتالي فإن طرفَي العلاقة محققان، وبالتالي سيطبع `true`، وفي السطر الثاني `fmt.Println((8 == 8) || (6 != 6))` نلاحظ أنّ 8 تساوي 8 مُحققة، لكن 6 لا تساوي 6 غير مُحققة ولكن مع ذلك سيطبع `true`، لأنه في حالة العامل المنطقي `||` يكفي أن يكون أحد طرفَي العلاقة محققًا؛ أما في السطر الأخير `fmt.Println(!(3 <= 1))`، فنلاحظ أنّ 3>=1 تقييمها `false`، لكن بما أننا نستخدِم معامل النفي، فسيكون الخرج هو الحالة المعاكسة، أي `true`.

لا يختلف المثال التالي عما سبق إلا في نوع البيانات، إذ سنستخدِم نوع البيانات العشرية بدل الصحيحة:

```
fmt.Println((-0.2 > 1.4) && (0.8 < 3.1)) // تعبير واحد فقط صحيح
fmt.Println((7.5 == 8.9) || (9.2 != 9.2)) // التعبيران خاطئان
fmt.Println(!(-5.7 <= 0.3)) // التعبير صحيح ولكن يصبح مع النفي خاطئًا
```

يمكنك أيضًا كتابة تعليمات مركبة باستخدام && و || و !:

```
!((-0.2 > 1.4) && ((0.8 < 3.1) || (0.1 == 0.1)))
```

يكون الخرج كما يلي:

```
true
```

نلاحظ أن نتيجة التركيب `(0.1 == 0.1) || (3.1 > 0.8)` هي `true` فكلا الطـرفَين محققَين، لكن نلاحظ أنّ الطرف `(1.4 < 0.2-)` غير محقق، أي تقييمه `false`، وبالتالي سيكون خرج التركيب السابق بأكمله هو `false`، وبما أنه لدينا إشارة النفي، فسيكون الخرج هو `true`.

أي باختصار، العوامل المنطقية && و || و ! تقيِّم التعبيرات، ثم تعيد قيمة منطقية ناتجة عن تقييمها.

8.3 جداول الحقيقة

لتعزيز قدراتك البرمجية لا بدّ أن يكون لديك القليـل من الفهم عن الفـرع المنطقي للرياضـيات، وفيمـا يلي جداول الحقيقة لـعامل المقـارنة == ولكـل من العوامـل المنطقيـة && و || و !. ويفضـل أن تحفظهـا، لتصـبح راسخة في ذهنك إذ ستسرع من عملية اتخاذ القرار البرمجي لديك:

- جدول الحقيقة لعامِل المقارنة ==:

x	==	y	الخرج
true	==	true	true
true	==	false	false
false	==	true	false
false	==	false	true

- جدول الحقيقة للعامِل المنطقي and أو &&:

x	and	y	الخرج
true	and	true	true
true	and	false	false
false	and	true	false
false	and	false	false

- جدول الحقيقة للعامِل المنطقي or أو ||:

x	or	y	الخرج
true	or	true	true
true	or	false	true
false	or	true	true
false	or	false	false

- جدول الحقيقة للعامِل المنطقي not أو !:

not	x	الخرج
not	false	true
not	true	false

8.4 استخدام العوامل المنطقية للتحكم بسير عمل البرنامج

يمكنك استخدام العوامل المنطقية أو عوامل المقارنة للتحكم بسير عمل البرنامج من خلال استخدامها مع جمل الشرط if ... else، فعند تحقق شرط ما يتخذ البرنامج إجراءً محددًا استجابةً لذلك عند تلك النقطة، أي إذا تحقق هذا افعل هذا وإلا لا تفعل وتابع فيما كنت تفعله، ويوضح المثال التالي هذا الأمر:

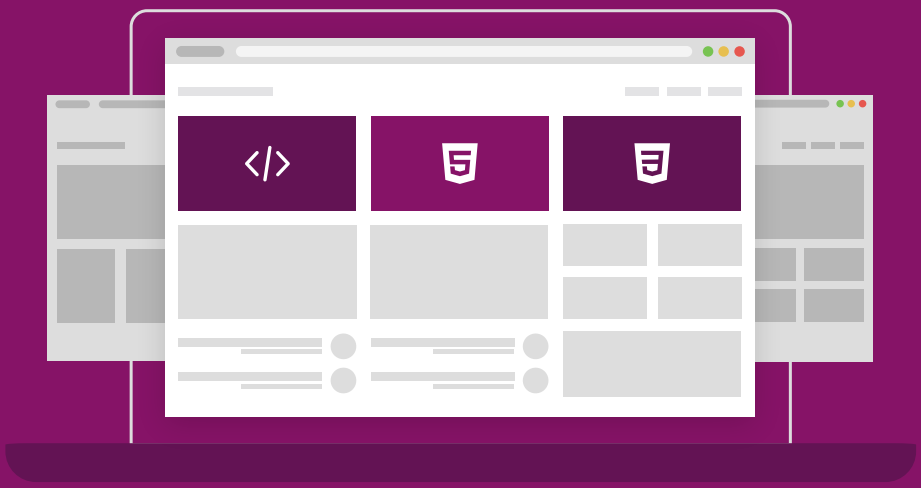
```
if grade >= 65 { // شرط Condition
   fmt.Println("Passing grade") // بند Clause
} else {
   fmt.Println("Failing grade")
}
```

المقصود من الشيفرة أنه إذا كانت درجة الطالب أكبر أوتساوي 65، فاطبع أنه ناجح، وإلا اطبع أنه راسب، أي إذا كانت درجة الطالب 83 مثلًا، فسيطبع Passing grade وإذا كانت 59 سيطبع Failing grade.

8.5 الخاتمة

استعرضنا في هذا الفصل العوامِل المنطقية وعوامل المقارنة التي تُستخدم في الجبر المنطقي وفي العديد من التطبيقات في مجال البرمجة، كما تعرّفنا أيضًا على جداول الحقيقة وأظهرنا كيفية استخدام العوامل المنطقية وعوامل المقارنة في التحكم في سير عمل البرنامج.

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

9. التعرف على الخرائط Maps

تمتلك معظم لغات البرمجة نوع بيانات يُمثّل بنية معطيات يُعرف بالقاموس dictionary أو التجزئة hash، إذ تربط هذه البنية أو هذا النوع البيانات على صورة أزواج (مفتاح:قيمة).

تُعَدّ الخرائط -أو الروابط- Maps مفيدةً للغاية في كثير من الحالات مثل أن تريد إعطاء كل طالب رقمًا يُميّزه عن بقية الطلاب، إذ يمكنك في هذه الحالة تعريف خريطة أو رابط يربط اسم كل طالب (قيمة) برقم يُميّزه (مفتاح)، فالخرائط في لغة جو هي بنية معطيات تكافئ الخرائط maps في لغات أخرى مثل بايثون، وتُعرّف هذه البنية في لغة جو من خلال كتابة الكلمة المفتاحية `map` متبوعةً بنوع بيانات المفاتيح (في مثالنا السابق سيكون أعدادًا صحيحة) ضمن قوسين `[ ]` ثم نوع بيانات القيم (سلاسل نصية في مثالنا)، وأخيرًا توضع أزواج (مفتاح:قيمة) ضمن قوسين معقوصين `{}`على هذا النحو:

```
map[key]value{}
```

سيوضّح المثال التالي الأمر أكثر؛ إذ سنعرّف خريطة map مفاتيحها هي الاسم `name` والحيوان `animal` واللون `color` والموقع `location`، وقيمها هي `Sammy` و `shark` و `blue` و `ocean` على التوالي.

```
map[string]string{"name": "Sammy", "animal": "shark", "color": "blue",
"location": "ocean"}
```

لاحظ أنّ المفتاح يُوضَع على اليسار والقيمة على اليمين ويُفصل بينهما بنقطتين `:` كما يُفصل بين كل زوج (مفتاح:قيمة) بفاصلة `,` ولاحظ أيضًا أننا استخدمنا نوع البيانات `string` لكل من القيم والمفاتيح، وهذا ليس إجباريًا، لكنه يعتمد على ما ستستخدِمه من بيانات.

يمكنك تخزين الخرائط ضمن متغيرات كما في أيّ نوع بيانات آخر:

```
sammy := map[string]string{"name": "Sammy", "animal": "shark",
"color": "blue", "location": "ocean"}
fmt.Println(sammy)
```

يكون الخرج كما يلي:

```
map[animal:shark color:blue location:ocean name:Sammy]
```

لاحظ أنه قد حدث تغيير في مواقع الأزواج عند طباعة الخريطة، وسبب ذلك هـو أنّ جـو تُخـزّن هـذه الأزواج ضمن مناطق متفرقة من الذاكرة، فالترتيب هنا غير مهم، وما يهمنا فقط هو أن تكون كل قيمة مرتبطـةً بمفتـاح نصل إليها من خلاله.

9.1 الوصول إلى عناصر الخريطة

يمكننا الوصول إلى قيم محدَّدة في الخريطة بالرجوع إلى المفاتيح المرتبطة بها، فإذا أردنا مثلًا الحصول على اسم المستخدِم من الخريطة `sammy`، فيمكننا ذلك عن طريق كتابة `sammy["name"]` كما يلي:

```
fmt.Println(sammy["name"])
```

يكون الخرج:

```
Sammy
```

تتصرف الخريطة مثل قاعدة البيانـات، لكن بـدلًا من فهرسـة العناصـر بأعـداد صـحيحة كمـا هـو الحـال في الشرائح slice، فإنها تسند لمفتاح، ويمكنك عبر تلـك المفـاتيح الحصـول على القيم المقابلـة لهـا، أي باسـتدعاء المفتاح `"name"`، سنحصل على القيمة المرتبطة بـه وهي `"Sammy"`، وبالمِثـل، فيمكن اسـتدعاء القيم الأخـرى في الخريطة `sammy` باستخدام الصيغة ذاتها:

```
fmt.Println(sammy["animal"])
//shark تُعيد
fmt.Println(sammy["color"])
//blue تُعيد
fmt.Println(sammy["location"])
//ocean تُعيد
```

من خلال تخزين البيانات على هيئة أزواج (مفتاح:قيمة)، سيصبح بإمكانك الوصول إلى القيمة الـتي تريـدها من خلال ذكر اسم المفتاح الذي يُميّزه.

9.2 المفاتيح والقيم Keys and Values

لا تمتلك لغة جو أيّة دوال مُضمّنة تمكّنك من سرد جميع المفـاتيح أو القيم، ففي بـايثون مثلًا هنـاك الدالـة `.kyes()` التي تعرض لك جميع المفاتيح التي تتضمنها الخريطة أو الدالة `.values()` التي تعرض كل قيمها، لكن يمكنك التكرار على عناصر الخريطة في جو من خلال العامِل `range` كما يلي:

```
for key, value := range sammy {
  fmt.Printf("%q is the key for the value %q\n", key, value)
}
```

ستُعيد `range` هنا قيمتين من أجل كل زوج في الخريطة، إذ سـتُمثِّل القيمـة الأولى المفتـاح `key` والثانيـة القيمة `value`، كما أنّ نـوع بيانـات `key` سـيكون نفسـه نـوع بيانـات `key` في الخريطـة `sammy` والأمـر نفسـه بالنسبة لمتغير القيمة `value`، وبالتالي سيكون الخرج كما يلي:

```
"animal" is the key for the value "shark"
"color" is the key for the value "blue"
"location" is the key for the value "ocean"
"name" is the key for the value "Sammy"
```

صحيح أنه لا توجد دوال جاهزة لاستخراج المفاتيح، لكن يمكنك اسـتخراجها من خلال كتابـة شـيفرة بسـيطة كما يلي:

```
keys := []string{}
for key := range sammy {
  keys = append(keys, key)
}
fmt.Printf("%q", keys)
```

وبالتالي سنحصل على شريحة بجميع المفاتيح:

```
["color" "location" "name" "animal"]
```

كما ذكرنا فالمفاتيح غير مرتبة، ولكن يمكنك ترتيبها إذا أردت من خلال الدالة `sort.Strings()`:

```
sort.Strings(keys)
```

يكون الخرج كما يلي:

```
["animal" "color" "location" "name"]
```

يمكنك إجراء الأمر نفسه لاستخراج القيم، لكن هنا سنعرّف شريحةً يكون عدد عناصرها هو عدد الأزواج الموجودة في الخريطة وذلك لكي لا نضطر في كل مرة إلى طلب حجز مساحة إضافية كما فعلنا في المثال السابق حين عرّفنا شريحةً فارغةً ثم في كل تكرار وسّعنا حجمها بإضافة عنصر جديد.

```
sammy := map[string]string{"name": "Sammy", "animal": "shark",
"color": "blue", "location": "ocean"}
items := make([]string, len(sammy))
var i int
for _, v := range sammy {
  items[i] = v
  i++
}
fmt.Printf("%q", items)
```

إذًا عرّفنا شريحةً بالحجم المطلوب تمامًا بحيث يقابل عدد عناصر الخريطة (كل عنصر يمثّل زوج)، ثم نُعرّف متغير الفهرسة `i` وندخل في حلقة التكرار على عناصر الخريطة، ولاحظ أنه وضعنا عامِل `_` في بداية الحلقة إشارةً إلى أن المفتاح لا نريده، ويكون الخرج كما يلي:

```
["ocean" "Sammy" "shark" "blue"]
```

يمكنك استخدام الدالة المضمّنة `len` لمعرفة عدد العناصر الموجودة في الخريطة كما يلي:

```
sammy := map[string]string{"name": "Sammy", "animal": "shark",
"color": "blue", "location": "ocean"}
fmt.Println(len(sammy))
```

يكون الخرج:

```
4
```

9.3 تفقد وجود عنصر في الخريطة

عند محاولة الوصول إلى عنصر غير موجود في القائمة، ستُعيد جو `0` إشارةً إلى أن هذا العنصر غير موجود في الخريطة، ولهذا السبب ستحتاج إلى طريقة بديلة للتمييز بين الصفر المخزن والمفتاح المفقود، لذا انظر إلى المثال التالي، إذ سنعرّف خريطة فارغة ثم سنحاول الوصول إلى عنصر غير موجود:

```
counts := map[string]int{}
fmt.Println(counts["sammy"])
```

نظرًا لأن المفتاح `sammy` غير موجود، فستعيد جـو `0` كمـا ذكرنـا، فهـو يعطي قيمـة صـفرية تلقائيًـا لجميـع المتغيرات التي ليس لديها قيمة:

```
0
```

هذا أمر غير مرغوب فيه وقد يؤدي إلى حدوث خلل في برنامجـك في الكثـير من الحـالات، ولحـل المشـكلة هناك قيمة اختيارية ثانية يمكنك الحصول عليها عند البحث عن عنصر في الخريطة، وهي قيمـة منطقيـة `bool` تُدعى `ok` وتكون قيمتها `true` إذا عُثِر على العنصر أو `false` إذا لم يُعثر على العنصر، وتستخدَم كما يلي:

```
count, ok := counts["sammy"]
```

يمكنك إذا أحببت استخدام هذه القيمة المُعادة `ok` لطباعة رسالة تُرشدك فيما إذا كان العنصر موجودًا أم لا:

```
if ok {
  fmt.Printf("Sammy has a count of %d\n", count)
} else {
  fmt.Println("Sammy was not found")
}
```

يكون الخرج:

```
Sammy was not found
```

يمكنك في جو الجمع بين التصـريح عن متغـير والفحص الشـرطي مـع كتلـة `if/else`، إذ يـتيح لـك هـذا اختصار التعليمات البرمجية لإنجاز عملية الفحص:

```
if count, ok := counts["sammy"]; ok {
  fmt.Printf("Sammy has a count of %d\n", count)
} else {
  fmt.Println("Sammy was not found")
}
```

عند محاولة الوصول لعنصر ضمن خريطة مـا في جـو، من الأفضـل التحقـق من وجودهـا لتجنب العلَّات أو الأخطاء البرمجية في برنامجك.

9.4 تعديل الخريطة

تُعَدّ الخرائط maps هياكل بيانات قابلة للتغيير mutable، أي يمكن تعديلها، وسنتعلم في هذا القسم كيفية إضافة عناصر إلى خريطة وكيفية حذفها

ا. إضافة وتغيير عناصر خريطة

يمكنك إضافة أزواج قيمة-مفتاح إلى خريطة دون استخدام توابع أو دوال باستخدام الصياغة التالية، حيث نضع اسم متغير الخريطة متبوعةً بالمفتاح بين قوسين `[]` متبوعة بالعامل `=` ثم القيمة:

```
map[key] = value
```

سنضيف في المثال التالي زوج مفتاح-قيمة إلى خريطة تُسمى `usernames`:

```
usernames := map[string]string{"Sammy": "sammy-shark", "Jamie":
"mantisshrimp54"}
usernames["Drew"] = "squidly"
fmt.Println(usernames)
```

يكون الخرج كما يلي:

```
map[Drew:squidly Jamie:mantisshrimp54 Sammy:sammy-shark]
```

لاحظ أنّ الخريطة حُدّثت بالزوج `Drew:squidly` نظرًا لأنّ الروابط غير مرتبة، فيمكن أن يظهر الزوج المُضاف في أيّ مكان في مخرجات الخريطة، فإذا استخدمنا الخريطة `usernames` لاحقًا في ملف برنامجك، فسيظهر فيه الزوج المضاف حديثًا.ويمكن استخدام هذه الصياغة لتعديل القيمة المرتبطة بمفتاح معيّن، في هذه الحالة سنشير إلى مفتاح موجود سلفًا ونمرِّر قيمةً مختلفةً إليه.

سنعرِّف في المثال التالي خريطة باسم `followers` لتعقّب متابِعي المستخدِمين على شبكة معيّنة، إذ حصل المستخدِم `"drew"` على عدد من المتابعين الإضافيين اليوم، لذلك سنحدّث القيمة المرتبطة بالمفتاح `"drew"` ثم سنستخدِم الدالة `Println()` للتحقق من تعديل الخريطة:

```
followers := map[string]int{"drew": 305, "mary": 428, "cindy": 918}
followers["drew"] = 342
fmt.Println(followers)
```

يكون الخرج كما يلي:

```
map[cindy:918 drew:342 mary:428]
```

نرى في الخرج السابق أنّ عدد المتابعين قد قفز من 305 إلى 342.

يمكننا استخدام هذه الطريقة لإضافة أزواج قيمة-مفتاح إلى الخريطة عبر مدخلات المستخدِم، إذًا سنكتب برنامجًا سريعًا `usernames.go` يعمل من سطر الأوامر ويسمح للمستخدِم بإضافة الأسماء وأسماء المستخدِمين المرتبطة بها:

```
package main
import (
  "fmt"
  "strings"
)
func main() {
  usernames := map[string]string{"Sammy": "sammy-shark", "Jamie":
"mantisshrimp54"}
  for {
        fmt.Println("Enter a name:")
        var name string
        _, err := fmt.Scanln(&name)
        if err != nil {
              panic(err)
        }
        name = strings.TrimSpace(name)
        if u, ok := usernames[name]; ok {
              fmt.Printf("%q is the username of %q\n", u, name)
              continue
        }
        fmt.Printf("I don't have %v's username, what is it?\n", name)
        var username string
        _, err = fmt.Scanln(&username)
        if err != nil {
              panic(err)
        }
        username = strings.TrimSpace(username)
        usernames[name] = username
        fmt.Println("Data updated.")
  }
}
```

نُعرّف أولًا الخريطة ضمن ملف usernames.go، ثم نضبط حلقة التكرار على الأسماء، وتطلب بعد ذلك من المستخدِم إدخال اسم وتصرّح عن متغير لتخزّنه فيه، ويجب أن تتحقق بعد ذلك من حدوث خطأ في عملية الإدخال من المستخدِم، ففي حال حدث خطأ، فستخرج من البرنامج.

تلتقط الدالة `Scanln` المُدخلات التي يدخلها المستخدِم بالكامل بما في ذلك محرف العودة إلى بداية السطر Carriage return، لذا أنت بحاجة إلى إزالة أي فراغ زائد من الدخل باستخدام الدالة `strings.TrimSpace`.

تتحقق كتلة `if` من وجود الاسم في الخريطة أو لا، فإذا لم يكن الاسم موجودًا، فسيطبع ملاحظات للمستخدِم تُشير إلى ذلك ثم يطلب إدخال اسم جديد، ويتحقق البرنامج مرةً أخرى من وجود خطأ، ففي حال لم يكن هناك خطأ، فسيَقص محرف الإرجاع ويُسند الاسم المُدخل من المستخدِم على أساس قيمة لمفتاح الاسم ثم يطبع الملاحظات التي تفيد بأن البيانات قد حُدّثت.

سننفِّذ البرنامج من سطر الأوامر:

```
$ go run usernames.go
```

سترى الخرج التالي:

```
Enter a name:
Sammy
"sammy-shark" is the username of "Sammy"
Enter a name:
Jesse
I don't have Jesse's username, what is it?
JOctopus
Data updated.
Enter a name:
```

اضغط على Ctrl+C عند الانتهاء من اختبار البرنامج للخروج من البرنامج.

يوضح هذا المثال كيف يمكنك تعديل الخرائط بآلية تفاعلية، وستفقد جميع بياناتك بمجرد خروجك من هذا البرنامج باستخدام Ctrl+C، إلا إذا خزّنت البيانات في ملف بطريقةٍ ما.

للتلخيص، يمكنك إضافة عناصر إلى الخريطة أو التعديل عليها من خلال الصيغة `map[key] = value`.

ب. حذف عناصر من الخريطة

مثلما يمكنك إضافة أزواج قيمة-مفتاح إلى الخريطة أو تغيير قيمة، يمكنك أيضًا حذف العناصر الموجودة في الخريطة، فلإزالة زوج قيمة-مفتاح من الخريطة، استخدم الدالة `delete()` التي تحتوي على وسيطين، إذ يمثّل الوسيط الأول الخريطة والثاني المفتاح:

```
delete(map, key)
```

لتكن لدينا الخريطة التالية التي تُعبّر عن الأذونات:

```
permissions := map[int]string{1: "read", 2: "write", 4: "delete", 8:
"create", 16:"modify"}
```

بفرض أنك لم تعُد بحاجة إلى إذن التعديل، ستُزيله من الخريطة، ثم ستطبع الخريطة لتأكيد إزالتها:

```
permissions := map[int]string{1: "read", 2: "write", 4: "delete", 8:
"create", 16: "modify"}
delete(permissions, 16)
fmt.Println(permissions)
```

يكون الخرج كما يلي:

```
map[1:read 2:write 4:delete 8:create]
```

يزيل السطر `delete(permissions, 16)` القيمة المرتبطة بالمفتاح 16 ضمن خريطة الأذونات ثم يُزيل المفتاح نفسه.

إذا كنت ترغب في مسح الخريطة بجميع قيمها، فيمكنك إجراء ذلك عن طريق إسناد خريطة فارغة من النوع نفسه لها كما في المثال التالي:

```
permissions = map[int]string{}
fmt.Println(permissions)
```

سيُظهر الخرج أنّ الخريطة قد أصبحت فارغةً:

```
map[]
```

فبما أن الخريطة نمط بيانات قابل للتعديل mutable data type فيمكن التعديل عليها بالإضافة والحذف.

9.5 الخاتمة

ألقينا في هذا الفصل نظرةً على الخرائط أو الروابط maps في لغة البرمجة جو، وهي بـنى معطيـات تتـألف من أزواج (قيمة:مفتاح)، وتوفر حلًا ممتازًا لتخزين البيانات دون الحاجة إلى فهرستها، وتتيح بذلك اسـترداد القيم بناءً على معانيها وعلاقتها بأنواع البيانات الأخرى.

خمسات

لبيع وشراء الخدمات المصغرة

أكبر سوق عربي لبيع وشراء الخدمات المصغرة
اعرض خدماتك أو احصل على ما تريد بأسعار تبدأ من 5$ فقط

تصفح الخدمات

10. المصفوفات Arrays والشرائح Slices

تُعَدّ المصفوفات Arrays والشرائح Slices في لغة جو أنواع بيانات تتألف من تسلسل مرتب من العناصر، وهي ذات فائدة كبيرة عندما تعمل مع مجموعة من القيم التي ترتبط مع بعضها البعض بطريقة ما، كما أنها تسمح لك بالاحتفاظ بالبيانات المتشابهة أو ذات الصلة ببعضها معًا وتنفيذ العديد من العمليات عليها كلها أو على مجموعات جزئية منها ودفعةً واحدةً.

تتسم المصفوفات في لغة جو بأنها ثابتة الحجم، أي لن يكون بإمكانك تغيير ذلك بعد تحديد حجمها لأول مرة، فإذا صرّحت مثلًا عن مصفوفة a بعدد عناصر x، فستخصِّص جو لها ذاكرةً تتسع لعدد العناصر المُحدَّد x، وبالتالي لا يمكنك لاحقًا الإضافة إلى هذه المصفوفة أكثر من x عنصر، فصحيح أنّ ذلك يُفقدها المرونة، إلا أنه يجعلها أسرع.

بينما تتسم الشرائح في جو بكونها ديناميكيةً، مما يُتيح لك تغيير حجمها باستمرار أينما دعت الحاجة لذلك، وتعطي الخاصية الديناميكية المرونة العالية إلا أن ذلك يكون على حساب الأداء أو السرعة، وتُشبه الشرائح إلى حد كبير مفهوم القوائم Lists في لغة البرمجة بايثون والمصفوفات في أغلب اللغات الأخرى، ويمكنك في العديد من المواقف استخدام المصفوفات أو الشرائح، لكن في الحالات التي تحتاج فيها إلى زيادة أو تقليل عدد العناصر، فلا بد من استخدام الشرائح، لأنه في حالة المصفوفات لن تكون قادرًا على ذلك.

سيغطّي هذا الفصل المصفوفات والشرائح بالتفصيل، كما سيزودك بالمعلومات الضرورية لاتخاذ القرار المناسب عند الاختيار بين أنواع البيانات هذه، وستراجع أيضًا الطرق الشائعة للتصريح والعمل مع كل من المصفوفات والشرائح.

10.1 المصفوفات

تُعَدّ المصفوفة بنية معطيات تجميعية تتألف من مجموعة من العناصر من النوع نفسه، وكل عنصر في المصفوفة يمكنه تخزين قيمة واحدة فيه.

تتميز عناصر المصفوفة عن بعضها من خلال عدد محدد يعطى لكل عنصر ويسمى فهرسًا index، بحيث يكون فهرس أول عنصر في المصفوفة 0 دائمًا، كما تُحدَّد سعة المصفوفة لحظة إنشائها، وبمجرد تعريف حجمها لا يمكن تغييره، وبما أنّ حجم المصفوفة ثابت، فهذا يعني أنه يخصص الذاكرة مرةً واحدةً فقط، مما يجعل المصفوفات غير مرنة نوعًا ما للعمل معها، لكنه يزيد من أداء برنامجك، لهذا السبب تُستخدَم المصفوفات عادةً عند تحسين البرامج.

ا. تعريف المصفوفات

تُعرّف المصفوفات من خلال التصريح عن حجمها ضمن قوسين مربعين `[ ]` ثم نوع البيانات ثم القيم المحددة بين الأقواس المعقوصة `{ }`.

```
[capacity]data_type{element_values}
```

مثال:

```
[3]string{"blue coral", "staghorn coral", "pillar coral"}
```

> تمثِّل كل مصفوفة تُعرّفها نوع بيانات مختلف بحد ذاته اعتمادًا على نوع البيانات data_type والحجم capacity، فالمصفوفة السابقة مثلًا من نوع `string` وحجمها 3، تختلف عن مصفوفة من نوع `string` وحجمها 2.

عندما تُصرّح عن مصفوفة بدون تهيئتها بقيم أوليّة، فستُعدّ لغة جو أنّ قيم جميع العناصر أصفار في حالة الأعداد وسلاسل نصية فارغة `''` في حالة السلاسل، فالمصفوفة التالية مثلًا لم نحدد لها قيمًا أوليّةً، وبالتالي ستهيئها لغة جو بأصفار على أساس قيم افتراضية:

```
var numbers [3]int
```

يكون الخرج كما يلي:

```
[0 0 0]
```

يمكنك أيضًا إسناد قيم عناصر المصفوفة عند إنشائها بوضعها ضمن قوسين معقوصين `{ }` بالشكل التالي:

```
coral := [4]string{"blue coral", "staghorn coral", "pillar coral",
"elkhorn coral"}
```

```
fmt.Println(coral)
```

خزنا المصفوفة ضمن متغير ثم طبعنا قيمتها ويكون الخرج كما يلي:

```
[blue coral staghorn coral pillar coral elkhorn coral]
```

لاحظ أنه لا يوجد فصل واضح بين العناصر المطبوعة، لذا يُفضَّل استخدام الدالة `fmt.Printf()` مع العنصر النائب `%q` لكي توضَع علامتَي اقتباس مزدوجة لكل عنصر لتحديده:

```
fmt.Printf("%q\n", coral)
```

يكون الخرج كما يلي:

```
["blue coral" "staghorn coral" "pillar coral" "elkhorn coral"]
```

وضعنا الرمز `\n` للنزول سطر بعد طباعة المصفوفة.

ب. فهرسة المصفوفات والشرائح

يمكن استدعاء عنصر محدد من المصفوفة أو الشريحة من خلال الفهرسة، إذ يمتلك كل عنصر فهرسًا والذي قيمته `int` تبدأ من الصفر وتزداد تصاعديًا.

سنستخدِم مصفوفةً في الأمثلة التالية، ولكن يمكنك أيضًا استخدام شريحة، إذ ستكون الفهرسة متطابقةً في كل منهما، فمن أجل المصفوفة `coral` ستكون الفهرسة كما يلي:

"blue coral"	"staghorn coral"	"pillar cora"	"elkhorn coral"
0	1	2	3

لاحظ أنّ العنصر الأول "blue coral" يحمل الفهرس رقم 0 ويحمل العنصر الأخير "elkhorn coral" الفهرس رقم 3.

بما أن كل عنصر يمتلك فهرسًا خاصًا به، يمكننا الوصول إلى العنصر الذي نريده من خلال هذا الفهرس وإجراء العمليات عليه، فمثلًا نريد الآن الوصول إلى العنصر الثاني من المصفوفة:

```
fmt.Println(coral[1])
```

سيكون الخرج كما يلي:

```
staghorn coral
```

كما يمكنك بالطريقة ذاتها الوصول إلى أيّ عنصر من المصفوفة:

```
coral[0] = "blue coral"
coral[1] = "staghorn coral"
coral[2] = "pillar coral"
coral[3] = "elkhorn coral"
```

لاحظ أن هناك 4 عناصر في المصفوفة، وبالتالي تكون الفهارس من 0 إلى 3، لذا لا تحاول وضع فهرس خارج هذه الحدود، فإذا حاولت مثلًا في المصفوفة السابقة الوصول إلى عنصر يحمل الفهرس 4 أو 5 أو 18، فسيظهر لك خطأ لأن هذا العنصر غير موجود أو أنك تتجاوز حدود المصفوفة:

```
fmt.Println(coral[18])
```

يكون الخرج كما يلي:

```
panic: runtime error: index out of range
```

في العديد من لغات البرمجة الأخرى مثل بايثون، يمكنك الوصول للعناصر من خلال الفهرسة السلبية، لكن لا يمكنك ذلك في جو، أي يجب استخدام الأعداد الموجبة للوصول إلى العناصر، وإلا سيظهر لك خطأ كما في المثال التالي:

```
fmt.Println(coral[-1])
```

يكون الخرج كما يلي:

```
invalid array index -1 (index must be non-negative)
```

يمكنك ضم عناصر سلسلة في مصفوفة أو شريحة مع سلاسل أخرى باستخدام العامل +:

```
fmt.Println("Sammy loves " + coral[0])
```

يكون الخرج كما يلي

```
Sammy loves blue coral
```

تمكنا من وصل العنصر الموجود في الفهرس رقم 0 بالسلسلة النصية "Sammy loves"، لأن كل عنصر في المصفوفة هو سلسلة نصية بحد ذاته بما أننا حددنا نوع بيانات المصفوفة على أنه `string` وبالتالي كل عنصر فيها يُمثّل سلسلةً نصيةً.

يمكننا الوصول إلى كل عنصر بصورة منفصلة والعمل مع تلك العناصر باستخدام أعداد الفهرس التي تتوافق مع العناصر داخل مصفوفة أو شريحة ما.

ج. تعديل عناصر المصفوفة

يمكنك تعديل قيمة عنصر محدَّد ضمن المصفوفة بالطريقة نفسها التي تُعدّل فيها قيمة متغير من أيّ نـوع بيانات وذلك بعد الوصول إلى ذلك العنصر من خلال رقم الفهرس الخاص به، ويمنحنا هذا مزيـدًا من التحكم في البيانات الموجودة في الشرائح والمصفوفات، كما سيسمح لنا بمعالجة العناصر الفردية برمجيًا.

```
coral[1] = "foliose coral"
```

سنطبع الآن المصفوفة لنرى التعديل:

```
fmt.Printf("%q\n", coral)
```

يكون الخرج كما يلي:

```
["blue coral" "foliose coral" "pillar coral" "elkhorn coral"]
```

د. معرفة حجم المصفوفة

الآن بعد أن عرفت كيفية التعامل مع العناصر الفرديـة لمصـفوفة أو شـريحة، دعنـا نلقي نظـرةً على دالـتين ستمنحانك مزيدًا من المرونة عند العمل مع بُنى المعطيات هذه. الدالة الأولى هي `len()` وهي دالة مُضمّنة في جو تساعدك على العمل مع المصفوفات والشرائح، بحيث نُمرر لهذه الدالة المصفوفة أو الشريحة وتعيد لنا عـدد عناصرها كما في المثال التالي:

```
len(coral)
```

سيكون الخرج 4 لأن عدد العناصر في المصفوفة هو 4.

مثال آخر، سننشئ مصفوفة أعداد صحيحة ونحسب عدد عناصرها:

```
numbers := [13]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
fmt.Println(len(numbers))
```

يكون الخرج كما يلي:

```
13
```

قد لا نشعر بأهمية استخدام هذه الدالة عندما يكون عدد عناصر المصـفوفة قليلًا كمـا في الأمثلـة السـابقة، لكن عندما نتعامل مع مصفوفات كبيرة، ستظهر الحاجة أكثر لهذه الدالة، كما سـنغطي بعـد ذلـك كيفيـة إضـافة عنصر إلى نوع بيانات المجموعة ونوضِّـح كيـف أنّ إلحـاق أنـواع البيانـات الثابتـة هـذه -بسـبب الطـول الثـابت للمصفوفات- سيؤدي إلى حدوث خطأ.

ه. إضافة عناصر إلى مصفوفة

الدالة الأخرى التي سنتعرّف عليها هي append() وهي خاصة بالشرائح، إذ تضيف هذه الدالة عنصرًا جديدًا إلى الشريحة، ولا يمكن استخدام هذه الدالة مع المصفوفات لأن المصفوفات لا يمكن تغيير حجمها ببساطة كما سبق وذكرنا، فلتكن لدينا المصفوفة التالية على سبيل المثال:

```
coral := [4]string{"blue coral", "foliose coral", "pillar coral",
"elkhorn coral"}
```

إذا حاولت إضافة عنصر جديد black coral إلى هذه المصفوفة من خلال الدالة append():

```
coral = append(coral, "black coral")
```

ستحصل على خطأ:

```
first argument to append must be slice; have [4]string
```

ولحل هذه المشكلة، سنلجأ إلى نوع بيانات آخر وهو الشرائح.

10.2 الشرائح

تُعَدّ الشريحة تسلسلًا مرتبًا من العناصر والتي يمكن أن يتغير طولها (قابلة للتغيير)، حيث يمكن للشرائح أن تزيد حجمها ديناميكيًا، فعند إضافة عناصر جديدة إلى شريحة وإذا لم يكن للشريحة حجم ذاكرة كافي لتخزين العناصر الجديدة، فستطلب ذاكرةً أكبر من النظام حسب الحاجة، ولهذا السبب هي شائعة الاستخدام أكثر من المصفوفات.

تمكّنك الخاصيّة الديناميكية للشرائح من العمل معها في الحالات التي تحتاج فيها إلى تغيير طول عدد عناصر البيانات بأمان، وهذه هي الخاصية الوحيدة التي تتميز بها الشرائح عن المصفوفات.

ا. التصريح عن شريحة

يُصرّح عن الشرائح من خلال تحديد نوع البيانات مسبوقًا بقوسَي فتح وإغلاق مربعَين [] ليس فيهما أي قيمة -خلاف نوع المصفوفة التي تحتاج إلى تحديد حجمها- وقيم بين أقواس معقوصة {}. وتكون شريحة من الأعداد الصحيحة كما يلي:

```
[]int{-3, -2, -1, 0, 1, 2, 3}
```

شريحة من الأعداد الحقيقية:

```
[]float64{3.14, 9.23, 111.11, 312.12, 1.05}
```

شريحة من السلاسل:

```
[]string{"shark", "cuttlefish", "squid", "mantis shrimp"}
```

يمكنك أيضًا إسنادها إلى متغير:

```
seaCreatures := []string{"shark", "cuttlefish", "squid", "mantis
shrimp"}
```

ثم طباعة هذا المتغير:

```
fmt.Println(seaCreatures)
```

يكون الخرج كما يلي:

```
[shark cuttlefish squid mantis shrimp]
```

إليك مثالًا آخر:

```
seaCreatures := []string{"shark", "cuttlefish", "squid", "mantis
shrimp", "anemone"}
```

سنطبع الشريحة لكن سوف نستخدم الدالة `fmt.Printf()` مع العنصر النائب `%q` كما فعلنا مع المصفوفات سابقًا

```
fmt.Printf("%q\n", seaCreatures)
```

يكون الخرج كما يلي:

```
["shark" "cuttlefish" "squid" "mantis shrimp" "anemone"]
```

إذا رغبت في إنشاء شريحة بطول معيّن دون تحديد عناصرها، فيمكنك استخدام الدالة `make()` المضمنة:

```
oceans := make([]string, 3)
```

فإذا طبعناها، فسيكون الخرج كما يلي:

```
["" "" ""]
```

يمكنك أيضًا إنشاء شريحة بطول محدَّد كما فعلنا أعلاه لكن مع حجز مساحة إضافية من خلال تمرير عدد العناصر التي نريد حجز ذاكرة لها بصورة استباقية على أساس وسيط ثالث:

```
oceans := make([]string, 3, 5)
```

المقصود هنا أنه ستُنشأ شريحة بطول 3 لكن ستُحجز ذاكرة لخمس عناصر إضافية، أي ستحجز بالمجمل 8 عناصر، والهدف من ذلك هو تحسين الأداء، أي نتوقع أنه ستُضاف 5 عناصر على الأكثر لاحقًا، فنحجز لهم ذاكرةً مسبقًا لتجنب التأخير الذي سيحدث عند حجز ذاكرة إضافية في وقت تشغيل البرنامج بحيث نستفيد من الخاصية الديناميكية والأداء كما في المصفوفات.

يمكنك استخدام الدالة `append()` التي تحدثنا عنها سابقًا لإضافة عنصر جديد إلى الشريحة.

ب. تقطيع المصفوفات إلى شرائح

تدعم المصفوفات إضافةً إلى الفهرسة عملية الاقتطاع slicing أيضًا، إذ تُستخدم الفهرسة للحصول على عناصر مفردة؛ أما عملية الاقتطاع فتتيح الحصول على جزء من المصفوفة، أي عدة عناصر متتالية، ويمكنك إجراء ذلك عن طريق إنشاء نطاق من أرقام الفهرس مفصولةً بنقطتين [الفهرس الأول: الفهرس الثاني] أو [firstindex:secondindex]، ومن المهم ملاحظة أنه عند تقطيع المصفوفة ستكون النتيجة **شريحةً** وليست مصفوفةً.

لنفترض أنك ترغب فقط في طباعة العناصر الوسطى من المصفوفة `coral` بدون العنصر الأول والأخير، إذ يمكنك إجراء ذلك عن طريق إنشاء شريحة تبدأ من الفهرس 1 وتنتهي قبل الفهرس 3 مباشرةً كما يلي:

```
fmt.Println(coral[1:3])
```

يكون الخرج كما يلي:

```
[foliose coral pillar coral]
```

لاحظ أننا وضعنا الرقم 1 ثم نقطتين ثم 3، بالنسبة للرقم 1 فقد وضعناه لأنه يمثّل نقطة البداية حيث يتواجد ثاني عنصر من المصفوفة؛ أما 3 فهو نقطة النهاية لكنها لا تؤخذ، وإنما تؤخذ القيمة التي قبلها مباشرةً، أي أنه ستؤخذ العناصر من 1 إلى 2، باختصار، العنصر المُحدد بالرقم الأول سيكون مشمولًا دائمًا بعملية الاقتطاع؛ أما العنصر المُحدد بالرقم الأخير فلن يكون مشمولًا.

هناك بعض القيم الافتراضية المفيدة في خاصية التقطيع، فإذا حُذف الفهرس رقم 1 -أي العنصر الثاني-، فستكون القيمة الافتراضية له صفرًا، وإذا حُذف الرقم الثاني secondindex فإنّ القيمة الافتراضية تكون طول السلسلة النصية التي سيجري اقتطاعها.

```
coral[:2]  // اقتطاع العناصر من الموقع 0 إلى الموقع 2 (غير مشمول)
coral[1:]  // اقتطاع العناصر من الموقع 1 (مشمول) إلى نهاية السلسلة
```

نريد مثلًا من أول عنصر حتى العنصر الثاني:

```
fmt.Println(coral[:3])
```

يكون الخرج كما يلي:

```
[blue coral foliose coral pillar coral]
```

أو مثلًا نريد من العنصر الأول حتى النهاية:

```
fmt.Println(coral[1:])
```

يكون الخرج كما يلي:

```
[foliose coral pillar coral elkhorn coral]
```

ج. التحويل من مصفوفة إلى شريحة

في حال احتجت إلى تغيير حجم المصفوفة يمكنك تحويلها إلى شريحة من خلال الآلية التي اتبعناها في الفقرة السابقة لتقطيع المصفوفة، ويمكن إنجاز ذلك من خلال استخدام النقطتين : بدون تحديد البداية أو النهاية، أي كما يلي

```
coral[:]
```

لاحظ أنه هنا لا يُحوّل المُتغير بحد ذاته -والذي يمثّل المصفوفة- إلى شريحة، لكن ما يحدث هو أنه عند استخدامك للتقطيع مع المصفوفات، ستُعيد جو نسخةً تمثّل الجزء الذي اقتطعته على أساس شريحة، أي أنّ المتغير `coral` نفسه لا يحوّل، وإنما تُعاد نسخة منه، وهذا منطقي، ففي جو لا يمكنك تغيير نوع المتغير مهما كان نوعه، إذًا نكتب كما يلي:

```
coralSlice := coral[:]
```

إذًا ستمثّل `coralSlice` نسخة شريحة من المصفوفة `coral`، فإذا طبعناها، فسنحصل على ما يلي:

```
[blue coral foliose coral pillar coral elkhorn coral]
```

يمكنك الآن مثلًا إضافة عنصر جديد وليكن `black coral` إلى الشريحة كما يلي:

```
coralSlice = append(coralSlice, "black coral")
fmt.Printf("%q\n", coralSlice)
```

سنحصل على الخرج التالي عند طباعتها:

```
["blue coral" "foliose coral" "pillar coral" "elkhorn coral" "black
coral"]
```

يمكنك أيضًا إضافة أكثر من عنصر دفعةً واحدةً:

```
coralSlice = append(coralSlice, "antipathes", "leptopsammia")
```

يكون الخرج كما يلي:

```
["blue coral" "foliose coral" "pillar coral" "elkhorn coral" "black
coral" "antipathes" "leptopsammia"]
```

يمكنك دمج شريحتين معًا من خلال `append()` لكن يجب أن تضع ثلاث نقاط `...` بعد اسم الشريحة الثانية كما يلي:

```
moreCoral := []string{"massive coral", "soft coral"}
coralSlice = append(coralSlice, moreCoral...)
```

يكون الخرج كما يلي:

```
["blue coral" "foliose coral" "pillar coral" "elkhorn coral" "black
coral" "antipathes" "leptopsammia" "massive coral" "soft coral"]
```

د. حذف عنصر من شريحة

لا تمتلك جو دوالًا جاهزةً مثل باقي اللغات لحذف عنصر، إذ يجب عليك الاعتماد على مبدأ التقطيع لحذف العناصر، فلحذف عنصر يجب عليك تقطيع الشريحة لشريحتين، بحيث تمثِّل الأولى جميع العناصر التي قبله وتمثِّل الثانية العناصر الذي بعده، ثم دمج هاتين الشريحتين الجديدتين معًا بدون العنصر الذي تريد حذفه، أي كما يلي بفرض أنّ `i` هو فهرس العنصر المراد إزالته:

```
slice = append(slice[:i], slice[i+1:]...)
```

نريد فرضًا حذف العنصر `"elkhorn coral"` الموجود في الفهرس 3 من الشريحة `coralSlice`:

```
coralSlice := []string{"blue coral", "foliose coral", "pillar coral",
"elkhorn coral", "black coral", "antipathes", "leptopsammia", "massive
coral", "soft coral"}
coralSlice = append(coralSlice[:3], coralSlice[4:]...)
fmt.Printf("%q\n", coralSlice)
```

يكون الخرج كما يلي:

```
["blue coral" "foliose coral" "pillar coral" "black coral"
"antipathes" "leptopsammia" "massive coral" "soft coral"]
```

نلاحظ أنّ العنصر الذي حددناه قد حُذف من الشريحة، ويمكنك أيضًا حذف عدة قيم أو مجال من القيم دفعةً واحدةً، إذ تريد مثلًا حذف `"elkhorn coral"` و `"black coral"` و `"antipathes"`، أي تريد حذف العناصر الموجودة في الفهارس 3 و 4 و 5:

```
coralSlice := []string{"blue coral", "foliose coral", "pillar coral",
"elkhorn coral", "black coral", "antipathes", "leptopsammia", "massive
coral", "soft coral"}
coralSlice = append(coralSlice[:3], coralSlice[6:]...)
fmt.Printf("%q\n", coralSlice)
```

يكون الخرج كما يلي:

```
["blue coral" "foliose coral" "pillar coral" "leptopsammia" "massive
coral" "soft coral"]
```

٥. عدد عناصر الشريحة

صحيح أنه يمكنك استخدام التابع `len()` لمعرفة عدد عناصر الشريحة، لكنه سيعطيك عدد العناصر الظاهرة وليس عدد العناصر التي حُجز لها مكان في الذاكرة، وللتوضيح، يجب عليك العودة إلى الدالة `make()` وتتذكر حينما كتبنا:

```
oceans := make([]string, 3, 5)
```

هنا أنشأنا شريحة عدد عناصرها الظاهر 3 لكن حجزنا ذاكرة لخمسة عناصر وقد شرحنا ذلك سابقًا، فإذا استخدمنا الآن الدالة `len()` لمعرفة عدد عناصر الشريحة `oceans`، فستُعيد 3؛ أما إذا استخدمنا الدالة `cap()`، فستعطي 5 وهذا هو الفرق الوحيد بينهما.

لاحظ أن الدالة `cap()` ليس لاستخدامها معنًى مع المصفوفات لأن الحجم ثابت، أي الحجم الظاهر هو نفسه المحجوز دومًا، لذا اقتصر تعريف `cap()` على العمل مع الشرائح ولا تعمل مع المصفوفات.

الاستخدام الشائع للدالة `cap()` هو إنشاء شريحة بعدد محدد مسبقًا من العناصر ثم ملء تلك العناصر برمجيًا، إذ يؤدي ذلك إلى تجنب عمليات التخصيص غير الضرورية المحتملة التي قد تحدث باستخدام `append()` لإضافة عناصر تتجاوز السعة المخصصة حاليًا كما في المثال التالي:

```
numbers := []int{}
for i := 0; i < 4; i++ {
  numbers = append(numbers, i)
}
fmt.Println(numbers)
```

يكون الخرج كما يلي:

```
[0 1 2 3]
```

أنشأنا شريحةً ثم أنشأنا حلقة `for` تتكرر أربع مرات. أضفنا في كل تكرار القيمة الحالية لمتغير الحلقة `i` إلى شريحة الأعداد `numbers`، وقد يؤدي ذلك إلى عمليات تخصيص غير ضرورية للذاكرة والتي قد تؤدي إلى إبطاء برنامجك، فعند إضافة عناصر إلى شريحة فارغة، فإنه في كل مرة تستدعي فيها الدالة `append()` تتحقق جو من سعة الشريحة، فإذا كان العنصر المضاف سيجعل الشريحة تتجاوز هذه السعة، فستُخصص ذاكرة إضافية لها، مما يؤدي إلى إنشاء عبء إضافي في برنامجك ويمكن أن يؤدي إلى إبطاء التنفيذ.

سنملأ الآن الشريحة بدون استخدام `append()` وذلك عن طريق التخصيص المسبق للعناصر كما تعلمت سابقًا في بداية الفصل:

```
numbers := make([]int, 4)
for i := 0; i < cap(numbers); i++ {
  numbers[i] = i
}
fmt.Println(numbers)
```

يكون الخرج كما يلي:

```
[0 1 2 3]
```

استخدمنا هنا الدالة `make()` لإنشاء شريحة وجعلناها تخصص 4 عناصر مسبقًا، ثم استخدمنا بعد ذلك الدالة `cap()` في الحلقة للتكرار على كل عنصر من الشريحة (العناصر تكون كلها أصفار كما أشرنا سابقًا)، ثم ملأنا الشريحة بالقيم التي نريدها وهي قيم متغير الحلقة `i`، ولاحظ أنّ استخدام الدالة `cap()` هنا بدلًا من `append()` سيتجنب أيّ تخصيصات إضافية للذاكرة.

و. الشرائح متعددة الأبعاد

يمكنك أيضًا تعريف الشرائح متعددة الأبعاد، أي شريحة تتضمّن شريحةً أو أكثر؛ يمكن القول شرائح متداخلة، إذ نضع هنا الشرائح ضمن قوسين داخل الشريحة التي تضمها. مثال:

```
seaNames := [][]string{{"shark", "octopus", "squid", "mantis shrimp"},
{"Sammy", "Jesse", "Drew", "Jamie"}}
```

يجب عليك استخدام عدة مؤشرات للوصول إلى عنصر داخل هذه الشريحة، بحيث نستخدِم مؤشرًا واحدًا لكل بُعد من أبعاد البنية:

```
fmt.Println(seaNames[1][0])
fmt.Println(seaNames[0][0])
```

تعني التعليمة `seaNames[1][0]` أنك تريد الوصـول إلى العنصـر الأول 0 من الشـريحة ذات الفهـرس 1؛ أما التعليمة الثانية، فتشير إلى أنك تريد الوصول إلى العنصر الأول 0 من الشريحة ذات الفهرس 0، لـذا سـيكون الخرج كما يلي:

```
Sammy
shark
```

فيما يلي قيم الفهرس لبقية العناصر الفردية:

```
seaNames[0][0] = "shark"
seaNames[0][1] = "octopus"
seaNames[0][2] = "squid"
seaNames[0][3] = "mantis shrimp"
seaNames[1][0] = "Sammy"
seaNames[1][1] = "Jesse"
seaNames[1][2] = "Drew"
seaNames[1][3] = "Jamie"
```

عند العمل مع الشرائح متعددة الأبعاد من المهم أن تضع في الحسبان أنك ستحتاج إلى الإشارة إلى أكثر من رقم فهرس واحد من أجل الوصول إلى عناصر محددة داخل الشريحة المتداخلة التي تعمل عليها.

10.3 الخاتمة

تعلمت في هذا الفصل كيفية العمل مع المصفوفات والشرائح والخرائط في جو، وتعـرّفت على العديـد من حالات الاستخدام لكل منها، كمـا تعـرّفت على بعض الأخطـاء الشـائعة في التعامـل مـع هـذه البُـنى وعلى دوال أساسية فيها، وأصبحت لديك القدرة أيضًا على التفريق بين الشرائح والمصفوفات ومتى تستخدم أيًا منهما.

دورة تطوير التطبيقات
باستخدام لغة بايثون

احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن

11. معالجة الأخطاء

تتصف البرامج الحاسوبية المكتوبة بطريقة متينة بقدرتها التعامل مع الأخطاء المتوقعة وغير المتوقعة الـتي قد تحدث عند استخدام البرنامج، فهناك أخطاء ناتجة عن مدخلات غير صـحيحة من المسـتخدِم أو حـدوث خطـأ في عملية الاتصال بالشبكة ...إلخ.

نقصد بمعالجة الأخطاء Error handling التقاط الأخطـاء الـتي تولّـدها برامجنـا وإخفائهـا عن المسـتخدم، فرسائل الأخطاء لا تخيف المبرمجين وإنما يمكن توقعهـا أحيانًـا، إلا أن المسـتخدِم لا يتوقـع رؤيتهـا وهي تربكـه وتسبب له الحيرة، فإن كان سيرى رسالة خطأ للضرورة، فلتكن رسالةً سهلة الفهم، وحتى في هذه الحالة سيرغب المستخدِم في أن يحل المبرمج المشكلة، وهنا يأتي دور التعامل مع الأخطاء ومعالجتها، إذ توفِّر كـل لغـة تقريبًـا آليةً لالتقاط الأخطاء عند حدوثها لمعرفة الأجزاء التي تعطلت واتخاذ الإجراء المناسب لإصلاح المشكلة.

لمعالجة الأخطاء في لغات البرمجـة الأخـرى، عـادةً مـا يتطلب الأمـر من المُـبرمجين اسـتخدام بنيـة قواعـد محدَّدة، إلا أنّ الأمر مُختلف في جو، إذ تُعَدّ الأخطاء قيمًا مع نوع الخطأ الذي يُعاد من الدالة مثل أيّ قيمة معـادة أخرى، ولمعالجة الأخطاء في جو، يجب عليك فحص هذه الأخطاء الـتي قـد تُعيـدها الـدوال وتحديـد مـا إذا كـان هنـاك خطـأ فعلًا، واتخـاذ الإجـراء المناسـب لحمايـة البيانـات وإخبـار المسـتخدِمين أو أجـزاء البرنـامج الأخـرى بحدوث الخطأ.

11.1 إنشاء الأخطاء

قبل أن تبدأ بمعالجة الأخطاء عليك إنشاؤها أولًا، إذ توفر المكتبة القياسية دالتين مضـمنتين لإنشـاء أخطـاء وهما `errors.New()` و `fmt.Errorf()` بحيث تتيح لك هاتان الدالتان تحديد رسالة خطأ مخصصة يمكنك تقديمها لاحقًا للمستخدِمين.

تأخذ الدالة `errors.New()` وسيطًا واحدًا يمثّل سلسلةً تُمثّل رسالة الخطأ، بحيث يمكنك تخصيصها لتنبيه المستخدِمين بالخطأ، وسنستخدِم في المثال التالي الدالة `errors.New()` لإنشاء خطأ وستكون رسالة الخطأ هي "barnacles" ثم سنطبع هذا الخطأ من خلال الدالة `fmt.Println()`، ولاحظ أننا كتبنا جميع أحرف الرسالة بدون استخدام محارف كبيرة تقيّدًا بالطريقة التي تكتب بها الأخطاء في جو والتي تُكتب بمحارف صغيرة.

```
package main
import (
  "errors"
  "fmt"
)
func main() {
  err := errors.New("barnacles")
  fmt.Println("Sammy says:", err)
}
```

يكون الخرج كما يلي:

```
Sammy says: barnacles
```

تسمح لك الدالة `fmt.Errorf()` بإنشاء رسالة خطأ ديناميكيًا، بحيث يمثّل الوسيط الأول لهذه الدالة سلسلةً تمثّل رسالة الخطأ مع إمكانية استخدام العناصر النائبة مثل العنصر `%s` لينوب عن سلسلة نصية والعنصر `%d` لينوب عن عدد صحيح؛ أما الوسيط الثاني لهذه الدالة، فهو قيم العناصر النائبة بالترتيب كما في المثال التالي:

```
package main
import (
  "fmt"
  "time"
)
func main() {
  err := fmt.Errorf("error occurred at: %v", time.Now())
  fmt.Println("An error happened:", err)
}
```

يكون الخرج كما يلي:

```
An error happened: Error occurred at: 2019-07-11 16:52:42.532621 -0400
EDT m=+0.000137103
```

استخدمنا الدالة `fmt.Errorf()` لإنشاء رسالة خطأ تتضمن التوقيت الزمني الحالي، إذ تتضمن السلسلة المُعطاة إلى الدالة `fmt.Errorf()` العنصر النائب `%v` والذي سيأخذ القيمة `time.Now()` لاحقًا عند استدعاء هذا الخطأ، وأخيرًا نطبع الخطأ من خلال دالة الطباعة كما فعلنا مع الدالة السابقة.

11.2 معالجة الأخطاء

ما سبق كان مُجرّد مثال لتوضيح كيفية إنشاء الأخطاء؛ أما الآن فستتعلم كيفية إنشائها وتوظيفها، فمن الناحية العملية، يكون من الشائع جدًا إنشاء خطأ وإعادته من دالة عندما يحدث خطأ ما، وبالتالي عند استدعاء هذه الدالة يمكن استخدام عبارة `if` لمعرفة ما إذا كان الخطأ موجودًا أم أنه لايوجد خطأ، فعندما لا يكون هناك خطأً سنجعل الدالة تعيد قيمة `nil`.

```
package main
import (
  "errors"
  "fmt"
)
func boom() error {
  return errors.New("barnacles")
}
func main() {
  err := boom()
  if err != nil {
        fmt.Println("An error occurred:", err)
        return
  }
  fmt.Println("Anchors away!")
}
```

يكون الخرج:

```
An error occurred: barnacles
```

عرّفنا في هذا المثال دالة تُسمى `boom()` تُعيد خطأً يُنشأ من خلال الدالة `errors.New` لاحقًا عند استدعاء هذه الدالة والتقاط الخطأ في السطر `err := boom`، فبعد إسناد الخطأ إلى المتغير `err` سنتحقق مما إذا كان موجودًا من خلال التعليمة `if err != nil`، وفي هذا المثال ستكون نتيجة الشرط دومًا `true`

لأننا نُعيد دومًا خطأً من الدالة، وطبعًا لن يكون الأمر دائمًا هكذا، لـذلك يجب أن نحـدد في الدالـة الحـالات الـتي يحدث فيها خطأ والحالات التي لا يحدث فيها خطأ وتكون القيمة المعادة `nil`.

نستخدِم دالة الطباعة `fmt.Println()` لطباعة الخطأ كما في كل مرة عند وجود خطـأ، ثم نسـتخدِم أخـيرًا تعليمة `return` لكي لا تُنفّذ تعليمة `fmt.Println("Anchors away!")`، فهذه التعليمـة يجب أن تُنفّـذ فقط عند عدم وجود خطأ.

> تُعَدّ التعليمة `if err != nil` محورًا أساسيًا في عملية معالجة الأخطاء، فهي تنقل البرنامج إلى مسار مختلف عن مساره الأساسي في حال وجود خطأ أو تتركه يكمل مساره الطبيعي.

تتيح لك جو إمكانية استدعاء الدالة ومعالجة أخطاءها ضمن تعليمة الشرط `if` مباشرةً، ففي المثـال التـالي سنكتب المثال السابق نفسه لكن من خلال الاستفادة من هذه السمة كما يلي:

```
package main
import (
  "errors"
  "fmt"
)
func boom() error {
  return errors.New("barnacles")
}
func main() {
  if err := boom(); err != nil {
        fmt.Println("An error occurred:", err)
        return
  }
  fmt.Println("Anchors away!")
}
```

يكون الخرج كما يلي:

```
An error occurred: barnacles
```

لاحظ أننا لم نغير كثيرًا عن الشيفرة السابقة، فكل ما فعلناه هو أننا استدعينا الدالة التي تُعيد الخطأ واختبرنـا الشرط في السطر نفسه ضمن التعليمة `if`.

تعلمنا في هذا القسم كيفية التعامل مع الدوال التي تعيد الخطأ فقط، وهذه الـدوال شـائعة، لكن من المهم أيضًا أن تكون قادرًا على معالجة الأخطاء من الدوال التي يمكن أن تُعيد قيمًا متعددةً.

11.3 إعادة الأخطاء والقيم

غالبًا ما تُستخدَم الدالات التي تُعيد قيمة خطأ واحدة في الحالات التي نحتاج فيها إلى إحداث تغييرات مُحددة مثل إدراج صفوف في قاعدة بيانات أي عندما نحتاج لمعرفة الحال الذي انتهت عليه العملية، ومن الشائع أيضًا كتابة دالات تُعيد قيمة إذا اكتملت العملية بنجاح أو خطأ مُحتمل إذا فشلت هذه العملية، كما تسمح جو للدالات بإعادة أكثر من نتيجة واحدة، وبالتالي إمكانية إعادة قيمة ونوع الخطأ في الوقت نفسه.

لإنشاء دالة تُعيد أكثر من قيمة واحدة، يجب تحديد أنواع كل قيمة مُعادة داخل أقواس ضمن ترويسة الدالة، فالدالة `capitalize` مثلًا، تُعيد سلسلة `string` وخطأ `error`، وصرّحنا عن ذلك بكتابة شيفرة كتلية:

```
func capitalize(name string) (string, error) {}
```

يخبر الجزء (سلسلة، خطأ) مُصرّف جو أنّ هذه الدالة ستعيد سلسلةً نصيةً وخطأً بهذا الترتيب، ويمكنك تشغيل البرنامج التالي لرؤية الخرج من هذه الدالة التي تُعيد قيمتين وهما سلسلة نصية وخطأ:

```
package main
import (
  "errors"
  "fmt"
  "strings"
)
func capitalize(name string) (string, error) {
  if name == "" {
        return "", errors.New("no name provided")
  }
  return strings.ToTitle(name), nil
}
func main() {
  name, err := capitalize("sammy")
  if err != nil {
        fmt.Println("Could not capitalize:", err)
        return
  }
  fmt.Println("Capitalized name:", name)
}
```

يكون الخرج كما يلي:

```
Capitalized name: SAMMY
```

عرّفنا الدالة `capitalize()` التي تأخذ سلسلة نصية كوسيط وهي السلسلة التي نريد تحويل محارفها إلى محارف كبيرة، وتُعيد سلسلةً نصيةً وقيمة خطأ، ثم استدعينا في الدالة الرئيسية `main()` الدالة `capitalize()` وأسندنا القيم التي تُعيدها إلى المتغيرين `name` و `err` وفصلنا بينهما بفاصلة، ثم استخدمنا التعليمة الشرطية `if err != nil` للتحقق من وجود خطأ لنطبعه في حال وجوده ونخرج من خلال التعليمة `return` وإلا نكمل في المسار الطبيعي ونطبع`fmt.Println("Capitalized name:", name)`.

مرَّرنا في المثال الكلمة `sammy` للدالة `capitalize()`، لكن إذا حاولت تمرير سلسلة فارغة `""`، ستحصل مباشرةً على رسالة الخطأ `Could not capitalize: no name provided`، فعند تمرير سلسلة فارغة، تُعيد الدالة خطأً، وعند تمرير سلسلة عادية، ستستخدِم الدالة `capitalize()` الدالة `strings.ToTitle` لتحويل السلسلة الممرَّرة لمحارف كبيرة وتُعيدها وتعيد `nil` أيضًا للإشارة لعدم وجود خطأ.

هناك بعض الاصطلاحات الدقيقة التي اتبعها هذا المثال والتي تُعَدّ نموذجيةً في شيفرات جو ولكن ليست إجباريةً من قِبَل مصرِّف جو، فعندما تكون لدينا دالة تُعيد عدة قيم مثلًا، يُشاع أن تكون قيمة الخطأ المُعادة هي القيمة الأخيرة، أيضًا عندما تُعاد قيمة خطأ ما، فستُسنَد القيمة الصفرية إلى كل قيمة لا تمثّل خطأ، والقيم الصفرية مثلًا هي القيمة 0 في حالة الأعداد الصحيحة أو السلسلة الفارغة في حالة السلاسل النصية أو السجل الفارغ في حالة نوع البيانات `struct` أو القيمة `nil` في حالة المؤشر والواجهة interface، وقد تحدّثنا عن ذلك سابقًا بالتفصيل في فصل المتغيرات والثوابت.

ا. تقليل استخدام الشيفرة المتداولة

يمكن أن يصبح الالتزام بهذه الاصطلاحات مملًا في حال وجود العديد من القيم المُعادة من دالة، إذ يمكننا استخدام مفهوم الدالة مجهولة الاسم anonymous function لتقليل الشيفرة المتداولةboilerplate.

تُعَدّ الدوال مجهولة الاسم إجرائيات مسنَدَة إلى المتغيرات، على عكس الدوال التي عرَّفناها في الأمثلة السابقة، فهي متوفرة فقط ضمن الدوال التي تُصرّح عنها، وهذا يجعلها مثاليةً لتعمل على أساس أجزاء منطقية قصيرة مُساعدة وقابلة لإعادة الاستخدام.

سنعدّل المثال السابق ونضيف له طول الاسم الذي نريد تحويل حالة المحارف فيه إلى الحالة الكبيرة، وبما أنه لدينا ثلاث قيم تجب إعادتها، فقد يصبح التعامل مع الأخطاء مرهقًا بدون دالة مجهولة الاسم تُساعدنا:

```
package main
import (
  "errors"
```

```
    "fmt"
    "strings"
)
func capitalize(name string) (string, int, error) {
    handle := func(err error) (string, int, error) {
        return "", 0, err
    }
    if name == "" {
        return handle(errors.New("no name provided"))
    }
    return strings.ToTitle(name), len(name), nil
}
func main() {
    name, size, err := capitalize("sammy")
    if err != nil {
        fmt.Println("An error occurred:", err)
    }
    fmt.Printf("Capitalized name: %s, length: %d", name, size)
}
```

يكون الخرج:

```
Capitalized name: SAMMY, length: 5
```

نستقبل 3 وسائط مُعادة من الدالة `capitalize()` داخل الدالة `main()`، وهي `name` و `size` و `err` على التوالي، ثم نختبر بعد ذلك فيما إذا كانت الدالة `capitalize()` قد أعادت خطأً أم لا وذلك من خلال فحص قيمة المتغير `err` إذا كان `nil` أم لا، فمن المهم فعل ذلك قبل محاولة استخدام أيّ من القيم الأخرى المُعادة من الدالة `capitalize` لأن الدالة مجهولة الاسم `handle` يمكن أن تضبطها على قيم صفرية، وفي هذا المثال لم نُمرِّر سلسلةً فارغةً، لذا أكمل البرنامج عمله وفقًا للمسار الطبيعي، ويمكنك تمرير سلسلة فارغة لترى أنّ الخرج سيكون رسالة خطأ `An error occurred: no name provided`.

عرّفنا المتغير `handle` داخل الدالة `capitalize` وأسندنا إليه دالة مجهولة الاسم، أي أنّ هذا المتغير أصبح يُمثّل دالةً مجهولة الاسم، وستأخذ خطأ `error` على أساس وسيط وتُعيد قيمًا تُطابق القيم التي تُعيدها الدالة `capitalize` وبالترتيب نفسه لكن تجعل قيمها صفريّة، كما تُعيد الدالة الخطأ الذي مرِّر لها أيضًا كما هو، وبناءً على ذلك سيكون بإمكاننا إعادة أي أخطاء تحدث في الدالة `capitalize` باستخدام تعليمة `return` يليها استدعاء الدالة مجهولة الاسم `handle` مع تمرير الخطأ على أساس وسيط.

تذكَّر أنّ الدالة `capitalize` يجب أن تُعيد دومًا ثلاث قيم فهكذا عرّفناها، ولكن في بعض الأحيان لا نريد التعامل مع جميع القيم التي يمكن أن تُعيدها الدالة، لذا لحسن الحظ يمكننا التعامل مع هذا الأمر من خلال استخدام الشَرطة السفلية _ كما سترى بعد قليل.

11.4 معالجة الأخطاء في الدوال التي تعيد عدة قيم

عندما تُعيد الدالة العديد من القيم، ينبغي علينا إسناد كل منها إلى متغير وهذا ما فعلناه في المثال السابق مع الدالة `capitalize`، ويجب فصل هذه المتغيرات بفواصل.

لا نحتاج في بعض الأحيان إلا لقيم محددة منها، فقد لا نحتاج مثلًا إلا لقيمة الخطأ `error`، ففي هذه الحالة يمكنك استخدام الشرطة السفلية _ لتجاهل القيم الأُخرى المُعادة، وقد عدّلنا المثال الأول عن الدالة `capitalize()` في المثال التالي ومرّرنا لها سلسلةً فارغةً لكي تُعطينا خطأً واستخدمنا الشرطة السفلية لتجاهل القيمة الأولى التي تُعيدها الدالة كما يلي:

```
package main
import (
  "errors"
  "fmt"
  "strings"
)
func capitalize(name string) (string, error) {
  if name == "" {
        return "", errors.New("no name provided")
  }
  return strings.ToTitle(name), nil
}
func main() {
  _, err := capitalize("")
  if err != nil {
        fmt.Println("Could not capitalize:", err)
        return
  }
  fmt.Println("Success!")
}
```

يكون الخرج كما يلي:

```
Could not capitalize: no name provided
```

استدعينا الدالة `capitalize()` في المثال أعلاه داخل الدالـة الرئيسـية `main()` وأسـندنا القيم المُعـادة منها إلى المتغير _ والمتغير `err` على التوالي، وبذلك نكون قد تجاهلنا القيمة الأولى المُعادة من الدالة واحتفظنا بقيمة الخطأ داخل المتغير `err`، وما تبقى شرحناه سابقًا.

11.5 تعريف أنواع أخطاء جديدة مخصصة

نحتاج في بعض الأوقات إلى تعريف أنواع أكـثر تعقيـدًا من الأخطـاء من خلال تنفيـذ الواجهـة `error`، إذ لا تكفينـا دوال المكتبـة القياسـية `errors.New()` و `fmt.Errorf()` في بعض الأحيـان لالتقـاط مـا حـدث والإبلاغ عنه بالطريقة المناسبة، لذا تكون البنية التي نحتاج إلى تحقيقها كما يلي:

```
type error interface {
 Error() string
}
```

تتضمّن الواجهة `error` تابعًا وحيدًا هو `Error()` والذي يُعيد سلسلةً نصيةً تمثّل رسالة خطأ، فبهذا التابع ستتمكن من تعريف الخطأ بالطريقة التي تناسبك، وفي المثال التالي سننفّذ الواجهة `error` كما يلي:

```
package main
import (
  "fmt"
  "os"
)
type MyError struct{}
func (m *MyError) Error() string {
  return "boom"
}
func sayHello() (string, error) {
  return "", &MyError{}
}
func main() {
  s, err := sayHello()
  if err != nil {
        fmt.Println("unexpected error: err:", err)
        os.Exit(1)
  }
```

```
    fmt.Println("The string:", s)
}
```

سنرى الخرج التالي:

```
unexpected error: err: boom
exit status 1
```

عرّفنا نوع بيانات عبارة عن سجل `struct` فارغ وأسميناه `MyError`، كما عرّفنا التابع `Error()` داخله بحيث يُعيد الرسالة "boom".

نستدعي الدالة `sayHello` داخل الدالة الرئيسية `main()` التي تُعيد سلسلةً فارغةً ونسخةً جديدةً من `MyError`، وبما أنّ `sayHello` ستُعطي خطأً دومًا، فسيُنفَّذ استدعاء `fmt.Println()` الموجود داخل التعليمة الشرطية دومًا وستُطبع رسالة الخطأ.

لاحظ أنه لا نحتاج إلى استدعاء التابع `Error()` مُباشرةً لأن الحزمة `fmt` قادرة تلقائيًا على اكتشاف أنّ هذا تنفيذ للواجهة `error`، وبالتالي يُستدعى التابع `Error()` تلقائيًّا وتُطبع رسالة الخطأ.

11.6 الحصول على معلومات تفصيلية عن خطأ

يكون الخطأ المخصص custom error عادةً هو أفضل طريقة لالتقاط معلومات تفصيلية عن خطأ، فلنفترض مثلًا أننا نريد التقاط رمز الحالة status code عند حدوث أخطاء ناتجة عن طلب HTTP، لذا سننفّذ الواجهة `error` في البرنامج التالي بحيث يمكننا التقاط هكذا معلومات:

```
package main
import (
    "errors"
    "fmt"
    "os"
)
type RequestError struct {
    StatusCode int
    Err error
}
func (r *RequestError) Error() string {

    return fmt.Sprintf("status %d: err %v", r.StatusCode, r.Err)
}
```

```
func doRequest() error {
  return &RequestError{
        StatusCode: 503,
        Err:    errors.New("unavailable"),
  }
}
func main() {
  err := doRequest()
  if err != nil {
        fmt.Println(err)
        os.Exit(1)
  }
  fmt.Println("success!")
}
```

سنرى الخرج التالي:

```
status 503: err unavailable
exit status 1
```

أنشأنا في هذا المثال نسخةً من `RequestError` وزوّدناها برمز الحالة والخطأ باستخدام الدالة `errors.New` من المكتبة القياسية، ثم نستخدِم بعد ذلك الدالة `fmt.Println()` لطباعتها كما هو الحال في الأمثلة السابقة. استخدمنا الدالة `fmt.Sprintf()` داخل التابع `Error()` في `RequestError` لإنشاء سلسلة باستخدام المعلومات المقدَّمة عند إنشاء الخطأ.

11.7 توكيدات النوع والأخطاء المخصصة

تعرض الواجهة `error` دالةً واحدةً فقط، لكن قد نحتاج إلى الوصول إلى دوال أخرى من تنفيذات أخرى للواجهة `error` لمعالجة الخطأ بطريقة مناسبة، فقد يكون لدينا مثلًا العديد من التنفيذات للواجهة `error` والتي تكون مؤقتةً ويمكن إعادة طلبها، إذ يُشار إليها بوجود التابع `Temporary()`.

توفِّر الواجهات رؤيةً ضيقةً لمجموعة أوسع من التوابع التي يمكن للأنواع أن توفرها، لذلك يجب علينا تطبيق عملية توكيد النوع type assertion لتغيير التوابع التي تُعرض أو إزالتها بالكامل، ويوسِّع المثال التالي النوع `RequestError` بتضمينه التابع `Temporary()` والذي سيشير إذا كان يجب على من يستدعي الدالة إعادة محاولة الطلب أم لا:

```
package main
```

```
import (
  "errors"
  "fmt"
  "net/http"
  "os"
)
type RequestError struct {
  StatusCode int
  Err error
}
func (r *RequestError) Error() string {
  return r.Err.Error()
}
func (r *RequestError) Temporary() bool {
  return r.StatusCode == http.StatusServiceUnavailable // 503
}
func doRequest() error {
  return &RequestError{
        StatusCode: 503,
        Err:    errors.New("unavailable"),
  }
}
func main() {
  err := doRequest()
  if err != nil {
        fmt.Println(err)
        re, ok := err.(*RequestError)
        if ok {
              if re.Temporary() {
                    fmt.Println("This request can be tried again")
              } else {
                    fmt.Println("This request cannot be tried again")
              }
        }
        os.Exit(1)
  }
```

```
    fmt.Println("success!")
}
```

يكون الخرج كما يلي:

```
unavailable
This request can be tried again
exit status 1
```

نستدعي الدالة `doRequest()` ضمن الدالة `main` التي تُعيد لنا الواجهة `error`، إذ نطبع أولًا رسالة الخطأ المُعادة من التابع `Error()` ثم نحاول كشف جميع التوابع من `RequestError()` باستخدام توكيد النوع `re, ok := err.(*RequestError)`، فإذا نجح توكيد النوع، فإننا سنستخدِم التابع `Temporary()` لمعرفة ما إذا كان هذا الخطأ خطأً مؤقتًا.

بما أنّ المتغير `StatusCode` الذي هُيئ من خلال الدالة `doRequest()` يحمل القيمة 503 والذي يتطابق مع `http.StatusServiceUnavailable`، فإنّ ذلك يُعيد `true` وبالتالي طباعة "This request can be tried again"، كما يمكننا من الناحية العملية تقديم طلب آخر بدلًا من طباعة رسالة.

11.8 تغليف الأخطاء

سيكون منشأ الخطأ غالبًا خارجيًا أي من خارج برنامجك مثل قاعدة بيانات واتصال بالشبكة ومدخلات مستخدِم غير صحيحة ...إلخ. فرسائل الخطأ المقدَّمة من هذه الأخطاء لا تساعد أيّ شخص في العثور على أصل الخطأ.

سيُقدِّم تغليف الأخطاء بمعلومات إضافية في بداية رسالة الخطأ رؤيةً أفضل (معلومات عن السياق أو الطبيعة التي حدث فيها) لتصحيح الأخطاء بنجاح، ويوضِّح المثال التالي كيف يمكننا إرفاق بعض المعلومات السياقية لخطأ خفي من دالة أخرى:

```
package main
import (
    "errors"
    "fmt"
)
type WrappedError struct {
    Context string
    Err   error
}
```

```
func (w *WrappedError) Error() string {
  return fmt.Sprintf("%s: %v", w.Context, w.Err)
}
func Wrap(err error, info string) *WrappedError {
  return &WrappedError{
        Context: info,
        Err:   err,
  }
}
func main() {
  err := errors.New("boom!")
  err = Wrap(err, "main")
  fmt.Println(err)
}
```

يكون الخرج كما يلي:

```
main: boom!
```

يحتوي السجل `WrappedError` على حقلين هما رسالة عن السياق على هيئة سلسلة نصية والخطأ الذي يقدّم عنه معلومات إضافيةً، فعندما يُستدعى التابع `Error()`، فإننا نستخدِم `fmt.Sprintf()` مرةً أخرى لطباعة رسالة السياق ثم الخطأ، إذ يستدعي `fmt.Sprintf()` التابع `Error()` ضمنيًا.

نستدعي الدالة `errors.New` داخل الدالة `main()` ثم نغلّف الخطأ باستخدام الدالة `Wrap` التي عرّفناها، إذ يسمح لنا ذلك بالإشارة إلى أنّ هذا الخطأ قد أُنشئ في الدالة `main`، وبما أنّ `WrappedError` هي خطأ، لذا يمكننا تغليف العديد منها، إذ يسمح لنا ذلك بالحصول على سلسلة تمكننا من تتبع مصدر الخطأ، كما يمكننا أيضًا تضمين كامل مسار المكدس في الأخطاء التي تحدث مع القليل من المساعدة من المكتبة القياسية.

بما أن الواجهة `error` لا تحتوي إلا تابعًا واحدًا، فسيمنحنا ذلك مرونةً كبيرةً في تقديم أنواع مختلفة من الأخطاء لمواقف مختلفة، ويمكن أن يشمل ذلك كل شيء بدءًا من وصل أجزاء متعددة من المعلومات على أساس جزء من الخطأ الأساسي وصولًا إلى تحقيق التراجع الأسي Exponential backoff.

11.9 معالجة حالات الانهيار في لغة جو Go

تنقسم الأخطاء التي قد تحدث في البرنامج إلى فئتين رئيسيتين هما أخطاء يتوقع المبرمج حدوثها وأخطاء لم يتوقع حدوثها، وتُعالِج الواجهة `error` التي تحدّثنا عنها في الفقرات السابقة إلى حد كبير الأخطاء التي نتوقعها أثناء كتابة البرامج حتى تلك الأخطاء التي تكون احتمالات حدوثها نادرةً.

تندرج حالات الانهيار أو الهلع Panics تحت الفئة الثانية والتي تؤدي إلى إنهاء البرنامج تلقائيًا والخروج منه، وعادةً تكون الأخطاء الشائعة هي سبب حالات الانهيار، لذا سندرس في فقراتنا التالية بعض الحالات الشائعة التي يمكن أن تؤدي إلى حالات الانهيار، كما سنلقي نظرةً على بعض الطرق التي يُمكن أن نتجنب من خلالها وقوع حالات الانهيار، وسنستخدم أيضًا تعليمات التأجيل defer statements جنبًا إلى جنب مع الدالة `recover` لالتقاط حالات الانهيار قبل أن تؤدي إلى إيقاف برامجنا.

ا. ما هي حالات الانهيار؟

هناك عمليات محددة في لغة جو تؤدي إلى حالات الانهيار ومن ثم إيقاف البرنامج مباشرةً، وتتمثّل بعض هذه العمليات في محاولة الوصول إلى فهرس لا تتضمنه حدود المصفوفة -أي تجاوز حدود المصفوفة- أو إجراء عمليات توكيد النوع type assertions أو استدعاء توابع على مؤشرات لا تُشير إلى أيّ شيء `nil` أو استخدام كائنات المزامنة mutexes بطريقة خاطئة أو محاولة العمل مع القنوات المغلقة closed channels، فمعظم هذه المواقف تنتج عن أخطاء أثناء البرمجة ولا يستطيع المُصرّف اكتشافها أثناء تصريف البرنامج.

بما أن حالات الانهيار تتضمن تفاصيل مفيدةً لحل مشكلة ما، فعادةً ما يستخدِم المطورون حالات الانهيار بوصفها مؤشرًا على ارتكابهم خطأ أثناء تطوير البرنامج.

ب. حالات الانهيار الناتجة عن تجاوز الحدود

ستولد جو حالة انهيار في وقت التشغيل runtime عند محاولة الوصول إلى فهرس خارج حدود المصفوفة أو الشريحة، ويجسّد المثال التالي هكذا حالة، إذ سنحاول الوصول إلى آخر عنصر من الشريحة من خلال القيمة المُعادة من الدالة `len`، أي من خلال طول الشريحة.

```
package main
import (
  "fmt"
)
func main() {
  names := []string{
        "lobster",
        "sea urchin",
        "sea cucumber",
  }
  fmt.Println("My favorite sea creature is:", names[len(names)])
}
```

يكون الخرج كما يلي:

```
panic: runtime error: index out of range [3] with length 3
goroutine 1 [running]:
main.main()
  /tmp/sandbox879828148/prog.go:13 +0x20
```

لاحظ أننا حصلنا على تلميح `panic: runtime error: index out of range` والذي يشير إلى أننا نحاول الوصول إلى فهرس خارج حدود المصفوفة.

أنشأنا شريحة `names` بثلاث قيم ثم حاولنا طباعة العنصر الأخير من خلال القيمة المُعادة من الدالـة `len()` والتي ستُعيد القيمة 3، لكن آخر عنصر في المصفوفة يملك الفهرس 2 وليس 3 بما أنّ الفهرسة تبدأ من الصـفر وليس من الواحد، لذا في هذه الحالة لا يوجد خيارًا لوقت التشغيل إلا أن يتوقف ويُنهي البرنـامج، أيضًـا لا يمكن للغة جو أن تُبرهِن أثناء عملية التصريف أن الشيفرة ستحاول فعل ذلك، لذلك لا يمكن للمُصرِّف التقاط هذا.

تُسبب حالة الانهيار إيقاف تنفيذ برنامجك تمامًا، وبالتالي لن تُنفّذ أيّة تعليمات برمجيـة تاليـة، كمـا تحتـوي الرسالة الناتجة على العديد من المعلومات المفيدة في تشخيص سبب حالة الانهيار.

ج. مكونات حالة الانهيار

تتكون حالات الانهيار من رسالة تحاول توضيح سبب حالة الانهيار إضافةً إلى مسار المكدس stack trace الذي يساعدك على تحديد مكان حدوث حالـة الانهيـار في تعليماتـك البرمجيـة، إذ تبـدأ رسـالة الانهيـار بالكلمـة `panic:` تتبعها سلسلة نصية توضيحية تختلف تبعًا لسبب حالة الانهيـار، فرسـالة الانهيـار في المثـال السـابق كانت كما يلي:

```
panic: runtime error: index out of range [3] with length 3
```

تخبرنا السلسلة `runtime error:` التي تتبع الكلمة `panic:` أنّ الخطأ قد حدث في وقت التشغيل؛ أما بقية الرسالة، فتخبرنا بأن الفهرس 3 خارج حدود الشريحة.

الرسالة التالية هي رسالة مسار المكدس، وهي خريطة يمكننا تتبُعها لتحديد سطر التعليمات البرمجية الذي أدى تنفيذه إلى حدوث حالة الانهيار وكيفية استدعاء هذه الشيفرة من شيفرة أخرى.

```
goroutine 1 [running]:
main.main()
  /tmp/sandbox879828148/prog.go:13 +0x20
```

`/tmp/sandbox879828148/prog.go` يوضّح مسار المكدس هذا أن حالة الانهيـار حـدثت في الملـف في السطر رقم 13، كما يخبرنا أيضًا أنه نشأ في الدالة `main()` من الحزمة الرئيسية `main`.

ينقسم مسار المكدس إلى عدة أقسام بحيث يكون هناك قسم لكل روتين goroutine فكل عملية تنفيذ لبرنامج في جو تُنجز من خلال روتين واحد أو أكثر، أي كل روتين يُنفذ جزء محدد من الشيفرة، وكل من هذه الروتينات يُمكن تنفيذها باستقلالية عن باقي الروتينات وبالتزامن معها.

تبدأ كل كتلة بالرأس goroutine X [state]:، تُمثّل الإشارة X إلى رقم الروتين ID (مُعرّفه) مع الحالة التي كان عليها عندما حدثت حالة الانهيار، ويعرض مسار المكدس بعد الرأس الدالة التي حدثت فيها حالة الانهيار أثناء تنفيذها إلى جانب اسم الملف ورقم سطر تنفيذ الدالة.

د. الإشارة إلى العدم nil

يمكن أن تحدث حالة الانهيار أيضًا عند محاولة استدعاء تابع على مؤشر لا يُشير إلى أيّ شيء، إذ تُمكننا المؤشرات في جو من الإشارة إلى نسخ مُحددة من بعض الأنواع الموجودة في الذاكرة خلال وقت التشغيل، وعندما لا يُشير المؤشر إلى أيّ شيء، فستكون قيمة المؤشر nil، وعندما نحاول استدعاء توابع على مؤشر لا يشير إلى شيء، فستظهر حالة انهيار، والأمر ذاته بالنسبة للمتغيرات التي تكون من أنواع الواجهات، إذ تولّد أيضًا حالة الانهيار إذا استُدعيت توابع عليها كما في المثال التالي:

```
package main
import (
  "fmt"
)
type Shark struct {
  Name string
}
func (s *Shark) SayHello() {
  fmt.Println("Hi! My name is", s.Name)
}
func main() {
  s := &Shark{"Sammy"}
  s = nil
  s.SayHello()
}
```

يكون الخرج كما يلي:

```
panic: runtime error: invalid memory address or nil pointer
dereference
[signal SIGSEGV: segmentation violation code=0xffffffff addr=0x0
pc=0xdfeba]
```

```
goroutine 1 [running]:
main.(*Shark).SayHello(...)
  /tmp/sandbox160713813/prog.go:12
main.main()
  /tmp/sandbox160713813/prog.go:18 +0x1a
```

عرّفنا في هذا المثال سجل `struct` وسميناه `Shark` وهو يحتوي على تابع وحيد مُعرَّف ضمن مُستقبل مؤشره واسمه `SayHello` والذي يطبع تحية عند استدعائه، كما نُنشئ داخل جسم الدالة `main` نسخةً جديدةً من السجل `Shark` ونحاول أخذ مؤشر عليها باستخدام العامِل `&`، إذ يُسنَد المؤشر إلى المتغير `s` ثم نُسند المتغير `s` مرةً أخرى إلى القيمة `nil` من خلال التعليمة `s = nil` ثم نحاول استدعاء التابع `SayHello` على المتغير `s`.

نتيجةً لذلك نحصل على حالة انهيار تُشير إلى محاولة الوصول إلى عنوان ذاكرة غير صالح لأن المتغير `s` كانت قيمته `nil`، وبالتالي عند استدعاء الدالة `SayHello` فإنه يحاول الوصول إلى الحقل `Name` الموجود على النوع `*Shark`، وبالتالي تحدث حالة انهيار لأنه لا يمكن تحصيل dereference قيمة غير موجودة `nil` بما أنّ المؤشر هو مؤشر استقبال والمستقبل في هذه الحالة هو `nil`.

حددنا في هذا المثال القيمة `nil` صراحةً، لكن تظهر هذه القيم في الحالات العملية من دون انتباهنا، إذًا، عندما تظهر لديك حالات انهيار تتضمن الرسالة `nil pointer dereference`، ينبغي عليك التحقق مما إذا كنت تُسند قيمة `nil` إلى أحد المؤشرات التي تستخدِمها في برنامجك.

٥. استخدام دالة panic المضمنة

حالات الانهيار الناتجة عن تجاوز حدود الشريحة أو تلك الناتجة عن المؤشرات ذات القيمة `nil` هي الأشهر، لكن من الممكن أيضًا أن نُظهر حالة انهيار يدويًا من خلال الدالة المُضمّنة `panic` التي تأخذ وسيط واحد فقط يُمثِّل السلسلة النصية التي ستمثّل رسالة الانهيار، وعادةً ما يكون عرض هذه الرسالة أكثر راحةً من تعديل الشيفرة بحيث تعيد خطأً، كما يمكننا أيضًا استخدامها داخل الحزم الخاصة بنا لتنبيه المطورين إلى أنهم قد ارتكبوا خطأً عند استخدام شيفرة الحزمة خاصتنا، عمومًا، يُفضَّل إعادة قيم الخطأ `error` إلى مُستخدِمي الحزمة، وسيطلق المثال التالي حالة انهيار ناتجةً عن دالة تُستدعى من دالة أخرى:

```
package main
func main() {
  foo()
}
func foo() {
```

```
  panic("oh no!")
}
```

ستكون حالة الانهيار الناتجة كما يلي:

```
panic: oh no!
goroutine 1 [running]:
main.foo(...)
  /tmp/sandbox494710869/prog.go:8
main.main()
  /tmp/sandbox494710869/prog.go:4 +0x40
```

عرّفنا في هـذا المثـال الدالـة `foo` الـتي تسـتدعي الدالـة `panic` والـتي نُمـرر لهـا السلسـلة "oh no!" ثم استدعينا هذه الدالة `foo` من داخل الدالة `main`.

لاحظ أنّ الخرج يتضمن الرسـالة الـتي حـددناها للدالـة `panic` ويعـرض مسـار المكـدس لنـا روتينًـا واحـدًا وسطرين من المسارات؛ الأول للدالة `main()` والثاني للدالة `foo()`.

وجدنا مما سبق أنّ حالات الانهيار تؤدي إلى إنهاء البرنامج لحظة حدوثها، ويمكن أن يؤدي ذلـك إلى حـدوث مشكلات عند وجود موارد مفتوحة يجب إغلاقها بطريقة سـليمة، وهنـا تـوفِّر جـو آليـةً لتنفيـذ بعض التعليمـات البرمجية دائمًا حتى عندما تظهر حالة الانهيار.

و. الدوال المؤجلة

قد يحتوي برنامجك على موارد يجب تنظيفها بطريقة سليمة، حتى أثنـاء معالجـة حـالات الانهيـار في وقت التشغيل، إذ تسمح لك جو بتأجيل تنفيذ استدعاء دالة ريثما تكون الدالة المُستدعاة ضمنها قد اكتمل تنفيذها.

يُمكن للدوال المؤجلة أن تُنفّذ حـتى عنـد ظهـور حالـة انهيـار، وتسـتخدم بوصـفها تقنيـة أمـان للحمايـة من الفوضى والمشكلات التي قد تسببها حالة الانهيار، ولتأجيل دالة نسـتخدم الكلمـة المفتاحيـة `defer` قبـل اسـم الدالة كما يلي `defer sayHello()`، ولاحظ في المثال التالي أن الرسالة ستظهر رغم حدوث حالة انهيار:

```
package main
import "fmt"
func main() {
  defer func() {
        fmt.Println("hello from the deferred function!")
  }()
  panic("oh no!")
}
```

يكون الخرج كما يلي:

```
hello from the deferred function!
panic: oh no!
goroutine 1 [running]:
main.main()
  /Users/gopherguides/learn/src/github.com/gopherguides/learn//handle-panics/src/main.go:10 +0x55
```

نؤجل استدعاء دالة مجهولة الاسم داخل الدالة `main()` والتي تطبع الرسالة التالية "hello from the deferred function!" ثم تُطلق الدالة `main()` حالة انهيار من خلال استدعاء الدالة `panic`، ونلاحظ من الخرج أنّ الدالة المؤجلة تُنفّذ أولًا وتُطبع الرسالة الخاصة بها، ثم تظهر لنا رسالة حالة الانهيار.

توفر لغة جو لنا أيضًا فرصة منع حالة الانهيار من إنهاء البرنامج من داخل الدوال المؤجلة من خلال دالة مُضمّنة أخرى.

ز. معالجة حالات الانهيار

تمتلك جو تقنية معالجة واحدة تتجسّد في الدالة `recover`، إذ تسمح لك هذه الدالة في اعتراض حالة الانهيار من مسار المكدس ومنع الإنهاء المُفاجئ للبرنامج.

هناك قواعد صارمة عند التعامل معها، لكنها دالة لا تقُدّر بثمن في تطبيقات الإنتاج، ويمكن استدعاء الدالة `recover()` مباشرةً دون الحاجة إلى استيراد أيّ حزمة لأنها دالة مُضمّنة كما ذكرنا.

```
package main
import (
  "fmt"
  "log"
)
func main() {
  divideByZero()
  fmt.Println("we survived dividing by zero!")
}
func divideByZero() {
  defer func() {
        if err := recover(); err != nil {
              log.Println("panic occurred:", err)
```

```
        }
    }()
    fmt.Println(divide(1, 0))
}
func divide(a, b int) int {
    return a / b
}
```

يكون الخرج كما يلي:

```
2009/11/10 23:00:00 panic occurred: runtime error: integer divide by
zero
we survived dividing by zero!
```

تُستدعى الدالة `divideByZero` من داخل الدالة `main`، كما داخل هذه الدالة نؤجل استدعاء دالة مجهولة الاسم ومسؤولة عن التعامل مع أيّ حالات انهيار قد تنشأ أثناء تنفيذ `divideByZero`، ونستدعي داخل هذه الدالة مجهولة الاسم الدالة المُضمّنة `recover` ونسند الخطأ الذي تُعيده إلى المتغير `err`.

إذا ظهرت حالة انهيار في الدالة `divideByZero()`، فستُهيّأ قيمة هذا الخطأ، وإلا فستكون `nil`، ومن خلال مقارنة قيمة المتغير `err` بالقيمة `nil`، سنستطيع تحديد فيما إذا كانت حالة الانهيار قد حدثت، وفي هذه الحالة نسجّل حالة الانهيار من خلال الدالة `log.Println` كما لو كانت أيّ خطأ آخر.

نلاحظ أننا نستدعي الدالة `divide` ونحاول طباعة ناتجها باستخدام الدالة `fmt.Println()` بتتبع الدالة المؤجلة مجهولة الاسم بما أن وسيط هذه الدالة سيُسبب القسمة على صفر، وبالتالي ستظهر حالة انهيار.

يُظهر خرج البرنامج رسالة الدالة `log.println()` من الدالة مجهولة الاسم والتي تُعالج حالة الانهيار متبوعة برسالة `we survived dividing by zero!`، والتي تُشير إلى أننا منعنا حالة الانهيار من إيقاف البرنامج، وذلك بفضل استخدام دالة المعالجة `recover`.

قيمة الخطأ `err` المُعادة من الدالة `recover()` هي نفسها القيمة المُعطاة لاستدعاء الدالة `panic()`، لذا من المهم التأكد مما إذا كانت قيمتها `nil`.

ج. اكتشاف حالات الانهيار باستخدام recover

تعتمد الدالة `recover` على قيمة الخطأ في اكتشاف حدوث حالات الانهيار، وبما أن وسيط الدالة `panic` هو واجهة فارغة، لذا يمكن أن تكون من أي نوع، كما أنّ القيمة الصفرية لأي نوع بيانات يُمثّل واجهةً هو `nil`، ويوضِّح المثال التالي أنه يجب تجنب تمرير القيمة `nil` على أساس وسيط للدالة `panic`:

```
package main
```

```
import (
  "fmt"
  "log"
)
func main() {
  divideByZero()
  fmt.Println("we survived dividing by zero!")
}
func divideByZero() {
  defer func() {
        if err := recover(); err != nil {
              log.Println("panic occurred:", err)
        }
  }()
  fmt.Println(divide(1, 0))
}
func divide(a, b int) int {
  if b == 0 {
        panic(nil)
  }
  return a / b
}
```

يكون الخرج كما يلي:

```
we survived dividing by zero!
```

هذا المثال هو المثال السابق نفسه مُتضمنًا الدالة `recover` مع بعض التعديلات الطفيفة، إذ عدّلنا دالة القسمة للتحقق مما إذا كان المقسوم عليه `b` يساوي 0، فإذا كان كذلك، فسيولّد حالة انهيار باستخدام الدالة `panic` مع وسيط من القيمة `nil`.

لن يتضمن الخرج في هذه المرة رسالة الدالة `log` التي تعرض حدوث حالة الانهيار بالرغم من حدوث حالة انهيار نتيجة القسمة، ويؤكد هذا السلوك الصامت أهمية التحقق من قيمة الخطأ فيما إذا كانت `nil`.

11.10 الخاتمة

استعرضنا في هذا الفصل طرق التعامل مع الأخطاء في لغة جو، وتعرفنا على عدة طرق لإنشاء الأخطاء باستخدام المكتبة القياسية على وكيفية إنشاء دوال تُعيد الأخطاء بطريقة اصطلاحية، وأنشانا العديد من الأخطاء

بنجاح باستخدام دوال المكتبة القياسية `errors.New()` و `fmt.Errorf()`، كما شـرحنا أيضًـا كيفيـة إنشـاء أنواع أخطاء مُخصصة وكيفية تتبع الأخطاء التي تحـدث في البرنـامج من خلال تغليفهـا، وأخـيرًا وضـحنا مفهـوم حالات انهيار وسبب ظهورهـا ومعالجتهـا، وليس من الضـروري أن تحتـاج إلى التعامـل مـع حـالات الانهيـار في برامجك، لكن عندما يتعلق الأمر بتطبيقات الإنتاج فهي بغاية الأهمية.

دورة علوم الحاسوب

مميزات الدورة

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

اشترك الآن

12. التعامل مع الحزم

عادةً ما نحتاج إلى استخدام دوال خارجية أخرى في البرنامج، هـذه الـدوال تكـون ضـمن حـزم خارجيـة بناهـا مبرمجون آخرون أو بنيناها بأنفسنا لتقسيم البرنامج إلى عدة ملفات بهـدف جعـل البرنـامج أبسـط وأقـل تعقيـدًا وأكثر أناقةً وقابليةً للفهم.

تُمثّل الحزمة مجلدًا يحتوي على ملف أو عدة ملفات، هذه الملفـات هي شـيفرات جـو قـد تتضـمن دوال أو أنواع بيانات أو واجهات يمكنك استخدامها في برنامجك الأساسي، وسنتحدث في هذا الفصل عن تثبيت الحزم واستيرادها وتسميتها.

12.1 حزمة المكتبة القياسية

تُعَدّ المكتبة القياسية التي تأتي مع جو مجموعةً من الحزم، إذ تحتـوي هـذه الحـزم على العديـد من اللبنـات الأساسية لكتابة البرامج المتطورة في جو، فتحتوي الحزمة `fmt` مثلًا على العديد من الدوال التي تُسهّل عمليـات تنسيق وطباعة السلاسل النصية، كما تحتوي حزمة `net/http` على دوال تسمح للمطوّر بإنشـاء خـدمات ويب وإرسال واسترداد البيانات عبر بروتوكول `http` إضافةً للعديد من الأمور الأخرى.

يجب عليك أولًا استيراد هذه الحزمة باستخدام التعليمة `import` متبوعة باسم الحزمة للاستفادة من الدوال الموجودة ضـمن أيّ حزمـة، فيمكنـك مثلًا اسـتيراد الحزمـة `math/rand` ضـمن الملـف random.go لتوليـد أعداد عشوائية:

```
import "math/rand"
```

عند استيراد حزمة ما داخل برنامج تصبح جميع مكوناتها قابلةً للاستخدام داخل البرنامج لكن بفضاء أسماء منفصل namespace، لذا يجب الوصول إلى كل دالة من الحزمة الخارجية بالاعتماد على تدوين النقطة dot notation، أي `package.function`، فللوصول مثلًا إلى الدوال الموجودة في الحزمة السابقة نكتب `rand.Int()` لتوليد عدد صحيح عشوائي أو `rand.Intn()` لتوليد عدد عشوائي بين الصفر والعدد المحدَّد.

سنستخدِم في المثال التالي حلقة `for` لتوليد أعداد عشوائية بالاستعانة بالدوال السابقة:

```
package main
import "math/rand"
func main() {
 for i := 0; i < 10; i++ {
  println(rand.Intn(25))
 }
}
```

يكون الخرج كما يلي:

```
6
12
22
9
6
18
0
15
6
0
```

نستورد بدايةً الحزمة `math/rand` ثم نستخدِم حلقة تتكرر 10 مرات وفي كل مرة تطبع عدد صحيح بين 0 و 25 بحيث تكون الأعداد الناتجة أقل تمامًا من 25 بما أننا مررنا العدد 25 للدالة `rand.Intn()`، وطبعًا عملية التوليد هي عشوائية، لذا ستحصل على أعداد مختلفة في كل مرة تُنفِّذ فيها هذا المقطع البرمجي.

عند استيراد أكثر من حزمة نضع هذه الحزم بين قوسين `()` بعد الكلمة `import` كما في المثال التالي:

```
import (
 "fmt"
 "math/rand"
)
```

سنستخدِم الحزمة الجديدة التي استوردناها من أجل تنسيق عملية الطباعة في الشيفرة السابقة:

```
package main
import (
 "fmt"
 "math/rand"
)
func main() {
 for i := 0; i < 10; i++ {
  fmt.Printf("%d) %d\n", i, rand.Intn(25))
 }
}
```

يكون الخرج كما يلي:

```
0) 6
1) 12
2) 22
3) 9
4) 6
5) 18
6) 0
7) 15
8) 6
9) 0
```

تعلّمت في هذا القسم كيفية استيراد الحزم واستخدامها لكتابة برنامج أكثر تعقيدًا، لكن لم نستخدِم إلا الحزم الموجودة في المكتبة القياسية، وفيما يلي سنرى كيفية تثبيت واستخدام الحزم التي كتبها مطورون آخرون.

12.2 تثبيت الحزم

تحتوي المكتبة القياسية على العديد من الحزم المفيدة لكنها لأغراض عامة، لذا يُنشئ المطورون حزمًا خاصةً بهم فوق المكتبة القياسية لتلبية احتياجاتهم الخاصة، وتتضمن أدوات جو الأمر `go get` الذي يسمح بتثبيت حزم خارجية ضمن بيئة التطوير المحلية الخاصة بك واستخدامها في برنامجك، وعند استخدام هذا الأمر تُشاع الإشارة إلى الحزمة من خلال مسارها الأساسي، كما يمكن أن يكون هذا المسار مسارًا إلى مشروع عام مُستضاف في مستودع شيفرات مثل جيت هاب GitHub، فلاستيراد الحزمة `flect` مثلًا نكتب:

```
$ go get github.com/gobuffalo/flect
```

سيعثر الأمر `go get` على هذه الحزمة الموجودة على جيت هاب ويثبتها في `$GOPATH`، إذ ستُثبّت الحزمة في هذا المثال ضمن المجلد:

```
$GOPATH/src/github.com/gobuffalo/flect
```

غالبًا ما تُحدَّث الحزم من قِبل مُطوريها لمعالجة العلّات البرمجية أو إضافة ميزات جديدة، وقد ترغب في هذه الحالة باستخدام أحدث إصدار من تلك الحزمة للاستفادة من الميزات الجديدة أو العلّة البرمجية التي عالجوها، إذ يمكنك تحديث حزمة ما باستخدام الراية `-u` مع الأمر `go get`:

```
$ go get -u github.com/gobuffalo/flect
```

سيؤدي هذا الأمر أيضًا إلى تثبيت الحزمة إذا لم تكن موجودةً لديك أساسًا، ويؤدي الأمر `go get` دائمًا إلى تثبيت أحدث إصدار من الحزمة المتاحة، لكن قد ترغب في استخدام حزمة سابقة أو قد تكون هناك نسخة سابقة قد حُدّثت وترغب في استخدامها، ففي هذه الحالة ستحتاج إلى استخدام أداة إدارة الحزم مثل `Go Modules`، واعتبارًا من الإصدار Go 1.11 اعتُمِدت `Go Modules` بوصفها أداة إدارة حزم رسمية لجو.

12.3 تسمية الحزم بأسماء بديلة

قد تحتاج إلى تغيير اسم الحزمة التي تستوردها في حال كان لديك حزمة أخرى تحمل الاسم نفسه، ففي هذه الحالة يمكنك استخدام الكلمة `alias` لتغيير اسم الحزمة وتجنب تصادم الأسماء وفق الصيغة التالية:

```
import another_name "package"
```

سنعدِّل في المثال التالي اسم الحزمة `fmt` ضمن ملف random.go ليصبح `f`:

```
package main
import (
 f "fmt"
 "math/rand"
)
func main() {
 for i := 0; i < 10; i++ {
  f.Printf("%d) %d\n", i, rand.Intn(25))
 }
}
```

كتبنا `f.Printf` في البرنامج السابق بدلًا من `fmt.Printf`، وعمومًا لا يُفضَّل في جو استخدام الأسماء البديلة كما في باقي اللغات مثل بايثون لأنه خروج عن النمط الشائع.

عند اللجوء إلى الأسماء البديلة لتجنب حدوث تصادم في الأسماء، يفضَّل إعادة تسمية الحزم المحلية وليس الحزم الخارجيّة، فإذا كان لديك مثلًا حزمة أنشأتها باسم `strings` وتريد استخدامها في برنامجك إلى جانب حزمة النظام `strings`، فيُفضّل تسمية الحزمة التي أنشأتها أنت وليس حزمة النظام.

تذكَّر دومًا أنّ سهولة القراءة والوضوح في برنامجك أمر مهم، لذا يجب عليك استخدام الأسماء البديلة فقط لجعل الشيفرة أكثر قابليةً للقراءة أو عندما تحتاج إلى تجنب تضارب الأسماء.

12.4 تنسيق الحزم

يمكنك فرز الحزم بترتيب معيّن من خلال تنسيق عمليات الاستيراد لجعل شفرتك أكثر اتساقًا ومنع حدوث تغييرات أو إيداعات عشوائية random commits عندما يكون الشيء الوحيد الذي يتغير هو ترتيب فرز عمليات الاستيراد، فمنع الإيداعات العشوائية سيمنع حدوث خلل في التعليمات البرمجية أو ارتباك أثناء مراجعة الشفرة من قِبل الآخرين.

تُنسّق معظم محررات الشيفرات استيراد الحزم تلقائيًا أو يسمحون لك باستخدام الأداة `goimports`، إذ يُعَدّ استخدام هذه الأداة أمرًا مفيدًا وشائعًا وممارسةً قياسيةً، فالحفاظ على ترتيب الفرز يدويًا قد يكون مُملًا ومعرّضًا للأخطاء، بالإضافة إلى ذلك، إذا أُجريَت أية تغييرات على التنسيق القياسي لفرز الحزم، فستُحدَّث `goimports` لتعكس تلك التغييرات في التنسيق، مما يضمن لك ولشركائك أنه سيكون لديكم تنسيق متسق لكتل الاستيراد، وإليك ما قد تبدو عليه كتلة الاستيراد قبل التنسيق:

```
import (
 "fmt"
 "os"
 "github.com/digital/ocean/godo"
 "github.com/sammy/foo"

 "math/rand"
 "github.com/sammy/bar"
)
```

سيكون لديك الآن التنسيق التالي بتشغيل الأداة `goimport` -أو في حالة مُحرّر يستخدِم هذه الأداة، فسيؤدي حفظ الملف إلى تشغيله نيابة عنك-:

```
import (
 "fmt"
 "math/rand"
 "os"
```

```
    "github.com/sammy/foo"
    "github.com/sammy/bar"
    "github.com/digital/ocean/godo"
)
```

لاحظ أنه يُجمِّع حزم المكتبة القياسية أولًا ثم الحزم الخارجية مع فصلها بسطور فارغة مما يجعل من السهل قراءة وفهم الحزم المستخدَمة.

سيؤدي استخدام الأداة `goimports` إلى الحفاظ على تنسيق جميع كتل الاستيراد بطريقة مناسبة، ويمنع حدوث ارتباك حول التعليمات البرمجية بين المطورين الذين يعملون على الملفات نفسها.

12.5 إنشاء الحزم

تُعَدّ الحزمة مجموعةً من الملفات الموجودة ضمن مجلد واحد والمتضمّنة لتعليمة الحزمة نفسها في بدايتها، ويمكنك تضمين العديد من الحزم ضمن برنامجك عند الحاجة لبناء برمجيات أكثر تعقيدًا.

وكما تعلمت في الفقرات السابقة، تكون بعض الحزم موجودةً في مكتبة جو القياسية وبعضها الآخر يمكنك تثبيته من خلال الأمر `go get`، كما يمكنك أيضًا إنشاء الحزم الخاصة بك من خلال بناء ملفات لغة جو التي تحتاجها ووضعه ضمن المجلد نفسه مع الالتزام بكتابة تعليمة الحزمة الضرورية في بداية كل ملف، كما ستتعلم في هذا الفصل كيفية إنشاء حزم لغة جو الخاصة بك لاستخدامها في برامجك.

كي تتمكن من إنشاء الحزم أن يكون لديك مساحة عمل خاصة في لغة جو، فإذا لم يكن لديك، فارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبت لغة جو Go حسب نظام تشغيلك وإعداد بيئة تطوير محلية، و يُفضّل أن تكون قد اطلعت أيضًا على فقرة التعرّف على GOPATH وفقرة استيراد الحزم في لغة جو Go قبل المتابعة في الفقرات التالية.

12.5.1 كتابة واستيراد الحزم

تشبه كتابة الحزم كتابة أيّ برنامج آخر في لغة جو، ويمكن أن تتضمن الحزم دوالًا أو متغيرات أو أنواع بيانات خاصة يمكنك استخدامها في برنامجك لاحقًا. ويجب أن تكون ضمن مساحة العمل الخاصة بك قبل إنشاء حزم جديدة، وهذا يعني أن تكون ضمن مسار البيئة `gopath`،

في هذا الفصل على سبيل المثال سنُنشئ الحزمة `greet`، لذا سوف نُنشئ المجلد `greet` ضمن مسار البيئة وضمن مساحة العمل الخاصة بك، فإذا كنا ضمن مساحة العمل `gopherguides` وأردنا إنشاء الحزمة `greet` ضمنها أثناء استخدام جيت هاب Github على أساس مستودع تعليمات برمجية، فسيبدو المجلد كما يلي:

```
└── $GOPATH
  └── src
    └── github.com
      └── gopherguides
```

ستكون الحزمة `greet` ضمن المجلد `gopherguides`:

```
└── $GOPATH
  └── src
    └── github.com
      └── gopherguides
        └── greet
```

يمكننا الآن إنشاء أول ملف في الحزمة، ويسمى الملف الأساسي -ويسمى أيضًا نقطة الدخول entry point- في الحزمة عادةً باسم الحزمة نفسها، إذًا سنُنشئ الملف `greet.go` ضمن المجلد `greet` كما يلي:

```
└── $GOPATH
  └── src
    └── github.com
      └── gopherguides
        └── greet
          └── greet.go
```

يمكننا بعد إنشاء الملف كتابة التعليمات البرمجية التي نريدها ضمنه، والهدف من هذه التعليمات عادةً هو الاستخدام في مكان آخر من المشروع، وفي هذه الحالة سننشئ دالة باسم `Hello()` تطبع السلسلة النصية "Hello, World!"، لذا افتح الملف greet.go من خلال محرر الشيفرات الخاص بك، واكتب الكود التالي:

```go
package greet
import "fmt"
func Hello() {
  fmt.Println("Hello, World!")
}
```

يجب دومًا أن نبدأ باسم الحزمة التي نعمل ضمنها لكي نُخبر المصرِّف أنّ هذا الملف هو جزء من الحزمة، لذا كتبنا الكلمة المفتاحية `package` متبوعةً باسم الحزمة:

```go
package greet
```

ثم نكتب اسم الحزم التي نحتاج إلى استخدامها ضمن الملف من خلال وضع أسمائها بعد الكلمة المفتاحيـة `import`، وهنا نحتاج إلى حزمة `fmt` فقط:

```
import "fmt"
```

أخيرًا سنكتب الدالة `Hello()` التي تستخدم الحزمة `fmt` لتنسيق طباعة جملة الخرج:

```
func Hello() {
  fmt.Println("Hello, World!")
}
```

انتهينا من إنشاء الملف الأول وأصبح بإمكاننا استخدام الدالة `Hello()` منه وفي المكان الذي نريده.

نُنشئ حزمةً جديدةً باسم `example`، لذا نُنشئ مجلدًا بالاسم نفسه في مساحة العمل `gopherguides`:

```
└── $GOPATH
  └── src
    └── github.com
      └── gopherguides
        └── example
```

لديك الآن مجلد خاص بالحزمة، وبإمكانك إنشاء ملف نقطة الدخول وهو ملف تنفيذي نسميه main.go:

```
└── $GOPATH
  └── src
    └── github.com
      └── gopherguides
        └── example
          └── main.go
```

افتح الملف من محرر الشيفرات الخاص بك واكتب التعليمات التالية:

```
package main
import "github.com/gopherguides/greet"
func main() {
  greet.Hello()
}
```

بما أن الدالة التي تريد استخدامها ضمن الملف الرئيسي موجـودة ضـمن حزمـة أخـرى، فسـيتوجب عليـك استدعاؤها من خلال ذكر اسم الحزمة أولًا متبوعًا بنقطة ثم اسم الدالة، فمثلًا وضعنا هنا اسـم الحزمـة `greet` ثم نقطة ثم اسم الدالة `greet.Hello()`، ويمكنك الآن فتح الطرفية وتشغيل البرنامج ضمنها:

```
go run main.go
```

سيكون الخرج كما يلي:

```
Hello, World!
```

سنضيف بعض المتغيرات إلى ملف `greet.go` لتتعلم كيفية استخدام المتغيرات ضمن الحزمة:

```
package greet
import "fmt"
var Shark = "Sammy"
func Hello() {
  fmt.Println("Hello, World!")
}
```

ثم افتح الملف main.go وأضف التعليمـة التاليـة `fmt.Println(greet.Shark)` لاسـتدعاء المتغـير `Shark` داخل الدالة `fmt.Println`، أي كما يلي:

```
package main
import (
  "fmt"
  "github.com/gopherguides/greet"
)
func main() {
  greet.Hello()
  fmt.Println(greet.Shark)
}
```

ثم شغّل الشيفرة مرةً أخرى:

```
$ go run main.go
```

ستحصل على الخرج التالي:

```
Hello, World!
Sammy
```

أخيرًا، سننشئ نوع بيانات جديد ضمن ملف `greet.go`، إذ سننشئ نوع البيانات `Octopus` الذي يتضمّن الحقلين `name` و `color`، كما سنعرّف دالةً تطبع هذه الحقول:

```
package greet
import "fmt"
var Shark = "Sammy"
type Octopus struct {
  Name string
  Color string
}
func (o Octopus) String() string {
  return fmt.Sprintf("The octopus's name is %q and is the color %s.",
o.Name, o.Color)
}
func Hello() {
  fmt.Println("Hello, World!")
}
```

سننشئ الآن نسخةً من هذا النوع داخل الملف `main.go`:

```
package main
import (
  "fmt"
  "github.com/gopherguides/greet"
)
func main() {
  greet.Hello()
  fmt.Println(greet.Shark)
  oct := greet.Octopus{
        Name: "Jesse",
        Color: "orange",
  }
  fmt.Println(oct.String())
}
```

بمجرّد إنشاء نسخة من النوع `Octopus` عند كتابة `oct := greet.Octopus` يُصبح بإمكاننا الوصول إلى الدوال والمتغيرات الموجودة ضمنه من الملف `main`، وبالتالي إمكانية استدعاء الدالة `oct.String()`

من دون الحاجة لكتابة اسم الحزمة greet، كما يمكنك الوصول إلى الحقول بالطريقة نفسها دون الحاجة إلى كتابة اسم الحزمة مثل oct.Color.

يستخدِم التابع String التي يتضمنها النوع Octopus الدالة fmt.Sprintf لإنشاء وإرجاع سلسلة في المكان الذي استُدعي فيه أي في هذه الحالة في الملف main، وسنشغّل البرنامج الآن كما يلي:

```
$ go run main.go
```

يكون الخرج كما يلي:

```
Hello, World!
Sammy
The octopus's name is "Jesse" and is the color orange.
```

إذًا سيصبح لدينا دالة يمكن استخدامها حيثما نريد لطباعة معلومات عن نوع البيانات الذي عرّفناه من خلال تعريفنا للتابع String ضمن النوع Octopus، فإذا أردت تغيير سلوك هذا التابع لاحقًا، فيمكنك ببساطة تعديله فقط حيثما يكون.

12.5.2 تصدير الشيفرة

لاحظ أنّ كل التصريحات داخل الملف greet.go تبدأ بمحرف كبير، ولا تمتلك لغة جو مفاهيم مُحددات الوصول العامة public والخاصة private والمحمية protected كما في باقي اللغات، ويمكن التحكم بالرؤية في لغة جو من خلال الكتابة بمحارف كبيرة، فالمتغيرات أو الدوال أو الأنواع التي تبدأ بمحارف كبيرة تكون مرئيةً من خارج الحزمة -أي عامة- ويُعتبر عندها مُصدّرًا exported.

إذا أضفت تابعًا جديدًا إلى النوع Octopus اسمه reset، فستتمكّن من استدعائه من داخل الحزمة greet، لكن لن تتمكن من استدعائه من الملف main.go لأنه خارج الحزمة greet:

```
package greet
import "fmt"
var Shark = "Sammy"
type Octopus struct {
  Name string
  Color string
}
func (o Octopus) String() string {
  return fmt.Sprintf("The octopus's name is %q and is the color %s.",
o.Name, o.Color)
}
```

```
func (o *Octopus) reset() {
  o.Name = ""
  o.Color = ""
}
func Hello() {
  fmt.Println("Hello, World!")
}
```

إذا حاولت استدعاء `reset` من الملف main.go:

```
package main
import (
  "fmt"
  "github.com/gopherguides/greet"
)
func main() {
  greet.Hello()
  fmt.Println(greet.Shark)
  oct := greet.Octopus{
        Name: "Jesse",
        Color: "orange",
  }
  fmt.Println(oct.String())
  oct.reset()
}
```

ستحصل على الخطأ التالي والذي يقول أنه لا يمكن الإشارة إلى حقل أو دالة غير مصدرة:

```
oct.reset undefined (cannot refer to unexported field or method
greet.Octopus.reset)
```

لتصدير دالة `reset` من `Octopus`، اجعل المحرف الأول من الدالة كبيرًا، أي `Reset`:

```
package greet
import "fmt"
var Shark = "Sammy"
type Octopus struct {
  Name string
  Color string
```

```
}
func (o Octopus) String() string {
   return fmt.Sprintf("The octopus's name is %q and is the color %s.",
o.Name, o.Color)
}
func (o *Octopus) Reset() {
   o.Name = ""
   o.Color = ""
}
func Hello() {
   fmt.Println("Hello, World!")
}
```

وبالتالي سيصبح بإمكانك استدعائها من الحزمة الأخرى بدون مشاكل:

```
package main
import (
   "fmt"
   "github.com/gopherguides/greet"
)
func main() {
   greet.Hello()
   fmt.Println(greet.Shark)
   oct := greet.Octopus{
         Name: "Jesse",
         Color: "orange",
   }
   fmt.Println(oct.String())
   oct.Reset()
   fmt.Println(oct.String())
}
```

الآن إذا شغّلت البرنامج:

```
$ go run main.go
```

ستحصل على الخرج التالي:

```
Hello, World!
```

```
Sammy
The octopus's name is "Jesse" and is the color orange
The octopus's name is "" and is the color.
```

نلاحظ من خلال استدعاء الدالة `Reset` أن جميع بيانات الحقول `Name` و `Color` قد مسحت، وعند استدعاء التابع `String` لا تطبع شيئًا لأن الحقول السابقة قد أصبحت فارغةً.

12.6 الخاتمة

تعرفنا في هذا الفصل على كيفية استيراد حزم المكتبة القياسية وكيفية استيراد وتثبيت الحزم الخارجية باستخدام الأمر `go get` للاستفادة من محتوياتها، كما تحدّثنا أيضًا عن كيفية تحديث الحزم واستخدام الأسماء البديلة معها، كما ناقشنا كيفية تعريف وإنشاء الحزم وكيفية الاستفادة منها في ملفات أخرى وأين يمكن وضعها للوصول إليها في لغة جو وكما لاحظت فإن كتابة الحزم يشبه كتابة أيّ ملف في لغة جو، إلا أن وضعها في مجلد مُختلف سيسمح لك بعزل الشيفرة لإعادة استخدامها في مكان آخر، إذ يتيح لنا استخدام الحزم جعل برامجنا أكثر قوةً ومتانةً وقابليةً للصيانة وللاستخدام المتعدّد من خلال تقسيمه إلى أجزاء مستقلّة.

13. فهم مجال رؤية الحزم

الهدف من إنشاء الحزم في لغة جو أو في أي لغة برمجة أخرى هو جعل هذه الحزمة متاحة للاستخدام وسهلة الوصول في أيّ وقت من قِبَل مطورين آخرين أو حتى نحن، إذ تُستخدَم هـذه الحـزم ضـمن برمجيـات محـددة أو بوصفها جزءًا من حـزم أكـثر تعقيـدًا أو أعلى مسـتوى، لكن لايمكن الوصـول إلى جميـع الحـزم من خـارج الحزمـة نفسها، ويعتمد ذلك على نطاق رؤيتها.

يشير مفهوم الـرؤية Visibility إلـى النطاق الذي يمكن الوصول للحزمة ضمنه، فإذا صرّحت مثلًا عن متغير داخل دالة ضمن حزمة A، فلن تتمكن من الوصول إلى هذا المتغير إلى ضمن الدالة التي صُرّح عنه فيها وضـمن الحزمـة A فقـط، في حين إذا صـرّحت عن المتغـير نفسـه ضـمن الحزمـة وليس بـداخل دالـة أو أيّ نطـاق آخـر، فيمكنك السماح للحزم الأخرى بالوصول إلى هذا المتغير أو لا.

عند التفكير بكتابـة شـيفرة مريحـة ومرتّبـة ومنظّمـة لا بـدّ من أن تكـون دقيقًـا في ضـبط **رؤية الحـزم** في مشروعك، ولاسيما عندما يكـون من المحتمـل تعـديل ملفـات مشـروعك باسـتمرار، فعنـدما تحتـاج إلى إجـراء تعديلات مثل إصلاح علّة برمجية أو تحسـين أداء أو تغيـير دوال ...إلخ. بطريقـة منظمـة وبـدون فوضـى أو إربـاك للمطورين الآخرين الذي يستخدمون الحزمة، فلا بدّ من أن تكون دقيقًـا في تحديـد الرؤيـة، وتتمثّـل إحـدى طـرق الضبط الدقيق لمثل هذه العمليـات في تقييـد الوصـول -أي تقييـد الرؤيـة-، بحيث تسـمح بالوصـول إلى أجـزاء مُحددة فقط من الحزمة -أي الأجزاء القابلة للاستخدام فقـط-، وبـذلك تتمكّن من إجـراء التغيـيرات الداخليـة على حِزَمك مع تقليل احتمالية التأثير على استخدام المطورين للحزمة.

ستتعلم في هذا الفصل كيفية التحكم في رؤية الحزمة، وكذلك كيفية حماية أجـزاء من التعليمـات البرمجيـة الخاصة بك والتي يجب استخدامها فقط داخل حزمتـك، إذ سننشـئ أداة تسـجيل لتسـجيل الرسـائل وتصـحيح الأخطاء باستخدام حزم تتضمن عناصر لها درجات متفاوتة من الرؤية.

13.1 المتطلبات

كي تتمكن من فهم مجال الرؤية الخاص بالحزم أن يكون لديك مساحة عمل خاصة في لغة جو، فـإذا لم يكن لديك، فارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبت لغة جو Go وقم بإعـداد بيئـة تطـوير محليـة بحسب نظام تشغيلك، و يُفضّل أن تكون قد اطلعت أيضًـا على فقـرة التعـرّف على GOPATH وفقـرة اسـتيراد الحزم في لغة جو Go قبل المتابعة في قراءة الفقرات التالية.

سيعتمد هذا الفصل على بنية المجلد التالية:

```
.
├── bin
│
└── src
  └── github.com
    └── gopherguides
```

13.2 العناصر المصدرة وغير المصدرة

لا تمتلك لغة البرمجة جو محددات وصـول لتحديـد الرؤيـة مثـل عـام `public` وخـاص `private` ومحمي `protected` كما في باقي لغات البرمجة، إذ تحدِّد لغة جو إمكانية رؤية العناصر اعتمادًا على ما إذا كـان مُصـدّرًا أم لا وذلك وفقًا لكيفية التصريح عنه، فإذا كان مصدّرًا فهو مرئي من خـارج الحزمـة وإلا فهـو مـرئي فقـط داخـل الحزمة التي أُنشئ فيها.

لكي نجعل عنصرًا ما مثل متغير أو دالـة أو نـوع بيانـات جديـد ...إلخ. داخـل الحزمـة مُصـدّرًا، سـنكتب أول محرف منه كبيرًا، أي إذا كان اسـم الدالـة `hsoue()`، فسـتكون مرئيـةً فقـط ضـمن الحزمـة نفسـها وتكـون غـير مُصدّرة؛ أما إذا كتبناها `Hsoub()`، فستكون مرئيةً من خارج الحزمة وتكون مُصدّرة.

لاحظ المثال التالي وانتبه لحالة أول محرف في كل التصريحات:

```go
package greet
import "fmt"
var Greeting string
func Hello(name string) string {
  return fmt.Sprintf(Greeting, name)
}
```

من الواضح أنّ هذه الشيفرة موجودة ضمن الحزمة greet، ولاحظ أننا صرّحنا عن متغير اسمه Greeting وجعلنا أول محرف كبيرًا، لذا فهذا المتغير سيكون مُصدّرًا ويمكن رؤيته من خارج الحزمة، وينطبق الأمر نفسه على الدالة Hello()، فقد كتبنا أول محرف منها كبيرًا.

كما ذكرنا سابقًا، إنّ تحديد إمكانية الوصول أو الرؤية يجعل الشيفرة منظمةً ويمنع حدوث فوضى أو إرباك للمطورين الآخرين الذي يستخدِمون الحزمة، كما يمنع أو يقلل من إمكانية حدوث تأثيرات سلبية على مشاريعهم التي تعتمد على حزمتك.

13.3 تحديد رؤية الحزمة

سنُنشئ الحزمة logging مع الأخذ بالحسبان ما نريده مرئيًا وما نريده غير مرئي لكي يكون لدينا نظرة أوضح عن آلية عمل قابلية الرؤية في برامجنا.

ستكون هذه الحزمة مسؤولةً عن تسجيل أيّ رسالة من البرنامج إلى وحدة التحكم console، كما يُحدد المستوى الذي يحدث عنده التسجيل، إذ يصف المستوى نوع السجل وسيكون أحد الحالات الثلاث: معلومات info أو تحذير warning أو خطأ error، لذا أنشئ بدايةً مجلد الحزمة logging داخل المجلد src كما يلي:

```
$ mkdir logging
```

انتقل إلى مجلد الحزمة:

```
$ cd logging
```

أنشئ ملف logging.go باستخدام محرر شيفرات مثل نانو nano:

```
$ nano logging.go
```

وضع فيه الشيفرة التالية:

```
package logging
import (
  "fmt"
  "time"
)
var debug bool
func Debug(b bool) {
  debug = b
}
func Log(statement string) {
```

```
    if !debug {
        return
    }
    fmt.Printf("%s %s\n", time.Now().Format(time.RFC3339), statement)
}
```

تصف التعليمة الأولى في هذه الشيفرة أننا ضمن الحزمة logging، وتوجد لدينا ضمن هذه الحزمة دالتين مُصدّرتين هما Debug و Log، وبالتالي يمكن استدعاء هذه الدوال من أيّ حزمة أخرى تستورد الحزمة logging، ولدينا أيضًا متغير غير مُصدّر اسمه debug، وبالتالي لا يمكن الوصول إليه إلى ضمن الحزمة.

على الرغم من أنه لدينا متغير ودالة بالاسم نفسه debug، إلا أنهما مختلفان، فالدالة بدأت بمحرف كبير؛ أما المتغير، فقد بدأ بمحرف صغير ولغة جو حساسة لحالة المحارف.

احفظ الملف بعد وضع الشيفرة فيه، ولاستخدام الحزمة في مكان آخر، يجب استيرادها، لذا سنُنشئ حزمةً جديدةً ونجرب عليها، وبالتالي انتقل إلى المجلد logging وأنشئ مجلدًا اسمه cmd وانتقل إليه:

```
$ cd ..
$ mkdir cmd
$ cd cmd
```

أنشئ الملف main.go بداخله:

```
$ nano main.go
```

أضف إليه الشيفرة التالية:

```
package main
import "github.com/gopherguides/logging"
func main() {
    logging.Debug(true)
    logging.Log("This is a debug statement...")
}
```

أصبح لدينا الآن كامل الشيفرة المطلوب، لكن نحتاج قبل تشغيلها إلى إنشاء ملفَي ضبط حتى تعمل التعليمات البرمجية بطريقة صحيحة.

تُستخدَم وحدات لغة جو Go Modules لضبط اعتماديات الحزمة لاستيراد الموارد، إذ تُعَدّ وحدات لغة جو ملفات ضبط موضوعة في مجلد الحزمة الخاص بك وتخبر المُصرّف بمكان استيراد الحزم منه، ولن نتحدث عن

وحدات لغة جو في هذا الفصل، لكن سنكتب السطرين التاليين لكي تعمل الشيفرة السابقة، لذا افتح ملف go.mod ضمن المجلد cmd:

```
$ nano go.mod
```

ثم ضع السطرين التاليين فيه:

```
module github.com/gopherguides/cmd
replace github.com/gopherguides/logging => ../logging
```

يُخبر السطر الأول المُصرّف أنّ الحزمة cmd لديها مسار الملف github.com/gopherguides/cmd؛ أما السطر الثاني، فيُخبر المُصرّف أنّ الحزمة github.com/gopherguides/logging يمكن العثور عليها في المجلد ../logging على القرص المحلي.

نحتاج أيضًا إلى وجود ملف go.mod بداخل الحزمة logging، لذا سننتقل إلى مجلدها ثم سننشئ هذا الملف بداخله:

```
$ cd ../logging
$ nano go.mod
```

أضف الشيفرة التالية إلى الملف:

```
module github.com/gopherguides/logging
```

يخبر هذا الكود مترجم لغة جو بأنّ حزمة logging التي أنشأناها هي الحزمة main في github.com/gopherguides/logging، إذ سيسمح لنا ذلك باستيراد الحزمة من داخل الحزمة كما يلي:

```
package main
import "github.com/gopherguides/logging"
func main() {
  logging.Debug(true)
  logging.Log("This is a debug statement...")
}
```

يجب أن يكون لديك الآن بنية المجلد التالية:

```
├── cmd
|   ├── go.mod
|   └── main.go
```

```
└── logging
    ├── go.mod
    └── logging.go
```

أصبح بإمكاننا الآن تشغيل البرنامج `main` من حزمة `cmd` بالأوامر التالية بعد أن أنشأنا جميع ملفات الضبط:

```
$ cd ../cmd
$ go run main.go
```

سنحصل على رسالة تطبع التوقيت وتقول أن هذه تعليمة تنقيح debug:

```
2019-08-28T11:36:09-05:00 This is a debug statement...
```

يطبع البرنامج الوقت الحالي بتنسيق RFC 3339 متبوعًا بالتعليمة المرسلة إلى المُسجّل logger، وقد صُمِّم RFC 3339 لتمثيل الوقت على الإنترنت ومن الشائع استخدامه في ملفات التسجيل والتتبع log files.

بما أن الدالتين `Debug` و `Log` هما دوال مُصدّرة، لذا يمكن استخدامهما في الحزمة `main`، لكن لا يمكن استخدام المتغير `debug` في حزمة `logging`، وإذا حاولت الوصول إليه، فستحصل على خطأ في وقت التصريف compile-time error.

سنضيف السطر `fmt.Println(logging.debug)` إلى main.go:

```
package main
import "github.com/gopherguides/logging"
func main() {
  logging.Debug(true)
  logging.Log("This is a debug statement...")
  fmt.Println(logging.debug)
}
```

احفظ وشغل الملف، إذ ستحصل على الخطأ التالي (لا يمكن الإشارة إلى العنصر debug لأنه غير مُصدّر):

```
. . .
./main.go:10:14: cannot refer to unexported name logging.debug
```

سنتعرّف الآن على كيفية تصدير الحقول والتوابع من داخل السجلات structs بعد أن تعرّفنا على الملفات المُصدّرة وغير المُصدّرة.

13.4 نطاق الرؤية داخل السجلات Structs

تحدّثنا في الأمثلة السابقة عن إمكانية الوصول إلى عنصر ينتمي إلى حزمة مُحددة من حزمة أخرى، كذلك رأينا أنه يمكننا تعديل العناصر التي تكون مُصدّرة. غالبًا ستسير الأمور على ما يُرام، لكن في بعض الحالات قد تظهر لدينا بعض المشاكل. مثلًا، عندما تستخدم أكثر من حزمة متغير وتعدّل إحداها على قيمته، فسيكون التعديل على النسخة نفسها والتي ستنعكس على كل تلك الحزم التي تصل إليه، وبالتالي ستحدث لدينا مشكلة عدم اتساق في البيانات.

تخيل أن تضبط المتغير `Debug` على `true` ويأتي شخص آخر لا تعرفه يستخدم الحزمة نفسها ويضبطه على `false`، لذا سنحل هذه المشكلة من خلال استخدام مفهوم النسخ الذي تتبناه السجلات Structs في جو، وبالتالي كل شخص يستخدِم هذه الحزمة سيأخذ نسخةً مستقلةً منها، وبالتالي سنتجنب مثل هذه المشاكل.

سنعدّل الآن الحزمة `logging` كما يلي:

```
package logging
import (
  "fmt"
  "time"
)
type Logger struct {
  timeFormat string
  debug   bool
}
func New(timeFormat string, debug bool) *Logger {
  return &Logger{
        timeFormat: timeFormat,
        debug:   debug,
  }
}
func (l *Logger) Log(s string) {
  if !l.debug {
        return
  }
  fmt.Printf("%s %s\n", time.Now().Format(l.timeFormat), s)
}
```

أنشأنا هنا السجل `Logger` الذي سيضم العناصر غير المُصدّرة وتنسيق الوقت `timeFormat` المطلوب طباعته والمتغير `debug` وقيمته سواءً كانت `true` أو `false`.

تضبط الدالة `New` الحالة الأولية لإنشاء السجل `logger` داخليًا ضمن المتغيران `timeFormat` و `debug` غير المُصدَّران، كما أنشأنا التابع `Log` بداخل هذا السجل الذي يأخذ المعلومات المراد طباعتها، كما يوجد بداخل التابع `Log` مرجع reference يعود إلى متغير الدالة المحلية `l` الخاص به للحصول على سماحية الوصول مرةً أخرى إلى الحقول الداخلية مثل `l.timeFormat` و `l.debug`.

سيسمح لنا هذا النهج بإنشاء مسجل `Logger` في العديد من الحزم المختلفة واستخدامه باستقلال عن الحزم الأخرى له، ولاستخدامه في حزمة أخرى سنعدّل ملف `cmd/main.go` كما يلي:

```
package main
import (
  "time"
  "github.com/gopherguides/logging"
)
func main() {
  logger := logging.New(time.RFC3339, true)
  logger.Log("This is a debug statement...")
}
```

بعد تشغيل البرنامج ستحصل على الخرج التالي:

```
2019-08-28T11:56:49-05:00 This is a debug statement...
```

أنشأنا في هذا المثال نسخةً من المُسجّل `logger` من خلال استدعاء الدالة المُصدّرة `New`، كما خزّنا مرجعًا لهذه النسخة في متغير المُسجّل `logger`، ويمكننا الآن استدعاء `logging.Log` لطباعة المعلومات.

إذا حاولنا الإشارة إلى حقول غير مُصدّرة من `Logger` مثل الحقل `timeFormat`، فسنحصل على خطأ في وقت التصريف، لذا جرّب أن تضيف السطر `fmt.Println(logger.timeFormat)` إلى الملف `cmd/main.go` كما يلي:

```
package main
import (
  "time"
  "github.com/gopherguides/logging"
)
func main() {
```

```
    logger := logging.New(time.RFC3339, true)
    logger.Log("This is a debug statement...")
    fmt.Println(logger.timeFormat)
}
```

ستحصل على الخطأ التالي (لا يمكن الإشارة إلى حقل أو دالة غير مصدرة):

```
. . .
cmd/main.go:14:20: logger.timeFormat undefined (cannot refer to
unexported field or method timeFormat)
```

يلاحظ المُصرّف أنّ `logger.timeFormat` غير مُصدّر، لذا لا يمكن الوصول له من الحزمة `logging`.

13.5 نطاق الرؤية في التوابع

يمكننا تصدير أو عدم تصدير الدوال بطريقة الحقول ضمن السجلات Structs نفسها، ولتوضيح ذلك سنضيف مستويات التسجيل إلى المسجل logger الخاص بنا، إذ تُعَدّ مستويات التسجيل وسيلةً لتصنيف السجلات logs الخاصة بك، بحيث يمكنك البحث فيها عن أنواع معينة من الأحداث، وتكون المستويات التي سنضعها في المسجل logger كما يلي:

- مستوى المعلومات `info`: تمثّل الأحداث التي تُعلِم المستخدِم بإجراء ما مثل بدء البرنامج `Program started` أو إرسال البريد الإلكتروني `Email sent`، كما تساعدنا في تصحيح الأخطاء وتتبع أجزاء من برنامجنا لمعرفة ما إذا كان السلوك المتوقع قد حدث أم لا.
- مستوى التحذير `warning`: تشير هذه الأحداث إلى حدوث أمر غير متوقع أو قد يُسبب مشاكل لاحقًا، لكنها ليست أخطاءً مثل فشل إرسال البريد الإلكتروني وإعادة محاولة الإرسالة `Email failed to send, retrying`، ويمكن القول أنها تساعدنا في رؤية أجزاء من برنامجنا لا تسير كما هو متوقع لها.
- مستوى الأخطاء `error`: تشير هذه الأحداث إلى حدوث أخطاء أو مشاكل في البرنامج مثل الملف غير موجود`File not found`، فهي أخطاء تؤدي إلى فشل البرنامج وبالتالي إيقافه.

قد ترغب في تفعيل أو إيقاف مستويات معينة من التسجيل، خاصةً إذا كان برنامجك لا يعمل كما هو متوقع له وترغب في تصحيح أخطاء البرنامج، لذا سنضيف وظيفة تعدّل البرنامج بحيث عندما تكون `debug` مضبوطةً على `true`، فستُطبع كل رسائل المستويات؛ أما إذا كانت `false`، فستطبع رسائل الخطأ فقط.

سنضيف مستويات التسجيل إلى الملف `logging/logging.go`:

```
package logging
import (
```

```
    "fmt"
    "strings"
    "time"
)
type Logger struct {
    timeFormat string
    debug   bool
}
func New(timeFormat string, debug bool) *Logger {
    return &Logger{
          timeFormat: timeFormat,
          debug:   debug,
    }
}
func (l *Logger) Log(level string, s string) {
    level = strings.ToLower(level)
    switch level {
    case "info", "warning":
          if l.debug {
                l.write(level, s)
          }
    default:
          l.write(level, s)
    }
}
func (l *Logger) write(level string, s string) {
    fmt.Printf("[%s] %s %s\n", level, time.Now().Format(l.timeFormat),
s)
}
```

لاحظ أننا صرّحنا عن وسيط جديد للتابع `Log` هو `level` بغيـة تحديـد مسـتوى التسـجيل، فـإذا كـان لـدينا رسالة معلومات `info` أو تحـذير `warning` وكـان حقـل `debug` مضـبوطًا على `true`، فسـيكتب الرسـالة، وإلا فسيتجاهلها؛ أما إذا كانت الرسالة هي رسالة خطأ `error`، فسيطبعها دومًا.

يوجد تحديد ما إذا كانت الرسالة ستُطبع أم لا في التابع `Log`، كما عرّفنا أيضًا تابعًا غير مُصدّر يدعى `write` وهو مَن سيطبع الرسالة في نهاية المطاف.

يمكننا الآن استخدام مستويات التسجيل في الحزم الأخرى من خلال تعديل ملـف `cmd/main.go` ليصـبح كما يلي:

```
package main
import (
  "time"
  "github.com/gopherguides/logging"
)
func main() {
  logger := logging.New(time.RFC3339, true)
  logger.Log("info", "starting up service")
  logger.Log("warning", "no tasks found")
  logger.Log("error", "exiting: no work performed")
}
```

بتشغيل الملف ستحصل على الخرج التالي:

```
[info] 2019-09-23T20:53:38Z starting up service
[warning] 2019-09-23T20:53:38Z no tasks found
[error] 2019-09-23T20:53:38Z exiting: no work performed
```

نلاحظ نجاح استخدام التابع `Log`، كمـا يمكننـا تمريـر الوسـيط `level` من خلال ضـبط القيمـة `false` إلى المتغير `debug`:

```
package main
import (
  "time"
  "github.com/gopherguides/logging"
)
func main() {
  logger := logging.New(time.RFC3339, false)
  logger.Log("info", "starting up service")
  logger.Log("warning", "no tasks found")
  logger.Log("error", "exiting: no work performed")
}
```

سنلاحظ أنه سيطبع فقط رسائل مستوى الخطأ (لم يُنفّذ أيّ عمل):

```
[error] 2019-08-28T13:58:52-05:00 exiting: no work performed
```

إذا حاولت استدعاء التابع `write` من خارج الحزمة `logging`، فستحصل على خطأ في وقت التصريف:

```
package main
import (
  "time"
  "github.com/gopherguides/logging"
)
func main() {
  logger := logging.New(time.RFC3339, true)
  logger.Log("info", "starting up service")
  logger.Log("warning", "no tasks found")
  logger.Log("error", "exiting: no work performed")
  logger.write("error", "log this message...")
}
```

يكون الخرج كما يلي:

```
cmd/main.go:16:8: logger.write undefined (cannot refer to unexported
field or method logging.(*Logger).write)
```

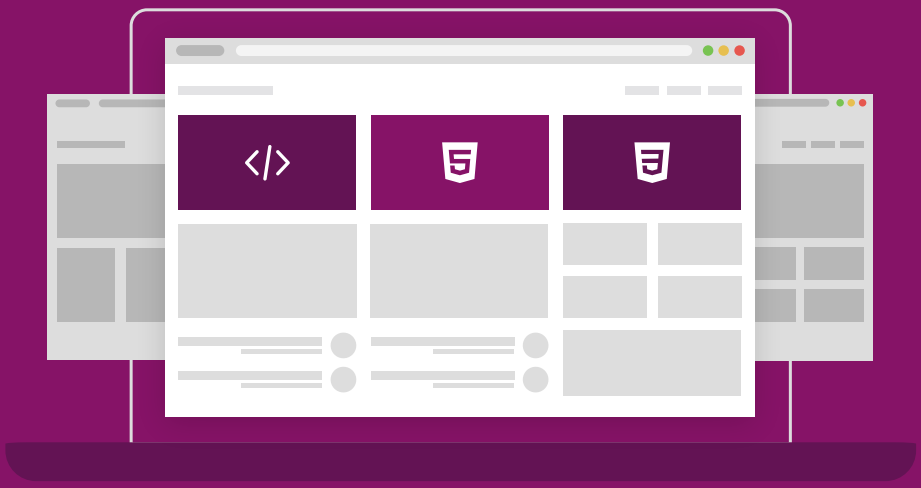
بمجرد أن يلاحظ المُصرّف أنك تحاول الوصول إلى شيء ما أول محرف منه صغير، فإنه يعلم أنه غير مصـدّر وسيرمي خطأ تصريف.

يوضّح المسجل هنا كيف يمكننا كتابة التعليمات البرمجية التي تعرض فقط الأجزاء التي نريد أن تستخدِمها الحزم الأخرى، وبما أننا نتحكم في أجزاء الحزمة التي تظهر خارج الحزمة، فيمكننا لاحقًا إجراء تغييرات دون التـأثير على أيّ شيفرة تعتمد على الحزمة الخاصة بنا، فإذا أردنا مثلًا إيقـاف تشـغيل رسـائل مسـتوى المعلومـات فقـط عندما يكون `debug` مضبوطًا على `false`، فيمكنك إجراء هذا التغيـير دون التـأثير على أي جـزء آخـر من واجهـة برمجة التطبيقات API الخاصة بك، كما يمكننا أيضًا إجراء تغييرات بأمان على رسالة السجل لتضمين المزيد من المعلومات مثل المجلد الذي كان البرنامج يعمل منه.

13.6 الخاتمة

وضح هذا الفصل من كتاب البرمجة بلغة جو كيفية مشاركة الشيفرة بين الحزم المختلفة مع حماية تفاصـيل تنفيذ الحزمة الخاصة بك، إذ يتيح لك ذلك تصدير واجهة برمجة تطبيقات بسيطة نادرًا ما تحصل عليها تغيـيرات للتوافق مع الإصدارات السابقة، بحيث تكون التغييرات الأساسية غالبيتها ضمن ذلك الصـندوق الأسـود، ويُعَـدّ هذا من أفضل الممارسات عند إنشاء الحزم وواجهات برمجة التطبيقات المقابلة لها.

دورة تطوير واجهات المستخدم



ابدأ مسارك المهني كمطور واجهات المواقع والمتاجر الإلكترونية
فور انتهائك من الدورة

التحق بالدورة الآن

14. كتابة التعليمات الشرطية If

لا تخلو أية لغة برمجية من التعليمات الشرطية Conditional Statements التي تُنفَّذ بناءً على تحقق شرط معيّن، وهي عبارة عن تعليمات برمجية يمكنها التحكم في تنفيذ شفرات معينة بحسب تحقق شرط ما من عدمه في وقت التنفيذ، وباستخدام التعليمات الشرطية يمكن للبرامج التحقق من استيفاء شروط معينة ومن ثم تنفيذ الشيفرة المقابلة.

تُنفّذ تعليمات البرامج الحاسوبية في الحالة الطبيعية سطرًا سطرًا ومن الأعلى إلى الأسفل؛ أما باستخدام التعليمات الشرطية، فيمكن للبرامج التحقق من استيفاء شروط معينة ومن ثم تنفيذ الشيفرة المقابلة وكسر تسلسل التنفيذ هذا أو تجاوز كتلة من التعليمات، أي باختصار تمكننا التعليمات الشرطية من التحكم بسير عمل البرنامج، وهذه بعض الأمثلة التي سنستخدِم فيها التعليمات الشرطية:

- إذا حصل الطالب على أكثر من 65% في الامتحان، فأعلن عن نجاحه؛ وإلا، فأعلن عن رسوبه.
- إذا كان لدى العميل مال في حسابه، اقتطع منه قيمة الفاتورة، وإلا، فاحسب غرامة.
- إذا اشترى الزبون 10 برتقالات أو أكثر، فاحسب خصمًا بمقدار 5%؛ وإلا، فلا تفعل.

تقيِّم الشيفرة الشرطية شروطًا ثم تُنفِّذ شيفرةً بناءً على ما إذا تحققت تلك الشروط أم لا، وستتعلم في هذا الفصل كيفية كتابة التعليمات الشرطية في لغة جو.

14.1 التعليمة if

سنبدأ بالتعليمة if والتي تتحقق مما إذا كان شرطًا محدَّدًا محقّقًا `true` أم لا `false`، ففي حال تحقق الشرط، فستُنفَّذ الشيفرة المقابلة، وإلا فسيتابع البرنامج في مساره الطبيعي، ولنبدأ بأمثلة عملية توضح ذلك، لذا افتح ملفًا واكتب الشيفرة التالية:

```
package main
import "fmt"
func main() {
  grade := 70
  if grade >= 65 {
        fmt.Println("Passing grade")
  }
}
```

أعطينا للمتغير `grade` القيمة الصحيحة 70 ثم استخدمنا التعليمة `if` لتقييم ما إذا كان أكبر من أو يساوي 65 وفي تلك الحالة سيطبع البرنامج السلسلة النصية التالية:

```
Passing grade.
```

احفظ البرنامج بالاسم grade.go ثم نفّذه في بيئة البرمجة المحلية من نافذة الطرفية باستخدام الأمر `go run grade.go`، وفي هذه الحالة تلبي الدرجة 70 تلبي لأنها أكبر من 65، لذلك ستحصل على الخرج التالي عند تنفيذ البرنامج:

```
Passing grade
```

لنغيّر الآن نتيجة هذا البرنامج عبر تغيير قيمة المتغير `grade` إلى 60:

```
package main
import "fmt"
func main() {
  grade := 60
  if grade >= 65 {
        fmt.Println("Passing grade")
  }
}
```

لن نحصل على أي خرج بعد حفظ وتنفيذ الشيفرة لأنّ الشرط لم يتحقق ولم نطلب من البرنامج تنفيذ تعليمة أخرى.

لنأخذ مثالًا آخرًا، إذا كنا نريد التحقق فيما إذا كان رصيد الحساب المصرفي أقل من 0، سننشئ ملفًا باسم account.go ونكتب فيه البرنامج التالي:

```
package main
import "fmt"
```

```
func main() {
  balance := -5
  if balance < 0 {
        fmt.Println("Balance is below 0, add funds now or you will be
charged a penalty.")
  }
}
```

سنحصل على الخرج التالي عند تنفيذ البرنامج باستخدام الأمر `go run account.go`:

```
Balance is below 0, add funds now or you will be charged a penalty.
```

أعطينا للمتغير `balance` القيمة 5- وهي أقل من 0 في البرنامج السابق، ولمَّا كان الرصيد مستوفيًا لشرط التعليمة `if` أي `balance < 0`، فسنحصل على سلسلة نصية في الخرج بمجرد حفظ الشيفرة وتنفيذها، وإذا غيّرنا الرصيد إلى القيمة 0 أو إلى عدد موجب مرةً أخرى، فلن نحصل على أيّ خرج.

14.2 التعليمة else

قد تريد من البرنامج فعل شيء ما في حال عدم تحقق شرط التعليمة if، ففي المثال أعلاه نريد طباعة خـرج في حال النجاح والرسوب، ولفعل ذلك سنضيف التعليمة else إلى شرط الدرجة أعلاه بالصياغة التالية:

```
package main
import "fmt"
func main() {
  grade := 60
  if grade >= 65 {
        fmt.Println("Passing grade")
  } else {
        fmt.Println("Failing grade")
  }
}
```

تساوي قيمة المتغير `grade` العدد 60، لذلك فشرط التعليمة `if` غير متحقق، وبالتـالي لن يطبـع البرنـامج درجة النجاح، إذ تخبر التعليمة `else` البرنامج أنه عليه طباعة السلسلة النصية Failing grade، لذا عندما نحفظ البرنامج وننفّذه، فسنحصل على الخرج التالي:

```
Failing grade
```

إذا عدّلنا البرنامج وأعطينا المتغير `grade` القيمة 65 أو أعلى منها، فسنحصل بدلًا من ذلك على:

```
Passing grade
```

لإضافة التعليمة else إلى مثال الحساب المصرفي، سنعيد كتابة الشيفرة كما يلي:

```
package main
import "fmt"
func main() {
  balance := 522
  if balance < 0 {
        fmt.Println("Balance is below 0, add funds now or you will be
charged a penalty.")
  } else {
        fmt.Println("Your balance is 0 or above.")
  }
}
```

سنحصل على الخرج التالي:

```
Your balance is 0 or above.
```

غيّرنا هنا قيمة المتغير balance إلى عدد موجب لكي تُنفَّذ الشيفرة المقابلة للتعليمة else، فإذا أردت تنفيذ الشيفرة المقابلة للتعليمة if، فغيِّر القيمة إلى عدد سالب.

من خلال دمج العبارتين if و else، فأنت تنشئ تعليمة شرطية مزدوجة والتي ستجعل الحاسوب ينفذ شيفرة برمجية معينّة سواءً تحقق شرط if أم لا.

14.3 التعليمة else if

عملنا حتى الآن على تعليمات شرطية ثنائية، أي إذا تحقق الشرط، فنفِّذ شيفرةً ما، وإلا، فنفِّذ شيفرةً أخرى فقط، لكن قد تريد في بعض الحالات برنامجًا يتحقق من عدة حالات شرطية، ولأجل هذا سنستخدِم التعليمة else if، إذ تشبه هذه التعليمة تعليمة if ومهمتها التحقق من شرط إضافي.

قد نرغب في برنامج الحساب المصرفي بالحصول على ثلاثة مخرجات مختلفة مقابلة لثلاث حالات مختلفة:

- الرصيد أقل من 0.
- الرصيد يساوي 0.
- الرصيد أعلى من 0.

لذا ستوضع التعليمة `else if` بين التعليمة `if` والتعليمة `else` كما يلي:

```
package main
import "fmt"
func main() {
  balance := 522
  if balance < 0 {
        fmt.Println("Balance is below 0, add funds now or you will be
charged a penalty.")
  } else if balance == 0 {
        fmt.Println("Balance is equal to 0, add funds soon.")
  } else {
        fmt.Println("Your balance is 0 or above.")
  }
}
```

هناك الآن ثلاثة مخرجات محتملة يمكن أن تُطبع عند تنفيذ البرنامج وهي:

- إذا كان المتغير `balance` يساوي 0، فسنحصل على الخرج من التعليمة `else if` (أي الرصيد يساوي 0، أضف مبلغًا قريبًا).
- إذا ضُبِط المتغير `balance` على عدد موجب، فسنحصل على المخرجات من التعليمة `else` (أي رصيدك أكبر من 0).
- إذا ضُبِط المتغير `balance` على عدد سالب، فسنحصل على المخرجات من التعليمة `if` (أي الحساب فارغ، أضف مبلغا الآن أو ستحصل على غرامة).

إذا أردنا أن نأخذ بالحسبان أكثر من ثلاثة احتمالات، فيمكننا كتابة عدة تعليمات `else if` في الشيفرة البرمجية، ولنُعِد الآن كتابة البرنامج grade.go بحيث يقابل كل نطاق من الدرجات العددية درجة حرفية محددة:

- الدرجة 90 أو أعلى تكافئ الدرجة A.
- من 80 إلى 89 تكافئ الدرجة B.
- من 70 إلى 79 تكافئ الدرجة C.
- من 65 إلى 69 تكافئ الدرجة D.
- الدرجة 64 أو أقل تكافئ الدرجة F.

سنحتاج لتنفيذ هذه الشيفرة إلى تعليمة `if` واحدة وثلاث تعليمات `else if` وتعليمة `else` تعالج جميع الحالات الأخرى، لذا دعنا نعيد كتابة الشيفرة من المثال أعلاه لطباعة سلسلة نصية مقابلة لكل علامة، إذ يمكننا الإبقاء على التعليمة `else` كما هي.

```
package main
import "fmt"
func main() {
  grade := 60
  if grade >= 90 {
        fmt.Println("A grade")
  } else if grade >= 80 {
        fmt.Println("B grade")
  } else if grade >= 70 {
        fmt.Println("C grade")
  } else if grade >= 65 {
        fmt.Println("D grade")
  } else {
        fmt.Println("Failing grade")
  }
}
```

تُنفّذ تعليمات `else if` بالترتيب، وسيكمل هذا البرنامج الخطوات التالية:

- إذا كانت الدرجة أكبر من 90، فسيطبع البرنامج الدرجة A، وإذا كانت الدرجة أقل من 90، فسيستمر البرنامج إلى التعليمة التالية.
- إذا كانت الدرجة أكبر من أو تساوي 80، فسيطبع البرنامج الدرجة B، إذا كانت الدرجة تساوي 79 أو أقل، فسيستمر البرنامج إلى التعليمة التالية.
- إذا كانت الدرجة أكبر من أو تساوي 70، فسيطبعُ البرنامجُ الدرجة C، إذا كانت الدرجة تساوي 69 أو أقل، فسيستمر البرنامج إلى التعليمة التالية.
- إذا كانت الدرجة أكبر من أو تساوي 65، فسيطبع البرنامج الدرجة D، وإذا كانت الدرجة تساوي 64 أو أقل، فسيستمر البرنامج إلى التعليمة التالية.
- أخيرًا، سيطبع البرنامج السلسة النصية Failing grade لأنه لم تستوفى أي من الشروط المذكورة أعلاه.

14.4 تعليمات if المتداخلة

يمكنك الانتقال إلى التعليمات الشرطية المتداخلة بعد أن تتعود على التعليمات `if` و `else if` و `else`، إذ يمكننا استخدام تعليمات `if` المتداخلة في الحالات التي نريد فيها التحقق من شرط ثانوي بعد التأكد من تحقق الشرط الرئيسي، وبالتالي يمكننا حشر تعليمة `if-else` داخل تعليمة `if-else` أخرى، ولنلقِ نظرةً على صياغة `if` المتداخلة:

```
if statement1 { // تعليمة if الخارجية
   fmt.Println("true")
   if nested_statement { // تعليمة if المتداخلة
         fmt.Println("yes")
   } else { // تعليمة else المتداخلة
         fmt.Println("no")
   }
} else { // تعليمة else الخارجية
   fmt.Println("false")
}
```

هناك عدة مخرجات محتملة لهذه الشيفرة:

- إذا كانت التعليمة `statement1` صحيحةً، فسيتحقق البرنامج مما إذا كانت `nested_statement` صحيحةً أيضًا، فإذا كانت كلتا الحالتين صحيحتين، فسنحصل على المخرجات التالية:

```
true
yes
```

- ولكن إذا كانت `statement1` صحيحةً و `nested_statement` خاطئةً، فسنحصل في هذه الحالة على المخرجات التالية:

```
true
no
```

- وإذا كانت `statement1` خاطئةً، فلن تُنفّذ تعليمة `if-else` المتداخلة على أيّ حال، لذلك ستُنفّذ التعليمة `else` وحدها وستكون المخرجات كما يلي:

```
false
```

يمكن أيضًا استخدام عدة تعليمات `if` متداخلة في الشيفرة:

```
if statement1 { // الخارجية if
  fmt.Println("hello world")
  if nested_statement1 { // المتداخلة الأولى if
        fmt.Println("yes")
  } else if nested_statement2 { // المتداخلة الأولى else if
        fmt.Println("maybe")
  } else { // المتداخلة الأولى else
        fmt.Println("no")
  }
} else if statement2 { // الخارجية else if
  fmt.Println("hello galaxy")
  if nested_statement3 { // المتداخلة الثانية if
        fmt.Println("yes")
  } else if nested_statement4 { // المتداخلة الثانية else if
        fmt.Println("maybe")
  } else { // المتداخلة الثانية else
        fmt.Println("no")
  }
} else { // الخارجية else
  statement("hello universe")
}
```

توجد في الشيفرة البرمجية أعلاه تعليمات `if` و `else if` متداخلة داخل كل تعليمات `if`، إذ سيفسح هذا مجالًا لمزيد من الخيارات في كل حالة.

دعنا نلقي نظرةً على مثال لتعليمات `if` متداخلة في البرنامج grade.go، إذ يمكننا التحقق أولًا مما إذا كان الطالب قد حقق درجة النجاح (أكبر من أو تساوي 65%) ثم نحدد العلامة المقابلة للدرجة، فإذا لم يحقق الطالب درجة النجاح، فلا داعي للبحث عن العلامة المقابلة للدرجة، وبدلًا من ذلك، يمكن أن نجعل البرنامج يطبع سلسلة نصية فيها إعلان عن رسوب الطالب، وبالتالي ستبدو الشيفرة المعدلة كما يلي:

```
package main
import "fmt"
func main() {
  grade := 92
  if grade >= 65 {
        fmt.Print("Passing grade of: ")
        if grade >= 90 {
```

```
            fmt.Println("A")
        } else if grade >= 80 {
            fmt.Println("B")
        } else if grade >= 70 {
            fmt.Println("C")
        } else if grade >= 65 {
            fmt.Println("D")
        }
    } else {
        fmt.Println("Failing grade")
    }
}
```

إذا أعطينا للمتغير `grade` قيمة 92، سيتحقق الشرط الأول ويطبع البرنامج العبارة "Passing grade of:"، بعد ذلك سيتحقق مما إذا كانت الدرجة أكبر من أو تساوي 90، وبما أنّ هذا الشرط محقق أيضًا، فستُطبع A؛ أما إذا أعطينا للمتغير `grade` القيمة 60، فلن يتحقق الشرط الأول، لذلك سيتخطى البرنامج تعليمات `if` المتداخلة وينتقل إلى التعليمة `else` ويطبع "Failing grade".

يمكننا بالطبع إضافة المزيد من الخيارات واستخدام طبقة ثانية من تعليمات `if` المتداخلة، فربما نودّ إضافة الدرجات التفصيلية A+ و A و A-، إذ يمكننا إجراء ذلك عن طريق التحقق أولًا من اجتياز درجة النجاح ثم التحقق مما إذا كانت الدرجة تساوي 90 أو أعلى ثم التحقق مما إذا كانت الدرجة تتجاوز 96، وفي تلك الحالة ستقابل العلامة A+، وإليك المثال التالي:

```
...
if grade >= 65 {
    fmt.Print("Passing grade of: ")
    if grade >= 90 {
        if grade > 96 {
            fmt.Println("A+")
        } else if grade > 93 && grade <= 96 {
            fmt.Println("A")
        } else {
            fmt.Println("A-")
        }
...
```

سيؤدي البرنامج السابق ما يلي في حال تعيين المتغير `grade` على القيمة 96:

- التحقق مما إذا كانت الدرجة أكبر من أو تساوي 65 (صحيح).
- طباعة :Passing grade of
- التحقق مما إذا كانت الدرجة أكبر من أو تساوي 90 (صحيح).
- التحقق مما إذا كانت الدرجة أكبر من 96 (خطأ).
- التحقق مما إذا كانت الدرجة أكبر من 93 وأقل من أو تساوي 96 (صحيح).
- طباعة A.
- تجاوز التعليمات الشرطية المتداخلة وتنفيذ باقي الشيفرة.

سيكون خرج البرنامج إذا كانت الدرجة تساوي 96 كما يلي:

```
Passing grade of: A
```

تساعد تعليمات `if` المتداخلة على إضافة عدة مستويات من الشروط الفرعية إلى الشيفرة.

14.5 الخاتمة

تعرفت في هذا الفصل على طريقة التحكم في مسار عمل البرنامج أي تدفق تنفيذ الشيفرة باستخدام التعليمات الشرطية، إذ تطلب التعليمات الشرطية من البرنامج التحقق من استيفاء شرط معيّن من عدمه، فإذا استُوفي الشرط، فستُنفّذ شيفرة معينة،وإلا فسيستمر البرنامج وينتقل إلى الأسطر التالية.

هل تريد كتابة سيرة ذاتية احترافية؟

نساعدك في إنشاء سيرة ذاتية احترافية عبر خبراء توظيف
مختصين في أكبر منصة توظيف عربية عن بعد

أنشئ سيرتك الذاتية الآن

15. التعامل مع تعليمة التبديل Switch

تمنح التعليمات الشرطية المبرمجين القـدرة على التحكم بسـير عمـل بـرامجهم وتفريعهـا وتوجيههـا لاتخـاذ بعض الإجراءات إذا كان الشرط المحدَّد صحيحًا وإجراء آخر إذا كان الشرط خاطئًا.

تتوفَّر عدة تعليمات للتفريع branching statements بلغة جو، فقد تناولنا في الفصل السابق كلًا من تعليمة `if` و `else` و `else if` والتعليمات الشرطية المتداخلة، وسننتقل الآن إلى تعليمة تفريع أخرى تدعى `switch` والتي يُعدّ استخدامها أقل شيوعًا من التعليمات السابقة، ولكن مع ذلك فهي مفيدة أحيانًا للتعبير عن نوع معيّن من التفريع المُتعدِّد multiway branches.

على سبيل المثـال، نستخدم التفريع المتعدد عندما نريد مقارنة قيمة متغير (أو دخل من المستخدِم) بعـدة قيم محتملة واتخاذ إجراء محدد بناءً على ذلك، وأفضل مثال عملي على ذلك هو برنامج الآلة الحاسبة الذي يُجـد ناتج عملية حسابية لعددين، بحيث يكون الدخل هو عملية حسابية وهذا يكافئ عدة قيم محتملة `+` أو `-` أو `*` أو `/` وعددين `x` و `y`، وسيعتمد الإجراء المُتخذ هنا على نوع العملية التي يُدخلهـا المستخـدِم، فإذا أدخـل `+` فسيكـون الإجراء `x+y`، وإذا أدخل `-` فسيكون `x-y`، وهكذا.

يمكن تحقيق التفريع المتعدد باستخدام التعليمات الشرطية `if`، كما يمكن تحقيقه أيضًا باستخدام تعليمـة `switch` كما سنشرح في هذا الفصل.

15.1 بنية التعليمة Switch

عادةً ما تُستخدَم تعليمة التبديل `switch` لوصف الحالات التي يكون لدينا فيها عدة إجـراءات ممكنـة تبعًـا لقيم متغير أو عدة متغيرات مُحتمَلة، ويوضح المثال التالي كيف يمكننا تحقيق ذلك باستخدام تعليمات `if`:

```
package main
```

```
import "fmt"
func main() {
  flavors := []string{"chocolate", "vanilla", "strawberry", "banana"}
  for _, flav := range flavors {
        if flav == "strawberry" {
              fmt.Println(flav, "is my favorite!")
              continue
        }
        if flav == "vanilla" {
              fmt.Println(flav, "is great!")
              continue
        }
        if flav == "chocolate" {
              fmt.Println(flav, "is great!")
              continue
        }
        fmt.Println("I've never tried", flav, "before")
  }
}
```

سيكون الخرج كما يلي:

```
chocolate is great!
vanilla is great!
strawberry is my favorite!
I've never tried banana before
```

نُعرّف بداخل الدالة `main` شريحةً slice من نكهات الآيس كريم ثم نستخدِم حلقة `for` للتكرار على عناصرها ثم نستخدِم ثلاث تعليمات `if` لطباعة رسائل مختلفة تشير إلى تفضيلات نكهات الآيس كريم المختلفة، إذ يجب على كل تعليمة `if` أن تتضمن تعليمة `Continue` لإيقاف التكرار الحالي في الحلقة لكي لا تُطبَع الرسالة الافتراضية في نهاية كتلة الحلقة.

لاحظ أنه كلما أضفنا نكهةً جديدةً إلى الشريحة السابقة، فسيتعين علينا إضافة التفضيل المقابل لها، وبالتالي الاستمرار في كتابة تعليمات `if` في كل مرة نضيف فيها عنصرًا جديدًا إلى الشريحة، كما يجب أيضًا تكرار تعليمات `if` مع الرسائل المكررة كما في حالة "Vanilla" و "chocolate".

كما ترى فإن تكرار تعليمات `if` باستمرار ووجود رسالة افتراضية في نهاية كتلة الحلقة بدون تعليمة `if` يجعل الأمر غريبًا وغير مرتب تمامًا، ولاحظ أيضًا أنّ كل ما يحدث هو مقارنة المتغير بقيم متعددة واتخاذ

إجراءات مختلفة بناءً على ذلك، وهنا تكون بنية switch قادرةً على التعبير عمّا يحدث بطريقة أفضل وأكثر تنظيمًا ورتابةً.

تبدأ تعليمة التبديل بالكلمة المفتاحية switch متبوعةً بمتغير أو عدة متغيرات لإجراء مقارنات على أساسها (أبسط شكل) متبوعةً بقوسين معقوصَين توضَع ضمنها الحالات المتعددة التي يمكن أن يحملها المتغير أو المتغيرات، وكل من هذه الحالات تبدأ بالكلمة المفتاحية case متبوعةً بقيمة محتملة للمتغير، بحيث تقابل كل حالة إجراءً مُحددًا يُنفَّذ في حال كانت قيمة المتغير تطابق القيمة المحتملة، وسيبدو الأمر أوضح في المثال التالي الذي يكافئ المثال السابق تمامًا:

```
package main
import "fmt"
func main() {
  flavors := []string{"chocolate", "vanilla", "strawberry", "banana"}
  for _, flav := range flavors {
        switch flav {
        case "strawberry":
              fmt.Println(flav, "is my favorite!")
        case "vanilla", "chocolate":
              fmt.Println(flav, "is great!")
        default:
              fmt.Println("I've never tried", flav, "before")
        }
  }
}
```

يكون الخرج كما يلي:

```
chocolate is great!
vanilla is great!
strawberry is my favorite!
I've never tried banana before
```

نُعرّف بداخل الدالة main كما في المرة السابقة شريحةً slice من نكهات الآيس كريم ثم نستخدِم حلقة for للتكرار على عناصرها، لكن سنستخدِم في هذه المرة تعليمة التبديل وسنلغي استخدام التعليمة if، إذ وضعنا المتغير flav بعد الكلمة switch وستُختبر قيمة هذا المتغير وستُنفّذ إجراءات محددة بناءً عليها:

- الحالة الأولى سيطبع رسالةً محددةً عندما تكون قيمة هذا المتغير تساوي "strawberry".

- الحالة الثانية سيطبع رسالةً محددةً عندما تكون قيمة هذا المتغير تساوي "vanilla" أو "chocolate"، ولاحظ أنه في المثال السابق اضطررنا لكتابة تعليمتَي `if`.
- الحالة الأخيرة هي الحالة الافتراضية وتسمى `default` حيث أنه إذا لم تتحقق أيّ من الحالات السابقة، فستُنفّذ هذه التعليمة ويطبع الرسالة المحددة، لكن إذا تحققت حالة ما من الحالات السابقة، فلن تُنفّذ هذه التعليمة إطلاقًا.

ملاحظة: لم نحتج إلى استخدام `continue` لأن تعليمة التبديل لا تُنفّذ إلا حالةً واحدةً فقط تلقائيًا.

يعرض هذا المثال الاستخدام الأكثر شيوعًا لتعليمة التبديل وهو مقارنة قيمة متغير بعدة قيم محتملة، إذ توفِّر لنا تعليمة التبديل الراحة عندما نريد اتخاذ الإجراء نفسه لعدة قيم مختلفة وتنفيذ إجراءات محددة عندما لا تُستوفَى أيّ من شروط الحالات المُعرَّفة.

15.2 تعليمات التبديل العامة

تُعَدّ تعليمة التبديل مفيدةً في تجميع مجموعات من الشروط الأكثر تعقيدًا لإظهار أنها مرتبطة بطريقة ما، وغالبًا ما يُستخدَم ذلك عند مقارنة متغير أو عدة متغيرات بمجال من القيم بدلًا من القيم المحددة كما في المثال السابق، ويُعَدّ المثال التالي تحقيقًا للعبة تخمين باستخدام تعليمات `if` التي يمكن أن تستفيد من تعليمة التبديل كما رأينا سابقًا:

```
package main
import (
  "fmt"
  "math/rand"
  "time"
)
func main() {
  rand.Seed(time.Now().UnixNano())
  target := rand.Intn(100)
  for {
        var guess int
        fmt.Print("Enter a guess: ")
        _, err := fmt.Scanf("%d", &guess)
        if err != nil {
              fmt.Println("Invalid guess: err:", err)
              continue
        }
```

```
            if guess > target {
                  fmt.Println("Too high!")
                  continue
            }
            if guess < target {
                  fmt.Println("Too low!")
                  continue
            }
            fmt.Println("You win!")
            break
      }
}
```

سيختلف الخرج بناءً على العدد العشوائي المحدد ومدى جودة لعبك، وفيما يلي ناتج دورة لعب واحدة:

```
Enter a guess: 10
Too low!
Enter a guess: 15
Too low!
Enter a guess: 18
Too high!
Enter a guess: 17
You win!
```

تحتاج لعبة التخمين إلى عدد عشوائي لمقارنة التخمينات به، لذا سنستخدِم الدالة `rand.Intn` من الحزمة `math/rand` لتوليده، كما سنستخدِم `rand.Seed` مولّد الأعداد العشوائية بناءً على الوقت الحالي لضمان حصولنا على قيم مختلفة للمتغير `target` في كل مرة نلعب فيها، ولاحظ أننا مررنا القيمة 100 إلى الدالة `rand.Intn` وبالتالي ستكون الأعداد المولَّدة ضمن المجال من 0 إلى 100، وأخيرًا سنستخدِم حلقة `for` لبدء جمع التخمينات من اللاعب.

تعطينا الدالة `fmt.Scanf` إمكانية قراءة مدخلات المستخدِم وتخزينها في متغير من اختيارنا، وهنا نريد تخزين القيمة المُدخَلة في المتغير `guess`، كما نريد أن تكون قيمة هذا المتغير من النوع الصحيح `int`، لذا سنمرر لهذه الدالة العنصر النائب `%d` على أساس وسيط أول والذي يشير إلى وجود قيمة من النوع الصحيح، ويكون الوسيط الثاني هو عنوان المتغير `guess` الذي نريد حفظ القيمة المدخَلة به.

نختبر بعد ذلك فيما إذا كان هناك دخل خاطئ من المستخدِم مثل إدخال نص بدلًا من عدد صحيح ثم نكتب تعليمتَي `if` بحيث تختبر الأولى فيما إذا كان التخمين الذي أدخله المستخدِم أكبر من قيمة العدد المولَّد

ليطبع Too high! وتختبر الثانية فيما إذا كان العدد أصغر ليطبع Too low!، وسيكون التخمين في الحالتين خاطئًا، أي القيمة المولّدة لا تتطابق مع التخمين وبالتالي يخسر، طبعًا لا ننسى كتابة تعليمة continue بعد كل تعليمة if كما ذكرنا سابقًا، وأخيرًا إذا لم تتحقق أي من الشروط السابقة فإنه يطبع You win! دلالةً إلى أنّ تخمينه صحيح وتتوقف الحلقة لوجود تعليمة break؛ أما إذا لم يكن تخمينه صحيح، فستتكرر الحلقة مرةً أخرى.

لاحظ أنّ استخدام تعليمات if يحجب حقيقة أن مجال القيم التي يُقارن معها المتغير مرتبطة كلها بطريقة ما، كما أنه قد يكون من الصعب معرفة ما إذا كنا قد فاتنا جزء من المجال، وفي المثال التالي سنعدّل المثال السابق باستخدام تعليمة التبديل:

```
package main
import (
  "fmt"
  "math/rand"
)
func main() {
  target := rand.Intn(100)
  for {
        var guess int
        fmt.Print("Enter a guess: ")
        _, err := fmt.Scanf("%d", &guess)
        if err != nil {
              fmt.Println("Invalid guess: err:", err)
              continue
        }
        switch {
        case guess > target:
              fmt.Println("Too high!")
        case guess < target:
              fmt.Println("Too low!")
        default:
              fmt.Println("You win!")
              return
        }
  }
}
```

سيكون الخرج مُشابهًا لما يلي:

```
Enter a guess: 25
Too low!
Enter a guess: 28
Too high!
Enter a guess: 27
You win!
```

استبدلنا هنا كل تعليمات `if` بتعليمة التبديل `switch`، ولاحظ أننا لم نذكر بعد تعليمة التبديل أيّ متغير كما فعلنا في أول مثال من الفصل لأن هدفنا هو تجميع الشروط معًا كما ذكرنا، وتتضمن تعليمة التبديل في هذا المثال ثلاث حالات:

- الحالة الأولى عندما تكون قيمة المتغير `guess` أكبر من `target`.
- الحالة الثانية عندما تكون قيمة المتغير `guess` أصغر من `target`.
- الحالة الأخيرة هي الحالة الافتراضية `default`، حيث أنه إذا لم تتحقق أي من الحالات السابقة أي أن `guess` يساوي `target` ستُنفَّذ هذه التعليمة ويطبع الرسالة المُحددة، لكن إذا تحققت حالة ما من الحالات السابقة فلن تُنفّذ هذه التعليمة على الإطلاق، إذًا تُشير هذه الحالة إلى أن التخمين يطابق القيمة المولّدة.

> لا حاجة إلى استخدام `continue` لأن تعليمة التبديل لا تُنفّذ إلا حالة واحدة فقط تلقائيًا.

نلاحظ في الأمثلة السابقة أن حالة واحدة ستُنفّذ، لكن قد تحتاج أحيانًا إلى دمج سلوك عدة حالات معًا، لذا توفِّر تعليمة التبديل كلمةً مفتاحيةً أخرى لإنجاز هذا السلوك.

15.3 التعليمة fallthrough

ربما ترغب بإعادة تنفيذ الشيفرة الموجودة ضمن حالة أخرى، وهنا يمكنك أن تطلب من مترجم لغة جو أن يُشغّل الشيفرة التي تتضمنها الحالة التالية من خلال وضع التعليمة `fallthrough` في نهاية شيفرة الحالة الحالية، وفي المثال التالي سنُعدِّل المثال الذي عرضناه في بداية الفصل والمتعلق بنكهات الآيس كريم لكي نوضّح كيف يمكننا استخدام هذه التعليمة فيه:

```
package main
import "fmt"
func main() {
  flavors := []string{"chocolate", "vanilla", "strawberry", "banana"}
  for _, flav := range flavors {
        switch flav {
```

```
            case "strawberry":
                  fmt.Println(flav, "is my favorite!")
                  fallthrough
            case "vanilla", "chocolate":
                  fmt.Println(flav, "is great!")
            default:
                  fmt.Println("I've never tried", flav, "before")
            }
      }
}
```

سيكون الخرج كما يلي:

```
chocolate is great!
vanilla is great!
strawberry is my favorite!
strawberry is great!
I've never tried banana before
```

نُعرّف داخل الدالة `main` شريحةً slice من نكهات الآيس كريم ثم نستخدِم حلقة `for` للتكرار على عناصرها ثم نستخدِم تعليمة التبديل ونضع المتغير `flav` بعد الكلمة `switch` بحيث تُختبر قيمته وتُنفَّذ إجراءات محددة بناءً عليها:

- الحالة الأولى سيطبع رسالةً محددةً عندما تكون قيمة هذا المتغير تساوي vanilla أو chocolate.
- الحالة الثانية سيطبع strawberry is my favorite! عندما تكون قيمة هذا المتغير تساوي strawberry ثم سيُنفّذ التعليمة `fallthrough` التي تؤدي إلى تنفيذ شيفرة الحالة التالية لها، وبالتالي سيطبع strawberry is great! أيضًا.
- الحالة الأخيرة هي الحالة الافتراضية `default`.

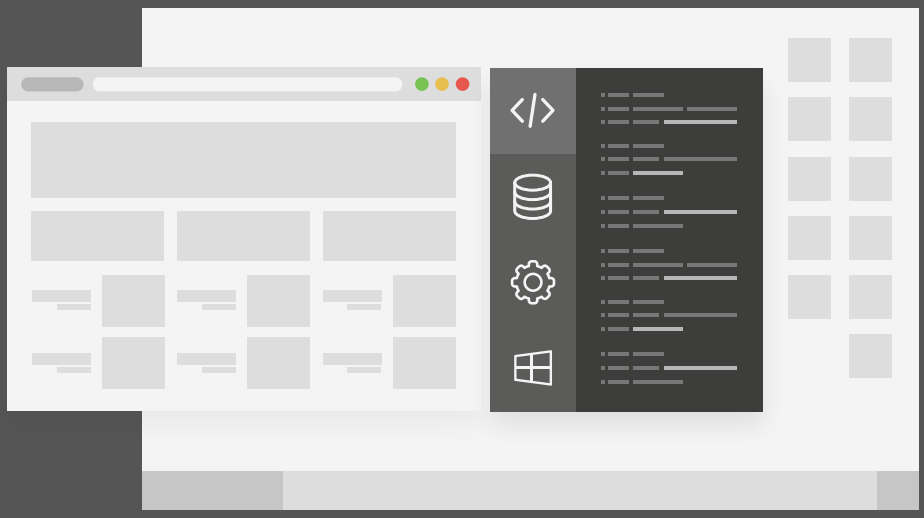
لا يستخدِم المطورون تعليمة `fallthrough` في لغة جو كثيرًا لأنه من الممكن الاستغناء عنها بتعريف دالة تؤدي الغرض واستدعائها ببساطة، وعمومًا لا يُنصَح باستخدامها.

15.4 الخاتمة

تعرفنا في هذا الفصل على تعليمة التبديل `switch` التي تساعدنا في التعبير عن ارتباط مجموعة من عمليات المقارنة ببعضها بطريقة ما، وهذا مهم لجعل المطورين الذين يقرؤون الشيفرة يفهمون هذا الارتباط، والتي تُسهّل عملية إضافة سلوك جديد وحالات جديدة لاحقًا للكود البرمجي عندما تدعو الحاجة إلى ذلك

وتضمن لك دومًا وجود تعليمة افتراضية تُنفَّذ في حال نسيت كتابة حالة من الحالات أو عند عدم تحقق أية حالة من الحالات، لذا جرّب استخدام تعليمة التبديل التي تغنيك عن كتابة عدة تعليمات `if`، وتجعل الكود أسهل بكثير وأكثر قابليةً لإعادة الاستخدام والتصحيح والتعديل.

دورة علوم الحاسوب



دورة تدريبية متكاملة تضعك على بوابة الاحتراف
في تعلم أساسيات البرمجة وعلوم الحاسوب

التحق بالدورة الآن

16. التعامل مع حلقة التكرار For

يسمح لنا استخدام حلقات التكرار في برمجة الحاسوب بأتمتة وتكرار المهام المتشابهة مرات عدة ريثما يتحقق شرط معيّن أو ظهور حدث محدد، فتخيل إذا كان لديك قائمة بالملفات التي تحتاج إلى معالجتها أو إذا كنت تريد حساب عدد الأسطر في المقالة أو إدخال 10 أعداد أو أسماء من المُستخدِم، فيمكنك استخدام حلقة في التعليمات البرمجية الخاصة بك لحل هذه الأنواع من المشاكل.

سنشرح في هذا الفصل كيفية استخدام حلقة `for` في لغة جو، وهي الحلقة الوحيدة المتوفرة فيها على عكس اللغات الأخرى التي تتضمن عدة أنواع من الحلقات مثل حلقة `while` أو `do`، فالهدف من وجود نوع حلقة واحدة هو التبسيط وتجنب الارتباك بين المبرمجين من الأنواع المختلفة، إذ نكتفي بواحدة فقط بدلًا من وجود عدة أنواع للحلقات تحقق الغرض نفسه وبالتكلفة نفسها.

تؤدي حلقة `for` إلى تكرار تنفيذ جزء من الشيفرات بناءً على عدّاد أو على متغير، وهذا يعني أنّ حلقات `for` تستعمل عندما يكون عدد مرات تنفيذ حلقة التكرار معلومًا قبل الدخول في الحلقة، لكن في حقيقة الأمر يمكنك استخدامها حتى إذا لم يكن عدد التكرارات معلومًا من خلال استخدام تعليمة `break` كما سترى لاحقًا.

سنبدأ بالحديث في هذا الفصل عن الأنواع المختلفة من هذه الحلقة ثم كيفية التكرار على أنواع البيانات التسلسلية مثل البيانات النصية Strings ثم نشرح الحلقات المتداخلة.

16.1 التصريح عن حلقة For

حددت لغة جو ثلاث طرق مختلفة لإنشاء الحلقات كُلٌّ منها بخصائص مختلفة بوضع كل حالات الاستخدام الممكنة للحلقات في الحسبان، وهي إنشاء حلقة for مع شرط Condition أو ForClause أو RangeClause، إذ سنشرح في هذا القسم كيفية تصريح واستخدام أنواع ForClause و Condition، لذا دعونا نلقي نظرةً على كيفية استخدام حلقة `for` مع ForClause أولًا.

تُعرّف حلقة ForClause من خلال كتابة تعليمة التهيئة initial statement متبوعة بشرط Condition متبوعة بتعليمة التقدّم Post Statement على هذا النحو:

```
for [ Initial Statement ] ; [ Condition ] ; [ Post Statement ] {
  [Action]
}
```

لكي نفهم المكونات السابق سنرى المثال التالي الذي يعرض حلقة `for` تتزايد ضمن نطاق محدد من القيم باستخدام التعريف السابق:

```
for i := 0; i < 5; i++ {
  fmt.Println(i)
}
```

الجزء الأول ضمن الحلقة هو تعليمة التهيئة `i := 0` والتي تعبر عن تصريح متغير اسمه `i` يُسمى متغير الحلقة وقيمته الأولية هي 0؛ أما الجزء الثاني فهو الشرط `i < 5` والذي يُمثّل الشرط الذي يجب أن يتحقق لكي ندخل في الحلقة أو في التكرار التالي من الحلقة، إذ يمكن القول ادخُل في الحلقة ونفّذ محتواها طالما أنّ قيمة `i` أصغر من 5، والجزء الأخير هو تعليمة التقدّم `i++` والتي تشير إلى التعديل الذي سنجريه على متغير الحلقة `i` بعد انتهاء كل تكرار، إذ يكون التعديل هنا هو زيادته بمقدار واحد.

سيكون خرج البرنامج كما يلي:

```
0
1
2
3
4
```

تتكرر الحلقة 5 مرات، بدايةً تكون قيمة `i` تساوي 0 وهي أصغر من 5، لذا ندخل في الحلقة وننفّذ محتواها `fmt.Println(i)` ثم نزيد قيمة متغير الحلقة بواحد فتصبح قيمته 1 وهي أصغر من 5، ثم ننفّذ محتوى الحلقة مرةً أخرى ونزيدها بواحد فتصبح 2 وهي أصغر من 5 وهكذا إلى أن تصبح قيمة متغير الحلقة 5 فينتهي تنفيذ الحلقة.

> تبدأ الفهرسة بالقيمة 0، لذا ستكون قيمة `i` في المرة الخامسة للتنفيذ هي 4 وليس 5.

طبعًا ليس بالضرورة أن تبدأ فهرس الحلقة بقيمة محددة مثل الصفر أو تنتهي بقيمة محددة وإنما يمكنك تحديد أي نطاق تريده كما في هذا المثال:

```
for i := 20; i < 25; i++ {
  fmt.Println(i)
}
```

سنبدأ هنا بالقيمة الأولية 20 وننتهي عند الوصول إلى القيمة 25 بحيث تكون أقل تمامًـا من 25، أي ننتهي بالعدد 24:

```
20
21
22
23
24
```

يمكنك أيضًا التحكم بخطوات الحلقة على عداد الحلقة كما تريد، إذ يمكنك مثلًا تقديم قيمـة المتغـير بثلاثـة بعد كل تكرار كما في المثال التالي:

```
for i := 0; i < 15; i += 3 {
  fmt.Println(i)
}
```

نبدأ هنا بالعدد 0 وننتهي عند الوصول إلى 15، وبعد كل تكرار نزيد العدّاد (متغـير الحلقـة) بثلاثـة، وسـيكون الخرج كما يلي:

```
0
3
6
9
12
```

يمكننا أيضًا التكرار بصورة عكسية، أي البدء بقيمة كبيرة والعودة خلفًـا، إذ سـنبدأ في المثـال التـالي بالعـدد 100 وننتهي عند الوصول إلى 0، وهنا لابد من أن يكون التقدم عكسيًا، أي باستخدام عملية الطرح وليس الجمع:

```
for i := 100; i > 0; i -= 10 {
  fmt.Println(i)
}
```

وضعنا هنا القيمة الأولية على 100 ووضعنا شرطًا للتوقف `i > 0` عند الصفر وجعلنا التقدم عكسيًا بمقدار 10 إلى الخلف بحيث سينقص 10 من قيمة متغير الحلقة بعد كل تكرار، وبالتالي سيكون الخرج كما يلي:

```
100
90
80
70
60
50
40
30
20
10
```

يمكنك أيضًا الاستغناء عن كتابة القيمة الأولية للمتغير وعن مقدار التقدّم وترك الشرط فقط، إذ نسمي هـذا النوع من الحلقات بحلقة الشرط Condition loop والتي تكافئ الحلقة while في لغات البرمجة الأخرى:

```
i := 0
for i < 5 {
  fmt.Println(i)

  i++
}
```

صرّحنا هنا عن المتغير `i` قبل الدخول إلى الحلقة، إذ يُنفّذ ما بداخل الحلقـة طالمـا أنّ الشـرط محقـق، أي `i` أصغر من 5، ولاحظ أيضًا أننا نزيد من قيمة المتغير `i` في نهاية الكتلة البرمجيـة الخاصـة بالحلقـة، فـذا لم نـزده، فسندخل في حلقة لانهائية، وفي الواقع نحن لم نستغنِ عن تعليمة التهيئة أو التقدّم (الخطوات)، وإنما عرّفنـاهم بطريقة مختلفة، والغاية من هذا التعريف هو المطابقة مع حلقة while المستخدَمة في باقي لغات البرمجة.

ذكرنا التعليمة `break` في بداية الفصل والتي سنستخدِمها عندما يكـون شـرط التوقـف غـير مرتبـط بعـدد محدد من التكرارات وإنما بحدث ما مثل تكرار عملية إدخال كلمة المرور من المستخدِم طالما ليسـت صـحيحةً، ويمكن التعبير عن هكذا حلقة من خلال كتابة `for` فقط كما يلي:

```
for {
  if someCondition {
        break
  }
  // do action here
}
```

مثال آخر على ذلك عندما نقرأ من بنية غير محددة الحجم مثل المخزِّن المؤقت buffer ولا نعرف متى سننتهي من القراءة:

```
package main
import (
  "bytes"
  "fmt"
  "io"
)
func main() {
  buf := bytes.NewBufferString("one\ntwo\nthree\nfour\n")
  for {
        line, err := buf.ReadString('\n')
        if err != nil {
              if err == io.EOF {
                    fmt.Print(line)
                    break
              }
              fmt.Println(err)
              break
        }
        fmt.Print(line)
  }
}
```

يُصرّح السطر `buf :=bytes.NewBufferString("one\ntwo\nthree\nfour\n")` عن مخزِّن مؤقت مع بعض البيانات، ونظرًا لأننا لا نعرف متى سينتهي المخزِّن المؤقت من القراءة، سننشئ حلقة `for` بدون أيّ بند، أي بدون شرط أو قيمة أولية أو مقدار تقدّم.

نستخدِم داخل الحلقة `line, err := buf.ReadString('\n')` لقراءة سطر من المخزِّن المؤقت والتحقق مما إذا كان هناك خطأ في القراءة منه، فإذا كان هناك خطأ، فسنعالج الخطأ ونستخدِم الكلمة المفتاحية `break` للخروج من حلقة `for`، فكما نلاحظ أننا من خلال تعليمة `break` لم نعُد بحاجة إلى كتابة بند الشرط لإيقافها.

تعلمنا في هذا المثال كيفية التصريح عن حلقة ForClause وكيفية استخدامها للتكرار على مجال محدد من القيم، كما تعلمنا أيضًا كيفية استخدام حلقة الشرط للتكرار حتى يتحقق شرط معيّن.

16.2 التكرار على أنواع البيانات المتسلسلة باستخدام RangeClause

من الشائع في جو استخدام الحلقات للتكرار على عناصر أنواع البيانات المتسلسلة أو التجميعية مثل الشرائح والمصفوفات والسلاسل النصية، ولتسهيل هذه العملية يمكننا استخدام حلقة for مع بنية RangeClause، ويمكنك عمومًا استخدام بنية ForClause للتكرار على أيّ نوع من البيانات بطرق محددة طبعًا، إلا أن بنية RangeClause تُعَدّ مُنظّمَةً أكثر وأسهل في القراءة.

بدايةً سنلقي نظرةً على كيفية التكرار باستخدام ForClause في هكذا حالات:

```
package main
import "fmt"
func main() {
  sharks := []string{"hammerhead", "great white", "dogfish",
"frilled", "bullhead", "requiem"}
  for i := 0; i < len(sharks); i++ {
        fmt.Println(sharks[i])
  }
}
```

بتشغيل الشيفرة السابقة سنحصل على الخرج التالي:

```
hammerhead
great white
dogfish
frilled
bullhead
requiem
```

سنستخدِم الآن بنية RangeClause لتنفيذ الإجراءات نفسها:

```
package main
import "fmt"
func main() {
  sharks := []string{"hammerhead", "great white", "dogfish",
"frilled", "bullhead", "requiem"}
  for i, shark := range sharks {
        fmt.Println(i, shark)
  }
```

```
}
```

سيُطبع في هذه الحالة كل عنصر في القائمة. هنا استخدمنا المتغيرين `i` و `shark` من أجـل عمليـة التكـرار ويمكنك استخدام أيّ اسم آخر لهذه المتغيرات، إذ تعيـد الكلمـة المفتاحيـة `range` قيمـتين؛ الأولى هي فهـرس العنصر والثانية هي قيمته، وبالتالي سيكون الخرج كما يلي:

```
0 hammerhead
1 great white
2 dogfish
3 frilled
4 bullhead
5 requiem
```

قد لا نحتاج لعرض فهرس العنصر في بعض الأحيان، لكن إذا حذفناها، فسنتلقى خطأً في وقت التصريف:

```
package main
import "fmt"
func main() {
  sharks := []string{"hammerhead", "great white", "dogfish",
"frilled", "bullhead", "requiem"}
  for i, shark := range sharks {
        fmt.Println(shark)
  }
}
```

سيكون الخرج كما يلي:

```
src/range-error.go:8:6: i declared and not used
```

نظرًا لأنّ `i` قد صُرّح عنه في الحلقة `for`، ولكن لم يُستخدم مطلقًا، فسوف يُعطي المُصرّف خطـأً يُشـير إلى أن هناك متغيرًا قد صُرّح عنه ولم يُستخدَم، وهذا الخطأ ستتلقاه في جو في أيّ وقت تُصـرّح فيـه عن متغـير ولا تستخدِمه، ولحل هذه المشكلة سنستخدِم المتغير المجهول الذي يُعبَّر عنه بالشرطة السفلية `_` والـذي يشـير إلى أنّ هناك قيمة ستُعاد ونعرف ذلك لكن لا نريدها، لذا سنستبدل المتغير `i` بالمتغير المجهول وسينجح الأمر:

```
package main
import "fmt"
func main() {
  sharks := []string{"hammerhead", "great white", "dogfish",
"frilled", "bullhead", "requiem"}
```

```
    for _, shark := range sharks {
        fmt.Println(shark)
    }
}
```

سيكون الخرج كما يلي:

```
hammerhead
great white
dogfish
frilled
bullhead
requiem
```

يوضِّح الخرج أنّ حلقة `for` قد تكررت على شريحة السلاسل وطُبع كل عنصر من الشريحة بدون الفهرس.

يمكنك أيضًا استخدام الكلمة `range` لإضافة عناصر إلى قائمة ما:

```
package main
import "fmt"
func main() {
    sharks := []string{"hammerhead", "great white", "dogfish",
"frilled", "bullhead", "requiem"}
    for range sharks {
        sharks = append(sharks, "shark")
    }
    fmt.Printf("%q\n", sharks)
}
```

سيكون الخرج كما يلي:

```
['hammerhead', 'great white', 'dogfish', 'frilled', 'bullhead',
'requiem', 'shark', 'shark', 'shark', 'shark', 'shark', 'shark']
```

أضفنا هنا السلسة 'shark' إلى الشريحة `sharks` عدة مرات من خلال التكرار عليها، أي أضفنا إليها x مـرة، بحيث تكون x هي طول القائمة التي نُكرر عليها، ولاحظ هنـا أننـا لسـنا بحاجـة إلى اسـتخدام المتغـير المجهـول إطلاقًا، وذلك لأن هكذا حالة مُعرّفة ضمن جو، كما أننا لم نستخدِم أيّ متغير.

يمكننا أيضًا استخدام العامِل `range` لملء قيم الشريحة كما يلي:

```
package main
import "fmt"
func main() {
  integers := make([]int, 10)
  fmt.Println(integers)
  for i := range integers {
        integers[i] = i
  }
  fmt.Println(integers)
}
```

نُصرِّح هنا عن الشريحة integers على أنها شريحة من الأعداد الصحيحة int ونحجز مساحةً لعشرة عناصر مُسبقًا ثم نطبع الشريحة ثم نملأ هذه الشريحة بقيم عناصر فهارسها، ثم نطبعها من جديد:

```
[0 0 0 0 0 0 0 0 0 0]
[0 1 2 3 4 5 6 7 8 9]
```

نلاحظ أنه في المرة الأولى للطباعة تكون عناصر الشريحة صفرية، لأننا حجزنا 10 عناصر ولم نُعطهم قيمًا، أي القيمة الافتراضية هي 0، لكن في المرة التالية تكون قيم الشريحة من 0 إلى 9 وهذا يُكافئ قيم الفهارس لها.

يمكننا أيضًا استخدام العامِل range للتكرار على محارف سلسلة:

```
package main
import "fmt"
func main() {
  sammy := "Sammy"
  for _, letter := range sammy {
        fmt.Printf("%c\n", letter)
  }
}
```

سيكون الخرج كما يلي:

```
S
a
m
m
y
```

عند استخدام range للتكرار على عناصر خريطة map، فستُعاد قيمة كل من المفتاح والقيمة:

```
package main
import "fmt"
func main() {
  sammyShark := map[string]string{"name": "Sammy", "animal": "shark",
"color": "blue", "location": "ocean"}
  for key, value := range sammyShark {
        fmt.Println(key + ": " + value)
  }
}
```

سيكون الخرج كما يلي:

```
color: blue
location: ocean
name: Sammy
animal: shark
```

عناصر الخريطة غير مُرتبة، وبالتالي ستُطبع عناصرها بترتيب عشوائي في كل مرة تُنفّذ فيها الشيفرة.

الآن بعد أن تعرّفنا على كيفية التكرار على العناصر المتسلسة، سنتعلم كيفية استخدام الحلقات المتداخلة.

16.3 الحلقات المتداخلة Nested Loops

يمكن لحلقات التكرار في جو أن تداخل كما هو الحال في بقية لغات البرمجة، فحلقة التكـرار المتداخلـة هي الحلقة الموجودة ضمن حلقـة تكـرار أخـرى وهـذا مفيـد لتكـرار عـدة عمليـات على كـل عنصـر مثلًا وهي شـبيهة بتعليمات if المتداخلة، وتُبنى حلقات التكرار المتداخلة كما يلي:

```
for {
  [Action]
  for {
    [Action]
  }
}
```

يبدأ البرنامج بتنفيذ حلقة التكرار الخارجية ويُنفَّذ أول تكرار فيها والذي سيؤدي إلى الدخول إلى حلقـة التكـرار الداخلية، مما يؤدي إلى تنفيذها إلى أن تنتهي تمامًا، بعد ذلـك سـيعود تنفيـذ البرنـامج إلى بدايـة حلقـة التكـرار الخارجية، ويبدأ بتنفيذ التكرار الثاني ثم سيصل التنفيذ إلى حلقة التكرار الداخلية، وستُنفَّذ حلقة التكـرار الداخليـة

بالكامل، ثم سيعود التنفيذ إلى بداية حلقة التكرار الخارجية، وهكذا إلى أن ينتهي تنفيذ حلقة التكـرار الخارجيـة أو إيقاف حلقة التكرار عبر استخدام `break` أو غيرها من التعليمات.

لنُنشِئ مثالًا يستعمل حلقة `for` متداخلة لكي نفهم كيف تعمل بدقة، حيث سـتمر حلقـة التكـرار الخارجيـة في المثال الآتي على شريحة من الأعداد اسمها `numList`؛ أما حلقـة التكـرار الداخليـة فسـتمر على شـريحة من السلاسل النصية اسمها `alphaList`:

```
package main
import "fmt"
func main() {
  numList := []int{1, 2, 3}
  alphaList := []string{"a", "b", "c"}
  for _, i := range numList {
        fmt.Println(i)
        for _, letter := range alphaList {

              fmt.Println(letter)
        }
  }
}
```

سيظهر الناتج التالي عند تشغيل البرنامج:

```
1
a
b
c
2
a
b
c
3
a
b
c
```

يُظهِر الناتج السابق أنَّ البرنامج قد أكمل أول تكرار على عناصر حلقة التكرار الخارجية بطباعة العـدد، ومن ثم بدأ بتنفيذ حلقة التكـرار الداخليـة، لـذا طبـع المحـارف a و b و c على التـوالي، وبعـد انتهـاء تنفيـذ حلقـة التكـرار

الداخلية، عاد البرنامج إلى بداية حلقة التكرار الخارجية طابعًا العدد 2، ثم بـدأ تنفيـذ حلقـة التكـرار الداخليـة، ممـا يؤدي إلى إظهار a و b و c مجددًا وهكذا.

يمكن الاستفادة من حلقات for المتداخلة عند المرور على عناصر شريحة تتألف من شرائح، فإذا اسـتعملنا حلقة تكرار وحيدة لعرض عناصر شريحة تتألف من عناصر تحتوي على شرائح، فسـتُعرَض قيم الشـرائح الداخليـة على أساس عنصر:

لاحظ المثال التالي:

```
package main
import "fmt"
func main() {
  ints := [][]int{
        []int{0, 1, 2},
        []int{-1, -2, -3},
        []int{9, 8, 7},
  }
  for _, i := range ints {
        fmt.Println(i)
  }
}
```

سيكون الخرج هنا كما يلي:

```
[0 1 2]
[-1 -2 -3]
[9 8 7]
```

إذا أردنا الوصول إلى العناصر الموجودة في الشرائح الداخلية، فيمكننا استعمال حلقة for متداخلة:

```
package main
import "fmt"
func main() {
  ints := [][]int{
        []int{0, 1, 2},
        []int{-1, -2, -3},
        []int{9, 8, 7},
  }
  for _, i := range ints {
```

```
        for _, j := range i {
            fmt.Println(j)
        }
    }
}
```

سيكون الخرج كما يلي:

```
0
1
2
-1
-2
-3
9
8
7
```

نستطيع الاستفادة من حلقات `for` المتداخلة عندما نريد التكرار على عناصر قابلة للتكرار داخل الشرائح.

16.4 استخدام تعليمات break و continue

تسمح لك حلقة `for` بأتمتة وتكرار المهام بطريقة فعّالة، لكن في بعض الأحيان قد يتدخل عامل خارجي في طريقة تشغيل برنامجك، وعندما يحدث ذلك، فربما تريد من برنامجك الخروج تمامًا من حلقة التكرار أو تجاوز جزء من الحلقة قبل إكمال تنفيذها أو تجاهل هذا العامِل الخارجي تمامًا، لذا يمكنك فعل ما سبق باستخدام تعابير `break` و `continue`.

16.4.1 تعليمة break

توفِّر لك تعليمة `break` -والتي تعاملنا معها سابقًا- القدرة على الخروج من حلقة التكرار عند حدوث عامل خارجي، وتوضَع عادةً ضمن تعبير `if`، وفيما يلي أحد الأمثلة الذي يستعمل تعليمة `break` داخل حلقة `for`:

```
package main
import "fmt"
func main() {
  for i := 0; i < 10; i++ {
        if i == 5 {
            fmt.Println("Breaking out of loop")
```

```
                break // break here
            }
            fmt.Println("The value of i is", i)
    }
    fmt.Println("Exiting program")
}
```

بنينا حلقة تكرار `for` التي تعمل طالما كانت قيمة المتغير `i` أصغر من 10 وحددنا التقدّم بمقدار 1 بعد كـل تكرار، ثم لدينا تعليمة `if` التي تختبر فيما إذا كان المتغير `i` مساوٍ للعدد 5، فعند حدوث ذلك، فسـتُنفَّذ `break` للخروج من الحلقة.

توجد داخل حلقة التكرار الدالة `fmt.Println()` التي تُنفَّذ في كـل تكـرار وتطبـع قيمـة المتغـير `i` إلى أن نخرج من الحلقة عبر `break` وهنا يكون لدينا تعليمة طباعة أخرى تطبع Breaking out of loop، ولكي نتأكد أننا خرجنا من الحلقة تركنا تعليمة طباعة أخرى تطبع Exiting program، وبالتالي سيكون الخرج كما يلي:

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
Breaking out of loop
Exiting program
```

يُظهر الناتج السابق أنَّه بمجرد أن أصبحت قيمة المتغـير `i` مسـاويةً للعـدد 5، فسـوف ينتهي تنفيـذ حلقـة التكرار عبر `break`.

16.4.2 تعليمة continue

تسمح لنا تعليمة `continue` تسمح لنا تنفيذ البرنامج إلى أوّل حلقة التكرار عند تنفيذ `continue`، وتوضَع عادةً ضمن تعليمة `if`.

سنستخدِم البرنـامج نفسـه الـذي اسـتعملناه لشـرح التعبـير `break` أعلاه، لكننـا سنسـتخدم حاليًـا التعبـير `continue` بدلًا من `break`:

```
package main
import "fmt"
func main() {
  for i := 0; i < 10; i++ {
```

```
        if i == 5 {
            fmt.Println("Continuing loop")
            continue // break here
        }
        fmt.Println("The value of i is", i)
    }
    fmt.Println("Exiting program")
}
```

الفرق الوحيد عن الشيفرة السابقة هو أننا استبدلنا تعليمة break بتعليمة continue، وبالتالي لن يتوقف البرنامج عند وصول متغير الحلقة إلى 5، وإنما فقط سيتخطاها ويتابع التنفيذ، وبالتالي سيكون الخرج كما يلي:

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
Continuing loop
The value of i is 6
The value of i is 7
The value of i is 8
The value of i is 9
Exiting program
```

نلاحظ أنَّ السطر الذي يجب أن يحتوي على `The value of i is 5` ليس موجودًا في المخرجات، لكن سيُكمَل تنفيذ حلقة التكرار بعد هذه المرحلة مما يؤدي إلى طباعة الأعداد من 6 إلى 10 قبل إنهاء تنفيذ الحلقة.

يمكنك استخدام التعليمة continue لتفادي استخدام تعابير شرطية معقدة ومتداخلة أو لتحسين أداء البرنامج عن طريق تجاهل الحالات التي ستُرفَض نتائجها، وبالتالي ستؤدي تعليمة continue إلى جعل البرنامج يتجاهل تنفيذ حلقة التكرار عند تحقيق شرط معيّن، لكن بعد ذلك سيُكمِل تنفيذ الحلقة مثل العادة.

16.5 تعليمة break مع الحلقات المتداخلة

من المهم أن تتذكر أنّ تعليمة break ستوقف فقط تنفيذ الحلقة الداخلية التي يتم استدعاؤها فيها، فإذا كان لديك مجموعة متداخلة من الحلقات، فستحتاج إلى تعليمة break لكل حلقة:

```
package main
import "fmt"
func main() {
  for outer := 0; outer < 5; outer++ {
        if outer == 3 {
              fmt.Println("Breaking out of outer loop")
              break // break تعليمة
        }
        fmt.Println("The value of outer is", outer)
        for inner := 0; inner < 5; inner++ {
              if inner == 2 {
                    fmt.Println("Breaking out of inner loop")
                    break // break تعليمة
              }
              fmt.Println("The value of inner is", inner)
        }
  }
  fmt.Println("Exiting program")
}
```

تتكرر الحلقتان 5 مرات ولكل منهما تعليمة `if` مع تعليمة `break`، فالحلقة الخارجية ستُنهى إذا كانت قيمة `outer` تساوي 3؛ و الحلقة الداخلية ستُنهى إذا كانت قيمة `inner` تساوي 2، وبالتالي سيكون الخرج:

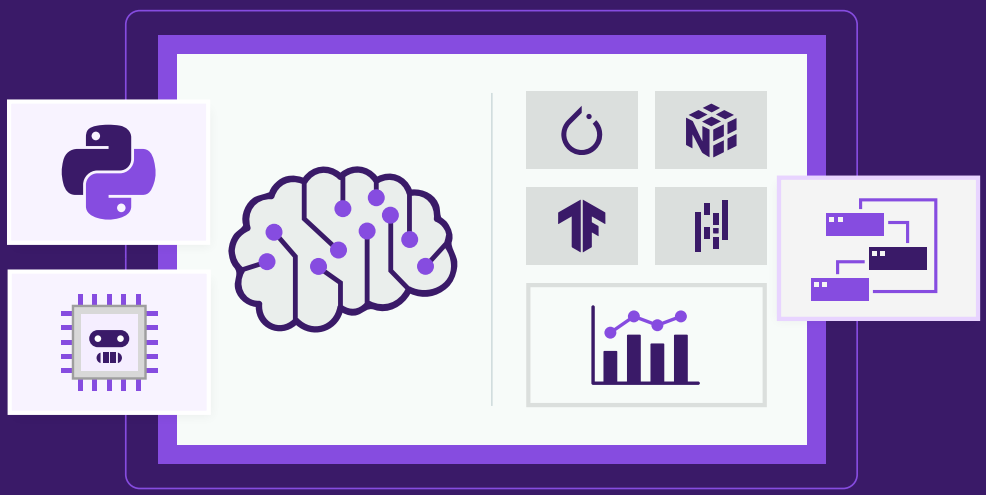
```
The value of outer is 0
The value of inner is 0
The value of inner is 1
Breaking out of inner loop
The value of outer is 1
The value of inner is 0
The value of inner is 1
Breaking out of inner loop
The value of outer is 2
The value of inner is 0
The value of inner is 1
Breaking out of inner loop
Breaking out of outer loop
Exiting program
```

لاحظ أنه في كل مرة تنكسر أو تُنهى فيها الحلقة الداخلية باستخدام break، فإن الحلقة الخارجيـة لا تنكسـر فتعليمة break ستنهي فقط تنفيذ الحلقة الداخلية التي استُدعيَت منها.

16.6 الخاتمة

رأينا في هذا الفصل كيف تعمل حلقة التكرار for في لغة جو وكيف نسـتطيع إنشـاءها واسـتعمالها، حيث تستمر حلقة for بتنفيذ مجموعة من الشـيفرات لعـدد محـدد من المـرات، كمـا تعرّفنـا أيضًـا على ثلاثـة أنـواع مختلفة من هذه الحلقة التكرارية وكيفية استخدام كل منها، وتعلمنا أيضًا كيفيـة اسـتخدام تعليمتَي continue و break للتحكم بتنفيذ حلقة for بفاعلية أكبر.

دورة الذكاء الاصطناعي



تعلم الذكاء الاصطناعي وتعلم الآلة والتعلم العميق
وتحليل البيانات، وأضفها إلى تطبيقاتك

التحق بالدورة الآن

17. تعريف واستدعاء الدوال Functions

الدالة function في البرمجة هي عبارة عن كتلةً من التعليمات التي تنفِّذ إجراءً ما، ويمكن بعد تعريفها إعادة استخدامها في أكثر من موضع. تجعل الدوال الشيفرة تركيبية modular، مما يسمح بتقسيم الشيفرة إلى أجـزاء صغيرة سهلة الفهم واستخدامها مرارًا وتكرارًا.

تضم لغة جو مكتبةً قياسيةً تدعى `fmt` تحتوي العديد من الدوال المُضمّنة التي قد تكون قـد أَلفت التعامـل معها واستخدمتها كثيرًا في الفصول السابقة مثل:

- الدالة `fmt.Println()` التي تُستخدَم للطباعة على شاشة الخرج القياسية المُستخدَمة.
- الدالة `fmt.Printf()` التي تُستخدَم للطباعة مع إمكانية تنسيق الخرج.

تتضمن أسماء الدوال الأقواس `()` وقد تتضمن معامِلات تمرر عبر هذه الأقواس أيضًـا، وسـنتعلم في هـذا الفصل كيفية تعريف الدوال وكيفية استخدامها في البرامج بالتفصيل.

17.1 تعريف الدالة

لنبدأ بتحويل برنامج "Hello, World!" إلى دالة، لذا أنشئ ملفًا نصـيًا جديـدًا وافتحـه في محـرر النصـوص المفضل عندك ثم استدع البرنامج hello.go.

تُعرَّف الدالة في لغة جو باستخدام الكلمة المفتاحية `func` متبوعة باسم من اختيارك ويفضل أن تختار اسمًا مناسبًا يعبِّر عن الوظيفة التي تقوم بها الدالة، ثم قوسين يمكن أن يَحتويـا المعـامِلات الـتي سـتأخذها الدالـة ثم ينتهي التعريف بنقطتين، وسنعرّف هنا دالةً باسم `hello()`:

```
func hello() {}
```

أعددنا في الشيفرة أعلاه تعليمة التهيئة لإنشاء الدالة، وبعد ذلك سنضيف سطرًا ثانيًا مُزاحًا بـأربع مسـافات بيضاء ثم سنكتب التعليمات التي ستنفّذها الدالة التي ستطبع العبارة Hello, World! في الطرفية كما يلي:

```
func hello() {
  fmt.Println("Hello, World!")
}
```

أتممنـا تعريـف دالتنـا ولكن إذا نَفَّـذنا البرنـامج الآن، فلن يحـدث أيّ شـيء لأننـا لم نسـتدع الدالـة، لـذلك سنستدعي الدالة `hello()` ضمن الدالة `main()` على هذا النحو:

```
package main
import "fmt"
func main() {
  hello()
}
func hello() {
  fmt.Println("Hello, World!")
}
```

لننفّذ البرنامج الآن:

```
$ go run hello.go
```

يجب أن تحصل على الخرج التالي:

```
Hello, World!
```

الدالة `main()` هي دالة خاصة تخبر المُصرّف أنّ هـذا هـو المكـان الـذي يجب أن يبـدأ منـه البرنـامج، فـأيّ برنامج تريده أن يكون قابلاً للتنفيذ (برنامج يمكن تشغيله من الطرفية)، فستحتاج إلى دالـة `main()`، كمـا يجب أن تظهر الدالة `main()` مرةً واحـدةً فقـط وأن تكـون في الحزمـة `main` ولا تسـتقبل أو تعيـد أيّ وسـائط، وهـذا يسمح بتنفيذ البرنامج في أيّ برنامج جو آخر حسب المثال التالي:

```
package main
import "fmt"
func main() {
  fmt.Println("this is the main section of the program")
}
```

بعض الـدوال أكـثر تعقيـدًا بكثـير من الدالـة `hello()` الـتي عرّفناهـا أعلاه، إذ يمكننـا على سـبيل المثـال استخدام حلقة `for` والتعليمات الشرطية وغيرها داخـل كتلـة الدالـة، فالدالـة المُعرّفـة أدنـاه على سـبيل المثـال تستخدِم تعليمةً شرطيةً للتحقق مما إذا كانت المدخلات الممرّرة إلى المتغـير `name` تحتـوي على حـرف صـوتي vowel، ثم تستخدِم الحلقة `for` للتكرار على الحروف الموجودة في السلسلة النصية `name`.

```
package main
import (
  "fmt"
  "strings"
)
func main() {
  names()
}
func names() {
  fmt.Println("Enter your name:")
  var name string
  fmt.Scanln(&name)
  // Check whether name has a vowel
  for _, v := range strings.ToLower(name) {
        if v == 'a' || v == 'e' || v == 'i' || v == 'o' || v == 'u' {
              fmt.Println("Your name contains a vowel.")
              return
        }
  }
  fmt.Println("Your name does not contain a vowel.")
}
```

تستخدِم الدالـة `names()` الـتي عرّفناهـا أعلاه تعليمـةً شـرطيةً وحلقـة `for`، وهـذا توضـيح لكيفيـة تنظيم الشيفرة البرمجية ضمن تعريف الدالة، كما يمكننا أيضًا جعل التعليمة الشرطية والحلقة `for` دالتين منفصلتين.

يجعل تعريف الدوال داخل البرامج الشـفرة البرمجيـة تركيبيـةً modular وقابلـةً لإعـادة الاسـتخدام، وذلـك سيتيح لنا استدعاء الدالة نفسها دون إعادة كتابة شيفرتها في كل مرة.

17.2 المعاملات

عرّفنا حتى الآن دالة ذات قوسين فارغين ولا تأخذ أيّ وسطاء arguments، وسنتعلم في هذا القسم كيفيـة تعريف المعامِلات parameters التي تمكننا من تمرير البيانات إلى الدوال.

يُعَدّ المعامِل paramete كيانًا مُسمًّى يوضَع في تعريف الدالة ويعـرِّف وسـيطًا يمكن أن تقبلـه الدالـة عنـد استدعائها، ويجب عليك في لغة Go أن تحدد نوع البيانات data type لكل معامِل.

لننشئ برنامجًا يُكرر كلمة عدة مرات، إذ سنحتاج إلى متغير من النوع `string` سنسميه `word` ومتغــير من النوع `int` سنسميه `reps` يُحدد عدد التكرارات.

```
package main
import "fmt"
func main() {
  repeat("Sammy", 5)
}
func repeat(word string, reps int) {
  for i := 0; i < reps; i++ {
        fmt.Print(word)
  }
}
```

مرّرنا السلسلة `Sammy` إلى المتغير `word` والقيمة 5 إلى المعامِل `reps` وذلك وفقًا لترتيب المعــامِلات في ترويسة الدالة `repeat`، إذ تتضمّن الدالة حلقة `for` تطبع قيمـة المعامِـل `word` عـدة مـرات يُحـدّدها المعامِـل `reps`، وسيكون الخرج كما يلي:

```
ammySammySammySammySammy
```

إذا كانت لديك مجموعة من المعامِلات وجميعها تمتلك القيمة نفسها، فيمكنـك تجاهـل تحديـد النـوع من أجل كل متغير كما سنرى، لـذا دعنـا ننشـئ برنامجًـا صـغيرًا يأخـذ ثلاثـة معـامِلات `x` و `y` و `z` من النـوع `int`، إذ سننشئ دالةً تجمع تلك المعامِلات وفق عدة مجموعات ثم تطبع الدالة حاصل جمعها.

```
package main
import "fmt"
func main() {
  addNumbers(1, 2, 3)
}
func addNumbers(x, y, z int) {
  a := x + y
  b := x + z
  c := y + z
  fmt.Println(a, b, c)
}
```

عند تعريف الدالة addNumbers لم نكن بحاجة إلى تحديد نوع كل متغير على حدة، وإنما وضعنا نوع بيانات كل المتغيرات مرةً واحدةً فقط.

مرّرنا العدد 1 إلى المعامل x والعدد 2 إلى المعامل y والعدد 3 إلى المعامل z، إذ تتوافق هذه القيم مع المعامِلات المقابلة لها في ترتيب الظهور، ويُجري البرنامج العمليات الحسابية على المعامِلات على النحو التالي:

```
a = 1 + 2
b = 1 + 3
c = 2 + 3
```

تطبع الدالة أيضًا a و b و c، وبناءً على العمليات الحسابية أعلاه، فستساوي قيمة a العدد 3 و b العدد 4 و c العدد 5، ولننفّذ البرنامج سنكتب ما يلي:

```
$ go run add_numbers.go
```

سيكون الخرج كما يلي:

```
3 4 5
```

عندما نمرر 1 و 2 و 3 على أساس معامِلات إلى الدالة addNumbers()، فإننا نتلقى الناتج المتوقع.

تُعَدّ المعامِلات وسائط تُعرَّف عادة على أساس متغيرات ضمن تعريف الدالة، كما يمكن تعيين أو إسناد قيم إليها عند تنفيذ التابع بتمرير وسائط إلى الدالة.

17.3 إعادة قيمة

يمكن تمرير قيم إلى الدالة ويمكن كذلك أن تُنتج الدالة قيمةً وتُعيدها لمن استدعاها، إذ يمكن أن تنتج الدالة قيمةً عبر استخدام التعليمة return وهي تعليمة اختيارية، ولكن في حال استخدامها ستُنهي الدالة عملها مباشرةً وتوقف تنفيذها وتُمرَّر قيمة التعبير الذي يعقُبها اختياريًا إلى المستدعي، كما يجب أن تحدد نوع البيانات المُعادة أيضًا.

استخدمنا حتى الآن الدالة fmt.Println() بدلاً من التعليمة return في دوالنا لطباعة شيء بدلًا من إعادته، ولننشئ الآن برنامجًا يعيد متغيرًا بدلًا من طباعته مباشرةً، لذا سننشئ برنامجًا في ملف نصي جديد يسمى double.go يحسب ناتج مُضاعفة المعامِل x ويُسند الناتج إلى المتغير y ثم يعيده، كما سنطبع المتغير result والذي يساوي ناتج تنفيذ الدالة double(3).

```
package main
import "fmt"
func main() {
```

```
  result := double(3)
  fmt.Println(result)
}
func double(x int) int {
  y := x * 2
  return y
}
```

لننفّذ البرنامج كما يلي:

```
$ go run double.go
```

وسيكون الخرج:

```
6
```

خرج هذا البرنامج هو العدد الصحيح 6 الذي أعادته الدالة وهو ما نتوقعه إذا طلبنا من جو حساب ناتج ضرب العدد 2 بالعدد 3.

إذا حددنا نـوع القيمـة المُعـادة فيجب علينـا إعـادة قيمـة من هـذا النـوع وإلا فسـيُعطي البرنـامج خطـأً في التصريف، ففي المثال التالي سنلغي تعليمة الإعادة المُستخدَمة في الشيفرة السابقة بوضع تعليق على تعليمـة الإعادة لكي يتجاهلها المُصرّف كما يلي:

```
package main
import "fmt"
func main() {
  result := double(3)
  fmt.Println(result)
}
func double(x int) int {
  y := x * 2
  // return y
}
```

لنحاول تنفيذ البرنامج:

```
$ go run double.go
```

سيُشير الخرج إلى خطأ لأنه لم يجد تعليمة الإعادة `return`:

```
./double.go:13:1: missing return at end of function
```

لا يمكن تصريف البرنامج بدون تعليمة الإعادة هذه، فعندما تصل الدالة إلى تعليمة `return` فإنها ستُنهي تنفيذ الدالة حتى إذا كان هناك تعليمات تالية ضمنها:

```
package main
import "fmt"
func main() {
  loopFive()
}
func loopFive() {
  for i := 0; i < 25; i++ {
        fmt.Print(i)
        if i == 5 {
              //i == 5 أوقف الدالة عندما
   return
        }
  }
  fmt.Println("This line will not execute.") // هذا السطر لن ينفَّذ
}
```

نستخدم هنا حلقة `for` تُؤدي عملية تكرار 25 مرة وبداخلها تعليمة `if` تتحقق مما إذا كانت قيمة `i` تساوي العدد 5، فإذا كانت كذلك، فسيكون لدينا تعليمة `return` تُنهي تنفيذ الحلقة وتُنهي تنفيذ الدالة أيضًا، وهذا يعني أنّ باقي التكرارات لن تُنفّذ وبالتالي فإن السطر الأخير من الدالة `This line will not execute` لن يُنفّذ.

وبما أن استخدام التعليمة `return` داخل الحلقة `for` يؤدي إلى إنهاء الدالة، وبالتالي لن يُنفَّذ السطر الموجود خارج الحلقة، فإذا استخدمنا بدلًا من ذلك التعليمة `break`، فسيُنفّذ السطر `fmt.Println()` الأخير من المثال السابق.

نعيد التذكير أنَّ التعليمة `return` تنهي عمل الدالة وقد تعيد قيمةً إذا أعقبها تعبير وكان ذلك محدد في تعريف الدالة.

17.4 إعادة عدة قيم

يمكن للدوال أن تُعيد أكثر من قيمة، إذ سنجعل برنامج repeat.go يُعيد قيمتين بحيث تكون القيمة الأولى القيمة المُكررة والثانية خطأً في حال كانت قيمة المعامِل `reps` أصغر أو تساوي 0.

```
package main
import "fmt"
func main() {
  val, err := repeat("Sammy", -1)
  if err != nil {
        fmt.Println(err)
        return
  }
  fmt.Println(val)
}
func repeat(word string, reps int) (string, error) {
  if reps <= 0 {
        return "", fmt.Errorf("invalid value of %d provided for reps.
value must be greater than 0.", reps)
  }
  var value string
  for i := 0; i < reps; i++ {
        value = value + word
  }
  return value, nil
}
```

تتحقق الدالة `repeat` بدايةً من أن الوسيط `reps` صالح، فأيّ قيمة أقل أو تساوي الصفر ستؤدي إلى خطأ. بما أننا مررنا القيمة -1 إلى `reps` فهذا سيؤدي إلى تحقق شرط حدوث خطأ، وبالتالي ستُعاد قيمتين الأولى هي سلسلة فارغة "" والثانية هي رسالة خطأ، وبالطبع يجب علينا إعادة قيمتين دومًا تبعًا للتعريف الذي وضعناه في ترويسة الدالة، إذ حددنا أنّ هناك قيمتان مُعادتان من الدالة؛ فالأولى هي سلسلة والثانية هي خطأ، ولهذا السبب أعدنا سلسلةً فارغةً في حال ظهور خطأ.

نستدعي الدالة `repeat` بداخل الدالة `main` ونُسند القيم المُعادة إلى متغيرين هما `value` و `err`، وبما أنّ هناك احتمال لحدوث خطأ، فسنتحقق في الأسطر التالية من وجوده، وفي حال وجوده سنطبع رسالةً تشير إلى الخطأ وسنستخدِم التعليمة `return` للخروج من الدالة `main()` والبرنامج، وبتنفيذ البرنامج نحصل على ما يلي:

```
invalid value of -1 provided for reps. value must be greater than 0.
```

> من السلوكيات الجيدة في البرمجة إعادة قيمتين أو ثلاثة، بالإضافة إلى إعادة كل الأخطاء بحيث تكون آخر قيمة معادة من الدالة.

17.5 الدوال المرنة Variadic

الدالة المرنة Variadic هي دالة لا تقبل أي قيمة أو تقبل قيمةً واحدةً أو قيمتين أو أكثر على أساس وسيط واحد، وبالرغم من أن الدوال المرنة ليست شائعة الاستخدام لكنها تجعل الشيفرة أنظف وأكثر قابليةً للقراءة، وأحد الأمثلة على هذا النوع من الدوال هو الدالة `Println` من الحزمة `fmt`:

```
func Println(a ...interface{}) (n int, err error)
```

نسمي الدالة التي تتضمن مُعامِلًا مُلحقًا بثلاثة نقاط `...` كما في الشيفرة أعلاه بالدالة المرنة، إذ تشير النقاط الثلاث إلى أنّ هذا المعامِل يمكن أن يكون صفر قيمة أو قيمةً واحدةً أو قيمتين أو عدة قيم، وبالتالي تُعَدّ الدالة `fmt.Println` دالةً مرنةً لأنها تتضمن مُعامِلًا مرنًا يسمى `a`.

سنستخدِم في المثال التالي الدالة المرنة السابقة، وسنبيّن كيف أنه من الممكن أن نمرر لها عددًا غير مُحدد من الوسائط:

```
package main
import "fmt"
func main() {
  fmt.Println()
  fmt.Println("one")
  fmt.Println("one", "two")
  fmt.Println("one", "two", "three")
}
```

كما تُلاحظ فإننا نستدعيها أول مرة بدون تمرير أيّ وسيط ثم نستدعيها مع تمرير وسيط واحد هو السلسلة `one` ثم نستدعيها مع وسيطين ثم نستدعيها مع ثلاثة وسائط، ولننفذ البرنامج الآن:

```
$ go run print.go
```

سيكون الخرج كما يلي:

```
one
one two
one two three
```

لاحظ أنّ السطر الأول من الخرج فارغ لأننا استدعينا تعليمة الطباعة بدون تمرير أيّ متغير ثم السطر الثاني يحتوي على `one` لأن دالة الطباعة التي استدعيناها في المرة الثانية تتضمنها، ثم `one two` لأننا مررناهما إلى دالة الطباعة في المرة الثالثة، وأخيرًا السطر الأخير `one two three` لأننا مررنا هذه الكلمات إلى دالة الطباعة في الشيفرة السابقة.

بعد أن اطلعنا على كيفية استخدام الدوال المرنة، سنتحدّث الآن عن كيفية تعريف هـذه الـدوال ضـمن لغـة البرمجة جو Go.

17.6 تعريف الدوال المرنة

كما ذكرنا سابقًا فإن الدوال المرنة تُعرّف من خلال وضع ثلاث نقاط `...` بعد اسم أحد المعامِلات فيها، وفي المثال التالي سنعرّف دالةً نمرر لها أسماءً لكي تُحَييهم:

```
package main
import "fmt"
func main() {
  sayHello()
  sayHello("Sammy")
  sayHello("Sammy", "Jessica", "Drew", "Jamie")
}
func sayHello(names ...string) {
  for _, n := range names {
        fmt.Printf("Hello %s\n", n)
  }
}
```

تتضمّن الدالة `sayHello` معامِل اسمه `names` من النوع `string`، وكما نلاحظ فإنه توجد ثلاث نقاط بعد اسمه، وبالتالي هو معامل مرن أي يقبل عـددًا غـير مُحـدد من الوسـائط، وبالتـالي تُعَـدّ الدالـة `sayHello()` دالةً مرنةً.

تُعامِل هذه الدالة المعامِل `names` على أنـه شـريحة من الأسـماء، أي أنهـا تعاملـه على أسـاس شـريحة من السلاسل النصية `[]string`، وبالتالي يمكننا التكرار عليه بحلقة `for` من خلال استخدام العامِل `range`.

ستحصل عند تنفيذ هذه الشيفرة على ما يلي:

```
Hello Sammy
Hello Sammy
Hello Jessica
Hello Drew
Hello Jamie
```

لاحظ أنه في المرة الأولى التي استدعينا فيها الدالة sayHello لم يُطبع أيّ شيء، وذلك لأننا لم نمرر لها أيّ قيمة، أي عمليًّا هي شريحة فارغة، وبالتالي لا تتضمن أي قيمة ليُكرر عليها، والآن سنُعدّل الشيفرة السابقة بحيث تطبع عبارةً مُحددةً عندما لا تُمرّر أيّ قيمة:

```
package main
import "fmt"
func main() {
  sayHello()
  sayHello("Sammy")
  sayHello("Sammy", "Jessica", "Drew", "Jamie")
}
func sayHello(names ...string) {
  if len(names) == 0 {
        fmt.Println("nobody to greet")
        return
  }
  for _, n := range names {
        fmt.Printf("Hello %s\n", n)
  }
}
```

أضفنا تعليمة if تتحقق مما إذا كانت الشريحة فارغةً وهذا يُكافئ أن يكون طولها 0، وفي هذه الحالة سنطبع السلسة النصية nobody to greet:

```
nobody to greet
Hello Sammy
Hello Sammy
Hello Jessica
Hello Drew
Hello Jamie
```

يجعل استخدام الدوال والمتغيرات المرنة شيفرتك أكثر قابليةً للقراءة، وسنُنشئ الآن دالةً مرنةً تربط الكلمات اعتمادًا على رمز مُحدّد، لكن سنكتب أولًا دالةً ليست مرنة لنُبيّن الفرق:

```
package main
import "fmt"
func main() {
```

```
    var line string
    line = join(",", []string{"Sammy", "Jessica", "Drew", "Jamie"})
    fmt.Println(line)
    line = join(",", []string{"Sammy", "Jessica"})
    fmt.Println(line)
    line = join(",", []string{"Sammy"})
    fmt.Println(line)
}
func join(del string, values []string) string {
    var line string
    for i, v := range values {
          line = line + v
          if i != len(values)-1 {
                line = line + del
          }
    }
    return line
}
```

مرّرنا هنا الفاصلة , إلى الدالة join لكي يُنجز الربط اعتمادًا عليها، ثم مرّرنا شريحةً من الكلمات لكي تُربط، فكان الخرج كما يلي:

```
Sammy,Jessica,Drew,Jamie
Sammy,Jessica
Sammy
```

لاحظ هنا أنّ تعريف شريحة في كل مرة نستدعي فيها هذه الدالة قد يكـون مُملًا وأكـثر صـعوبةً في القـراءة، لذا سنُعرّف الآن الشيفرة نفسها لكن مع دالة مرنة:

```
package main
import "fmt"
func main() {
    var line string
    line = join(",", "Sammy", "Jessica", "Drew", "Jamie")
    fmt.Println(line)
    line = join(",", "Sammy", "Jessica")
    fmt.Println(line)
```

```
    line = join(",", "Sammy")
    fmt.Println(line)
}
func join(del string, values ...string) string {
    var line string
    for i, v := range values {
        line = line + v
        if i != len(values)-1 {
            line = line + del
        }
    }
    return line
}
```

سنحصل عند تشغيل البرنامج على خرج المثال السابق نفسه:

```
Sammy,Jessica,Drew,Jamie
Sammy,Jessica
Sammy
```

يمكن بسهولة ملاحظة أنّ استخدام مفهوم الدالة المرنة قد جعل من الدالة `join` أكثر قابليةً للقراءة.

17.7 ترتيب الوسائط المرنة

يمكنك تعريف معامِل مرن واحدة فقط في الدالة، كما يجب أن يكون هو المعامِل الأخير في ترويسة الدالة، فإذا عرّفتَ أكثر من معامِل مرن أو وضعته قبل المعامِلات العادية، فسيظهر لك خطأ وقت التصريف compilation error.

```
package main
import "fmt"
func main() {
    var line string
    line = join(",", "Sammy", "Jessica", "Drew", "Jamie")
    fmt.Println(line)
    line = join(",", "Sammy", "Jessica")
    fmt.Println(line)
    line = join(",", "Sammy")
    fmt.Println(line)
```

```
}
func join(values ...string, del string) string {
  var line string
  for i, v := range values {
        line = line + v
        if i != len(values)-1 {
              line = line + del
        }
  }
  return line
}
```

وضعنا هنا المعامِل المرن `values` أولًا ثم وضعنا المعامِل العـادي `del`، وبالتـالي خالفنـا الشـرط المـذكور سلفًا، وبالتالي سنحصل على الخطأ التالي:

```
./join_error.go:18:11: syntax error: cannot use ... with non-final
parameter values
```

عند تعريف دالة مرنة لا يمكن أن يكون المعامِل الأخير إلا معاملًا مرنًا.

17.8 تفكيك الوسائط

رأينا كيف أنّ المعامل المرن سيسمح لنا بتمرير 0 قيمـة أو قيمـة واحـدة أو أكـثر من قيمـة إلى الدالـة، لكن هناك حالات سيكون لدينا فيها شريحة من القيم نريد تمريرها إلى الدالة المرنة، لـذا دعونـا نـرى الدالـة `join()` التي بنيناها مؤخرًا لرؤية ما يحدث:

```
package main
import "fmt"
func main() {
  var line string
  names := []string{"Sammy", "Jessica", "Drew", "Jamie"}
  line = join(",", names)
  fmt.Println(line)
}
func join(del string, values ...string) string {
  var line string
  for i, v := range values {
        line = line + v
```

```
        if i != len(values)-1 {
            line = line + del
        }
    }
    return line
}
```

سنحصل عند تشغيل البرنامج على خطأ وقت التصريف:

```
./join-error.go:10:14: cannot use names (type []string) as type string
in argument to join
```

على الرغم من أن الدالة المرنة ستحوّل المعامِل `...string values` إلى شريحة من السلاسل النصية `[]string`، إلا أنّ هذا لا يعني أنه بإمكاننا تمرير شريحة من السلاسل على أساس وسيط، فالمُصرّف يتوقع وسائط منفصلةً من نوع سلسلة نصية.

يمكننا لحل هذه المشكلة تفكيك قيم الشريحة -أي فصلها عمليًّا- من خلال وضع ثلاثة نقط بعد اسم الشريحة عندما نُمررها لدالة مرنة.

```
package main
import "fmt"
func main() {
    var line string
    names := []string{"Sammy", "Jessica", "Drew", "Jamie"}
    line = join(",", names...)
    fmt.Println(line)
}
func join(del string, values ...string) string {
    var line string
    for i, v := range values {
        line = line + v
        if i != len(values)-1 {
            line = line + del
        }
    }
    return line
}
```

لاحظ أننا وضعنا 3 نقاط بعد اسم الشريحة `names` عندما مررناها إلى الدالة `join()`، وهذا يؤدي إلى تفكيك عناصر الشريحة، وبالتالي كأنها قيم منفصلة، وسيكون الخرج كما يلي:

```
Sammy,Jessica,Drew,Jamie
```

لاحظ أنه مازال بإمكاننا عدم تمرير أيّ شيء أو تمرير أيّ عدد نريده من القيم، ويبيّن المثال التالي كل الحالات:

```
package main
import "fmt"
func main() {
  var line string
  line = join(",", []string{"Sammy", "Jessica", "Drew", "Jamie"}...)
  fmt.Println(line)
  line = join(",", "Sammy", "Jessica", "Drew", "Jamie")
  fmt.Println(line)
  line = join(",", "Sammy", "Jessica")
  fmt.Println(line)
  line = join(",", "Sammy")
  fmt.Println(line)
}
func join(del string, values ...string) string {
  var line string
  for i, v := range values {
        line = line + v
        if i != len(values)-1 {
              line = line + del
        }
  }
  return line
}
```

سيكون الخرج كما يلي:

```
Sammy,Jessica,Drew,Jamie
Sammy,Jessica,Drew,Jamie
Sammy,Jessica
Sammy
```

أصبحنا الآن نعرف كيف نمرِّر 0 قيمة أو أكثر إلى دالة مرنة، كما أصبحنا نعرف كيف يمكننا تمرير شريحة إلى دالة مرنة.

غالبًا ما تكون الدوال المرنة مفيدةً في الحالات التالية:

- عندما تكون بحاجة إلى تعريف شريحة مؤقتًا من أجل تمريرها إلى دالة.
- لجعل الشيفرة أكثر قابليةً للقراءة.
- عندما يكون عدد الوسائط غير معروف أو متغير مثل دالة الطباعة.

17.9 الخاتمة

تعرفت في هذا الفصل على مفهوم الدوال في لغة جو والتي تُعَدّ كتلًا من التعليمات البرمجية التي تُنفِّذ إجراءات معيّنة داخل البرنامج، كما تساعد على جعل الشفرة تركيبيةً وقابلةً لإعادة الاستخدام، كما أنها تنظمها وتسهل من قراءتها، كما تعرفت أيضًا على مفهوم الدوال المرنة التي الشيفرة أكثر قابليةً للقراءة، إلا أنها ليست شائعة الاستخدام. ولتتعلم كيف تجعل برنامجك تركيبيًّا أكثر، فيمكنك الاطلاع على فصل كيفية التعامل مع الحزم في لغة جو Go.

هل تطمح لبيع منتجاتك الرقمية عبر الإنترنت؟

استثمر مهاراتك التقنية وأطلق منتجًا رقميًا
يحقق لك دخلًا عبر بيعه على متجر بيكاليكا

أطلق منتجك الآن

18. تعرف على التعليمة defer

تتضمن لغة جو العديد من تعليمات التحكم بسير عمل البرنامج التي تعرفنا عليها في الفصول اللاحقة مثل `if` و `switch` و `for`. وهذه التعليمات موجودة في أغلب لغات البرمجة، إلا أنّ هناك تعليمة خاصة في لغة جو غير موجودة في معظم لغات البرمجة الأخرى وهي تعليمة التأجيل `defer`.

على الرغم من أن هذه التعليمة ليست مشهورة إلا أنها مفيدة، فالهدف الأساسي من هذه التعليمة هو إجراء عملية تنظيف الموارد بطريقة سليمة بعد انتهاء استخدامها مثل الملفات المفتوحة أو الاتصالات بالشبكة أو مقابض قاعدة البيانات handles بعد تأجيلها ريث الانتهاء منها، إذ تُعَدّ عملية تنظيف أو تحرير الموارد بعد استخدامها أمرًا مهمًا جدًا للسماح للمستخدِمين الآخرين باستخدام هذا المورد دون وجود أي بقايا تركها المستخدِم السابق سواءً كان المستخدِم آلةً أو شخصًا أو برنامجًا أو جزءًا آخر من الشيفرة نفسها.

تساعدنا تعليمة التأجيل `defer` في جعل الشيفرة البرمجية أنظف وأكثر متانةً وأقل عرضةً للأخطاء من خلال جعل تعليمة استدعاء أو حجز الملف أو المورد قريبة من تعليمة تحريره، وفي هذا الفصل سنتعلم كيفية استخدام تعليمة التأجيل `defer` بطريقة صحيحة لتنظيف الموارد، كما سنستعرض العديد من الأخطاء الشائعة التي تحدث عند استخدام هذه التعليمة.

18.1 ما هي تعليمة defer؟

تسمح لك جو بتأجيل تنفيذ استدعاء دالة بإضافتها إلى طابور تنفيذ خاص ريثما تكون الدالة المُستدعاة ضمنها قد أكتمل تنفيذها من خلال استخدام الكلمة المفتاحية `defer` قبلها بالشكل التالي:

```
package main
import "fmt"
func main() {
```

```
    defer fmt.Println("Bye")
    fmt.Println("Hi")
}
```

أجّلنا هنا استخدام الدالة `fmt.Println("Bye")` إلى حين انتهاء تنفيذ الدالة المُستدعاة ضمنها أي الدالة `main`، وبالتالي سيكون الخرج كما يلي:

```
Hi
Bye
```

أي سيُنفّذ هنا كل شيء داخل الدالة `main` ثم بعد الانتهاء من كل التعليمات (هنا لا توجد إلا تعليمة واحدة هي اتعليمة التي تطبع `Hi`) ستُنفَّذ التعليمة المؤجلة، كما رأيت بالمثال.

والآن قد يتبادر إلى الأذهان السؤال التالي: ماذا لو كان هناك أكثر من استدعاء مؤجل، أي أكثر من دالة مؤجلة، ماذا سيحدث؟ كل دالة تُسبق بتعليمة التأجيل تُضاف إلى مكدس، ومن المعروف أنّ المكدس هو بنية معطيات تخرج منها البيانات وفق مبدأ مَن يدخل آخرًا يخرج أولًا، وبالتالي إذا كان لدينا أكثر من استدعاء مؤجل، فسيكون التنفيذ وفق هذه القاعدة، ولجعل الأمور أبسط سنأخذ المثال التالي:

```
package main
import "fmt"
func main() {
  defer fmt.Println("Bye1")
  defer fmt.Println("Bye2")
  fmt.Println("Hi")
}
```

سيكون الخرج كما يلي:

```
Hi
Bye2
Bye1
```

لدينا في البرنامج السابق استدعاءان مؤجلان، إذ يُضاف أولًا إلى المكدس التعليمة التي تطبع `Bye1` ثم التعليمة التي تطبع `Bye2`، ووفق القاعدة السابقة ستُطبع التعليمة التي أُضيفت أخيرًا إلى المكدس أي `Bye2` ثم `Bye1`، وبالطبع تُنفَّذ التعليمة التي تطبع `Hi` وأيّ تعليمة أخرى ضمن الدالة `main()` قبل تنفيذ أيّ تعليمة مؤجلة، ولدينا مثال آخر كما يلي:

```
package main
import "fmt"
func main() {
  fmt.Println("Hi0")
  defer fmt.Println("Bye1")
  defer fmt.Println("Bye2")
  fmt.Println("Hi1")
  fmt.Println("Hi2")
}
```

سيكون الخرج كما يلي:

```
Hi0
Hi1
Hi2
Bye2
Bye1
```

عند تأجيل تنفيذ دالة ما ستقيَّم الوسائط على الفور، وبالتالي لن يؤثر أيّ تعـديل لاحـق، لـذا انتبـه جيـدًا إلى المثال التالي:

```
package main
import "fmt"
func main() {
  x := 9
  defer fmt.Println(x)
  x = 10
}
```

سيكون الخرج كما يلي:

```
9
```

كانت هنا قيمة x تساوي 9 ثم أجلنا تنفيذ دالة الطباعة التي تطبـع قيمـة هـذا المتغـير، وعلى الـرغم من أنّ تعليمة x=10 ستُنفَّذ قبـل تعليمـة الطباعـة، إلا أنّ القيمـة الـتي طُبعَت هي القيمـة السـابقة للمتغـير x وذلـك للسبب الذي ذكرناه منذ قليل.

ما تعلمناه حتى الآن كان بغرض التوضيح فقط، إذ لا تُستخدم بهـذا الشـكل بـل عـادة مـا تُسـتخدم تعليمـة التأجيل في تنظيف الموارد وهذا ما سنراه تاليًا.

18.2 تنظيف الموارد باستخدام تعليمة التأجيل

يُعَدّ استخدام تعليمة التأجيل لتنظيف الموارد أمرًا شائعًا جدًا في جو، وسنلقي الآن نظرةً على برنـامج يكتب سلسلةً نصيةً في ملف ولكنه لا يستخدِم تعليمة التأجيل لتنظيف المورد:

```
package main
import (
  "io"
  "log"
  "os"
)
func main() {
  if err := write("readme.txt", "This is a readme file"); err != nil {
        log.Fatal("failed to write file:", err)
  }
}
func write(fileName string, text string) error {
  file, err := os.Create(fileName)
  if err != nil {
        return err
  }
  _, err = io.WriteString(file, text)
  if err != nil {
        return err
  }
  file.Close()
  return nil
}
```

لدينا في هذا البرنامج دالة تسمى `write` والهدف منها بدايةً هو محاولة إنشاء ملف، وإذا حصل خطـأ أثنـاء هذه المحاولة، فستُعيد هذا الخطأ وتُنهي التنفيذ؛ أما في حـال نجـاح عمليـة إنشـاء الملـف، فسـوف تكتب فيـه السلسلة `This is a readme file`، وفي حال فشلت عملية الكتابة، فستُعيد الخطأ وتنهي تنفيذ الدالة أيضًا، وأخيرًا إذا نجح كل شيء، فستغلق الدالة الملف الذي أُنشِئ معيدة إياه إلى نظام الملفـات ثم تُعيـد القيمـة `nil` إشارةً إلى نجاح التنفيذ.

يعمل هذا البرنامج بطريقة سليمة، إلا أنّ هناك علّة برمجية صغيرة؛ فإذا فشلت عمليـة الكتابـة في الملـف، فسيبقى الملف المُنشئ مفتوحًا، أي توجد هناك موارد غير محررة، ولحل هـذه المشـكلة مبـدأيًا يمكنـك تعليمـة `file.Close()` أخرى ومن دون استخدام تعليمة التأجيل:

```
package main
import (
  "io"
  "log"
  "os"
)
func main() {
  if err := write("readme.txt", "This is a readme file"); err != nil {
        log.Fatal("failed to write file:", err)
  }
}
func write(fileName string, text string) error {
  file, err := os.Create(fileName)
  if err != nil {
        return err
  }
  _, err = io.WriteString(file, text)
  if err != nil {
        file.Close()
        return err
  }
  file.Close()
  return nil
}
```

الآن سيُغلق الملف حتى إذا فشل تنفيذ `io.WriteString`، وقد يكون هذا حلًا بسيطًا وأسهل من غـيره، لكن إذا كانت الدوال أكثر تعقيدًا، فقد ننسى إضافة هكذا تعليمات في الأمـاكن المُحتملـة الـتي قـد ينتهي فيهـا التنفيـذ قبـل تحريـر المـورد، لـذا يتمثّـل الحـال الأكـثر احترافيـةً وأمانًـا باسـتخدام تعليمـة التأجيـل مـع الدالـة `file.Close` وبالتالي ضمان تنفيذها دومًا بغض النظر عن أيّ مسار تنفيذ أو خطأ مُفاجئ قد ينهي الدالة:

```
package main
import (
  "io"
```

```
    "log"
    "os"
)
func main() {
    if err := write("readme.txt", "This is a readme file"); err != nil {
        log.Fatal("failed to write file:", err)
    }
}
func write(fileName string, text string) error {
    file, err := os.Create(fileName)
    if err != nil {
        return err
    }
    defer file.Close()
    _, err = io.WriteString(file, text)
    if err != nil {
        return err
    }
    return nil
}
```

أضفنا هنا السطر `defer file.Close` لإخبار المُصرّف بإغلاق الملف بعد انتهاء تنفيذ الدالة، وبالتالي ضمان تحريره مهما حدث.

وبالرغم من أننا قد انتهينا هنا من علّة برمجية ولكن ستظهر لنا علّة برمجية أخرى؛ ففي حال حدث خطأ في عملية إغلاق الملف `file.Close`، فلن تكون هناك أيّ معلومات حول ذلك الخطأ، أي لن يُعاد الخطأ، وبالتالي سوف يمنعنا استخدام تعليمة التأجيل `defer` من الحصول على الخطأ أو الحصول على أيّ قيمة تُعيدها الدالة المؤجلة.

يُعَدّ استدعاء الدالة `Close()` في جو أكثر من مرة آمنًا ولا يؤثر على سلوك البرنامج، وفي حال كان هناك خطأ، فسيُعاد من المرة الأولى التي تُستدعى فيها الدالة، وبالتالي سيسمح لنا هذا باستدعائها ضمن مسار التنفيذ الناجح في الدالة.

سنعالج في المثال التالي المشكلة السابقة التي تتجلى بعدم إمكانية الحصول على معلومات الخطأ في حال حدوثه مع تعليمة الإغلاق المؤجلة:

```
package main
import (
  "io"
  "log"
  "os"
)
func main() {
  if err := write("readme.txt", "This is a readme file"); err != nil {
        log.Fatal("failed to write file:", err)
  }
}
func write(fileName string, text string) error {
  file, err := os.Create(fileName)
  if err != nil {
        return err
  }
  defer file.Close()
  _, err = io.WriteString(file, text)
  if err != nil {
        return err
  }
  return file.Close()
}
```

لاحظ أنّ التعديل الوحيد على الشيفرة السابقة كان بإضافة السطر `return file.Close()`، وبالتالي إذا حصل خطأ في عملية الإغلاق، فسيُعاد مباشرةً إلى الدالة التي تستدعي الدالة `write`، ولاحظ أيضًا أنّ التعليمة `defer file.Close` ستُنفّذ بكافة الأحوال، مما يعني أنّ تعليمة الإغلاق ستُنفّذ مرتين غالبًا، وعلى الرغم من أنّ ذلك ليس مثاليًّا، إلا أنه مقبول وآمن.

إذا ظهر خطأ في تعليمة `WriteString`، فسيُعاد الخطأ وينتهي تنفيذ الدالة ثم ستُنفّذ تعليمة `file.Close` لأنها تعليمة مؤجلة، وهنا على الرغم من إمكانية ظهور خطأ عند محاولة إغلاق الملف، إلا أنّ هذا الخطأ لم يَعُد مهمًا لأن الخطأ الذي تعيده `WriteString` سيكون غالبًا هو السبب وراء حدوثه.

تعرّفنا حتى الآن على كيفية استخدام تعليمة التأجيل واحدة لضمان تحرير أو تنظيف مورد ما، وسنتعلّم الآن كيفية استخدام عدة تعليمات تأجيل من أجل تنظيف أكثر من مورد.

18.3 استخدام تعليمات تأجيل متعددة

من الطبيعي أن تكون لـديك أكـثر من تعليمـة `defer` في دالـة مـا، لـذا دعنـا ننشـئ برنامجًـا يحتـوي على تعليمات تأجيل فقط لنرى ما سيحدث في هذه الحالة (كما أشرنا سابقًا):

```
package main
import "fmt"
func main() {
  defer fmt.Println("one")
  defer fmt.Println("two")
  defer fmt.Println("three")
}
```

سيكون الخرج كما يلي:

```
three
two
one
```

كمـا ذكرنـا سـابقًا أنّ سـبب الطباعـة بهـذا الـترتيب هـو أن تعليمـة التأجيـل تعتمـد على تخـزين سلسـلة الاستدعاءات ضمن بنيـة المكـدس، والمهم الآن أن تعـرف أنـه يمكن أن يكـون لـديك العديـد من الاسـتدعاءات المؤجلة حسب الحاجة في دالة ما، ومن المهم أن تتذكر أنها تُستدعى جميعها بعكس ترتيبها في الشيفرة حسب بنية المكدس.

الآن بعد أن فهمنا هذه الفكرة جيدًا، سننشئ برنامجًـا يفتح ملفًـا ويكتب عليـه ثم يفتحـه مـرةً أخـرى لنسـخ المحتويات إلى ملف آخر:

```
package main
import (
  "fmt"
  "io"
  "log"
  "os"
)
func main() {
  if err := write("sample.txt", "This file contains some sample
text."); err != nil {
        log.Fatal("failed to create file")
```

```
    }
    if err := fileCopy("sample.txt", "sample-copy.txt"); err != nil {
          log.Fatal("failed to copy file: %s")
    }
}

func write(fileName string, text string) error {
    file, err := os.Create(fileName)
    if err != nil {
          return err
    }
    defer file.Close()
    _, err = io.WriteString(file, text)
    if err != nil {
          return err
    }
    return file.Close()
}
func fileCopy(source string, destination string) error {
    src, err := os.Open(source)
    if err != nil {
          return err
    }
    defer src.Close()
    dst, err := os.Create(destination)
    if err != nil {
          return err
    }
    defer dst.Close()
    n, err := io.Copy(dst, src)
    if err != nil {
          return err
    }
    fmt.Printf("Copied %d bytes from %s to %s\n", n, source,
destination)
    if err := src.Close(); err != nil {
          return err
```

```
    }
    return dst.Close()
}
```

لدينا هنا دالة جديدة `fileCopy` تفتح أولًا ملف المصدر الذي سننسخ منه ثم تتحقق من حدوث خطأ أثناء فتح الملف، فإذا كان الأمر كذلك، فسنعيد الخطأ ونخرج من الدالة، وإلا فإننا نؤجل إغلاق الملف المصدر الذي فتحناه.

نُنشئ بعد ذلك ملف الوجهة ونتحقق من حدوث خطأ أثناء إنشاء الملف، فإذا كان الأمر كذلك، فسنعيد هذا الخطأ ونخرج من الدالة، وإلا فسنؤجل أيضًا إغلاق ملف الوجهة، وبالتالي لدينا الآن دالتان مؤجلتان ستُستدعيان عندما تخرج الدالة من نطاقها.

الآن بعد فتح كل من ملف المصدر والوجهة فإننا سننسخ `Copy()` البيانات من الملف المصدر إلى الملف الوجهة، فإذا نجح ذلك، فسنحاول إغلاق كلا الملفين، وإذا تلقينا خطأً أثناء محاولة إغلاق أيّ ملف، فسنُعيد الخطأ ونخرج من الدالة.

لاحظ أننا نستدعي `Close()` لكل ملف صراحةً على الرغم من أنّ تعليمة التأجيل ستستدعيها أيضًا، والسبب وراء ذلك هو كما ذكرناه في الفقرة السابقة؛ وهو للتأكد من أنه إذا كان هناك خطأ في إغلاق ملف، فإننا نُعيد معلومات هذا الخطأ، كما يضمن أنه إذا أُنهيَت الدالة مُبكرًا لأيّ سبب من الأسباب بسبب خطأ ما مثل الفشل في عملية النسخ بين الملفين، فسيحاول كل ملف الإغلاق بطريقة سليمة من خلال تنفيذ الاستدعاءات المؤجلة.

18.4 الخاتمة

تحدثنا في هذا الفصل عن تعليمة التأجيل `defer` وكيفية استخدامها والتعامل معها وأهميتها في تنظيف وتحرير الموارد بعد الانتهاء من استخدامها، وبالتالي ضمان توفير الموارد للمستخدِمين المختلِفين وتوفير الذاكرة. كما يمكنك أيضًا الرجوع إلى قسم معالجة حالات الانهيار في لغة جو Go والاطلاع على حالةً خاصةً من حالات استخدام تعليمة التأجيل.

19. تعرف على دالة التهيئة init

تُستخدَم الدالة المُعرّفة مسبقًا `init()` في لغة جو لجعل شيفرة مُحدَّدة تُنفّذ قبل أيّ شيفرة أخرى ضمن الحزمة الخاصة بك، إذ ستُنفَّذ هذه الشيفرة عند استيراد الحزمة مباشرةً، وبالتالي يمكنك استخدام هذه الدالة عندما تحتاج إلى تهيئة تطبيقك في حالة معينة مثل أن يكون لديك إعدادات أوليّة محددة أو مجموعة من الموارد التي يحتاجها تطبيقك لكي يبدأ.

تُستخدم أيضًا عند استيراد تأثير جانبي side effect، وهي تقنية تستخدَم لضبط حالة البرنامج من خلال استيراد حزمة معينة، ويُستخدَم هذا غالبًا من أجل تسجيل `register` حزمة مع أخرى للتأكد من عمل البرنامج بطريقة صحيحة.

يُعَدّ استخدام الدالة `init()` مفيدًا، ولكنه يجعل من قراءة الشيفرة أمرًا صعبًا في بعض الأحيان وذلك لأنّ نسخة `init()` التي يصعُب العثور عليها ستؤثر إلى حد كبير في ترتيب تنفيذ شفرات البرنامج، وبالتالي لا بدّ من فهم هذه الدالة جيدًا لاستخدامها بطريقة صحيحة.

سنتعلم في هذا الفصل استخدام الدالة `init()` لإعداد وتهيئة متغيرات حزمة معيّنة وعمليات حسابية تجرى لمرة واحدة وتسجيل حزمة لاستخدامها مع حزمة أخرى.

19.1 المتطلبات

ستحتاج لامتلاك مساحة عمل خاصة مُهيئة وجاهزة في لغة جو، فإذا لم يكن لديك واحدة، فارجع للتعليمات الواردة في الفصل الأول من الكتاب، واعمل على تثبيت لغة جو Go لديك وإعداد بيئة تطوير محلية مناسبة بحسب نظام تشغيلك، إذ يستخدِم هذا الفصل بنية الملفات التالية:

```
.
├── bin
│
└── src
    └── github.com
        └── gopherguides
```

19.2 التصريح عن الدالة ()init

بمجرّد التصريح عن أيّ دالة `init()` سيُنفّذها مصرِّف جو قبل أيّ شيفرة أخرى في الحزمة، وسنوضِّح في هذا القسم كيفية تعريف هذه الدالة وكيف تؤثر على تشغيل الحزمة، لذا دعونا الآن نلقي نظرةً على مثال لا يتضمن هذه الدالة:

```
package main
import "fmt"
var weekday string
func main() {
  fmt.Printf("Today is %s", weekday)
}
```

صرّحنا في هذا البرنامج عن متغير عام `weekday` من النوع `string`، وبما أننا لم نُهيّئ هذا المتغير بأيّ قيمة، فستكون القيمة الافتراضية لها هي السلسلة الفارغة، وبالتالي لن نرى أيّ شيء عند طباعتها، ولنشغّل هذه الشيفرة كما يلي:

```
$ go run main.go
```

سيكون الخرج كما يلي:

```
Today is
```

يمكننا إعطاء قيمة أوليّة إلى المتغير `weekday` من خلال الدالة `init()` بحيث نُهيّئها بقيمة اليوم الحالي، لذا سنعدِّل على الشيفرة السابقة كما يلي:

```
package main
import (
  "fmt"
  "time"
)
```

```
var weekday string
func init() {
  weekday = time.Now().Weekday().String()
}
func main() {
  fmt.Printf("Today is %s", weekday)
}
```

استخدمنا في هذه الشيفرة الحزمة `time` للحصول على اليوم الحالي من خلال كتابة `Now().Weekday().String()` ثم استخدمنا الدالة `init()` لتهيئة المتغير `weekday` بها، والآن إذا شغّلنا البرنامج، فسنحصل على خرج مُختلف عن المرة السابقة:

```
Today is Monday
```

كان الهدف من هذا المثال هو توضيح عمل الدالة، إلا أن استخدامها الأكثر شيوعًا يكون عند استيراد حزمة، فقد نكون بحاجة إلى إتمام بعض عمليات التهيئة أو بعض العمليات الأولية قبل استخدام الحزمة، لذا دعنا ننشئ برنامجًا سيتطلب تهيئة محددة للحزمة لتعمل على النحو المنشود.

19.3 تهيئة الحزم عند استيرادها

بدايةً، سنكتب برنامجًا بسيطًا يختار عشوائيًا عنصرًا من شريحة ما ويطبعها، ولن نستخدِم هنا الدالة `init()` لكي نُبيّن المشكلة التي ستحدث وكيف ستحلّها هذه الدالة فيما بعد، لذا أنشئ مجلدًا اسمه `creature` من داخل المجلد `src/github.com/gopherguides/` كما يلي:

```
$ mkdir creature
```

أنشئ بداخله الملف creature.go:

```
$ nano creature/creature.go
```

ضع بداخل الملف التعليمات التالية:

```
package creature
import (
  "math/rand"
)
var creatures = []string{"shark", "jellyfish", "squid", "octopus",
"dolphin"}
func Random() string {
```

```
    i := rand.Intn(len(creatures))
    return creatures[i]
}
```

يُعرِّف هذا الملف متغيرًا يسمى `creatures` يحتوي على مجموعة من أسماء الكائنات البحرية على أساس قيم لهذه الشريحة، كما يحتوي هذا الملف على دالة عشوائية مُصدّرة تُعيد قيمةً عشوائيةً من هذه الشريحة. احفظ الملف واخرج منه.

سننشئ الآن الحزمة `cmd` التي سنستخدِمها لكتابة الدالة `main()` واستدعاء الحزمة `creature`، ولأجل ذلك يجب أن نُنشئ مجلدًا اسمه cmd بجانب المجلد creature كما يلي:

```
$ mkdir cmd
```

سنُنشئ بداخله الملف main.go:

```
$ nano cmd/main.go
```

أضف المحتويات التالية إلى المجلد:

```
package main
import (
    "fmt"
    "github.com/gopherguides/creature"
)
func main() {
    fmt.Println(creature.Random())
    fmt.Println(creature.Random())
    fmt.Println(creature.Random())
    fmt.Println(creature.Random())
}
```

استوردنا هنا الحزمة `creature`، ثم استدعينا الدالة `creature.Random()` بداخل الدالة `main()` للحصول على عنصر عشوائي من الشريحة وطباعته أربع مرات. احفظ الملف main.go ثم أغلقه.

انتهينا الآن من كتابة الشيفرة، لكن قبل أن نتمكن من تشغيل هذا البرنامج، سنحتاج أيضًا إلى إنشاء ملفَين من ملفات الضبط configuration حتى تعمل التعليمات البرمجية بطريقة صحيحة.

تُستخدَم وحدات لغة جو Go Modules لضبط اعتماديات الحزمة لاستيراد الموارد، إذ تُعَدّ وحدات لغة جو ملفات ضبط موضوعة في مجلد الحزمة الخاص بك والتي تخبر المُصرِّف بمكان استيراد الحزم منه، ولن نتحدث

عن وحدات لغة جو في هذا الفصل، لكن سنكتب السطرين التاليين لكي تعمل الشيفرة السابقة، لذا أنشئ الملف go.mod ضمن المجلد cmd كما يلي:

```
$ nano cmd/go.mod
```

ثم ضع بداخله التعليمات التالية:

```
module github.com/gopherguides/cmd
 replace github.com/gopherguides/creature => ../creature
```

يخبر السطر الأول المصرِّف أنّ الحزمة cmd التي أنشأناها هي في الواقع github.com/gopherguides/cmd؛ أما السطر الثاني، فيُخبر المصرِّف أنه يمكن العثور على المجلد github.com/gopherguides/creature محليًا ضمن المجلد ../creature، لذا احفظ وأغلق الملف ثم أنشئ ملف go.mod في مجلد creature:

```
$ nano creature/go.mod
```

أضف السطر التالي من التعليمات البرمجية إلى الملف:

```
module github.com/gopherguides/creature
```

يخبر هذا المصرِّف أنّ الحزمة creature التي أنشأناها هي في الواقع الحزمة github.com/gopherguides/creature، وبدون ذلك لن تعرف الحزمة cmd مكان استيراد هذه الحزمة. احفظ وأغلق الملف.

يجب أن يكون لديك الآن بنية المجلد التالية:

```
├── cmd
│   ├── go.mod
│   └── main.go
└── creature
    ├── go.mod
    └── creature.go
```

الآن بعد أن انتهينا من إنشاء ملفات الضبط اللازمة أصبح بالإمكان تشغيل البرنامج من خلال الأمر التالي:

```
$ go run cmd/main.go
```

سيكون الخرج كما يلي:

```
jellyfish
squid
squid
dolphin
```

حصلنا عند تشغيل هذا البرنامج على أربع قيم وطبعناها، وإذا أعدنا تشغيل هذا البرنامج عدة مرات، فسنلاحظ أننا نحصل دائمًا على الخرج نفسه بدلًا من نتيجة عشوائية كما هو متوقع، وسبب ذلك هو أنّ الحزمة `rand` تُنشئ أعدادًا شبه عشوائية تولِّد الخرج نفسه باستمرار من أجل حالة أولية initial state واحدة.

للحصول على عشوائية أكبر، يمكننا إعادة ضبط مفتاح توليد الأعداد seed في الحزمة، أو ضبط مصدر متغير بحيث تختلف الحالة الأولية في كل مرة نُشغّل فيها البرنامج، وفي لغة جو من الشائع استخدام الوقت الحالي على أساس مفتاح توليد في حزمة `rand`، وبما أننا نريد من الحزمة `creature` التعامل مع دالة عشوائية، افتح هذا الملف:

```
$ nano creature/creature.go
```

أضف التعليمات التالية إليه:

```
package creature
import (
  "math/rand"
  "time"
)
var creatures = []string{"shark", "jellyfish", "squid", "octopus",
"dolphin"}
func Random() string {
  rand.Seed(time.Now().UnixNano())
  i := rand.Intn(len(creatures))
  return creatures[i]
}
```

استوردنا في هذا البرنامج الحزمة `time` واستخدمنا الدالة `Seed()` لضبط مفتاح توليد الأعداد seed في الحزمة إلى وقت التنفيذ آنذاك. احفظ وأغلق الملف.

قم الآن بتشغيل البرنامج:

```
$ go run cmd/main.go
```

ستحصل على ما يلي:

```
jellyfish
octopus
shark
jellyfish
```

إذا شغلت البرنامج عدة مرات، فستحصل على نتائج مختلفة في كل مرة، وعمومًا هذا ليس نهجًا مثاليًا لأن الدالة `creature.Random()` تُستدعى في كل مرة ولأنه يُعاد ضبط مفتاح توليد الأعداد من خلال الاستدعاء `rand.Seed(time.Now().UnixNano())`، وإعادة ضبط مفتاح التوليد re-seeding باستمرار قد يصادف استعمال قيمتين متماثلتين في حالتنا لو استدعي أكثر من مرة بدون تغير الوقت، أي إذا لم تتغير الساعة الداخلية، وهذا سيؤدي إلى تكرار محتمل للنمط العشوائي أو سيزيد من وقت المعالجة في وحدة المعالجة المركزية عن طريق جعل البرنامج ينتظر حتى تتغير الساعة الداخلية، ولحل هذه المشكلة يمكننا استخدام الدالة `init()`، لذا سنعدّل الملف creature.go:

```
$ nano creature/creature.go
```

أضف ما يلي إلى الملف:

```
package creature
import (
  "math/rand"
  "time"
)
var creatures = []string{"shark", "jellyfish", "squid", "octopus",
"dolphin"}
func init() {
  rand.Seed(time.Now().UnixNano())
}
func Random() string {
  i := rand.Intn(len(creatures))
  return creatures[i]
}
```

إضافة الدالة `init()` ستخبر المصرّف أنه يجب استدعاؤها عند استيراد الحزمة `creature` ولمرة واحدة فقط، وبالتالي يكون لدينا مفتاح توليد seed واحد لتوليد العدد العشوائي، ويجنبنا هذا النهج تكرار تنفيذ التعليمات البرمجية، والآن إذا أعدنا تشغيل الشيفرة عدة مرات، فسنحصل على نتائج مختلفة:

```
$ go run cmd/main.go
```

سيكون الخرج كما يلي:

```
dolphin
squid
dolphin
octopus
```

رأينا في هذا القسم كيف يمكّننا استخدام الدالة `init()` من إجراء العمليات الحسابية أو التهيئة المناسبة قبل استخدام الحزمة، وسنرى فيما يلي كيف يمكننا استخدام عدة تعليمات تهيئة `init()` في حزمة ما.

19.4 استخدام عدة تعليمات من ()init

يمكن التصريح عن الدالة `init()` أكثر من مرة ضمن الحزمة على عكس الدالة `main()` التي لا يمكن التصريح عنها إلا مرةً واحدةً، وعمومًا يؤدي وجود أكثر من دالة تهيئة إلى حدوث التباس في أولوية التنفيذ، أي مَن يُنفَّذ أولًا ومَن لا يُنفَذ، لذا سيوضح هذا القسم كيفية استخدام تعليمات تهيئة متعددة وكيفية التحكم بها.

ستُنفّذ عادةً دوال `init()` بالترتيب نفسه الذي تظهر فيه في الشيفرة مثل ما يلي:

```
package main
import "fmt"
func init() {
  fmt.Println("First init")
}
func init() {
  fmt.Println("Second init")
}
func init() {
  fmt.Println("Third init")
}
func init() {
  fmt.Println("Fourth init")
}
func main() {}
```

لنُشغّل البرنامج كما يلي:

```
$ go run main.go
```

سيكون الخرج:

```
First init
Second init
Third init
Fourth init
```

لاحظ أنّ تعليمات التهيئة قد نُفِّذت حسب الترتيب الذي وجدها فيه المصرِّف، وعمومًا قد لا يكون من السهل دائمًا تحديد الترتيب الذي ستُنفّذ فيه تعليمات التهيئة، ففي المثال التالي سنعرض حزمةً أكثر تعقيدًا تتضمن عدة ملفات، وكل ملف لديه دالة تهيئة خاصة به. ولتوضيح الأمر دعنا ننشئ برنامجًا يشارك متغيرًا اسمه `message` ويطبعه، والآن احذف مجلدات `creature` و `cmd` ومحتوياتهما واستبدلهما بالمجلدات وبنية الملف التالية:

```
├── cmd
│   ├── a.go
│   ├── b.go
│   └── main.go
└── message
    └── message.go
```

سنضع الآن محتويات كل ملف في a.go، لذا أضف الأسطر التالية:

```go
package main
import (
  "fmt"
  "github.com/gopherguides/message"
)
func init() {
  fmt.Println("a ->", message.Message)
}
```

يتضمن الملف دالة تهيئة واحدة تطبع قيمة `message.Message` من الحزمة `message`، والآن أضف الأسطر التالية في b.go:

```go
package main
import (
  "fmt"
  "github.com/gopherguides/message"
)
```

```
func init() {
  message.Message = "Hello"
  fmt.Println("b ->", message.Message)
}
```

لدينا في هذا الملف دالة تهيئة واحدة تُسند السلسلة "Hello" إلى المتغير `message.Message` وتطبعها، والآن أنشئ الآن ملف main.go كما يلي:

```
package main
func main() {}
```

هذا الملف لا يفعل شيئًا، ولكنه يوفر نقطة دخول للبرنامج لكي يُنفّذ، والآن أنشئ ملف message.go:

```
package message
var Message string
```

تُصرّح حزمة `message` عن المتغير المُصدّر `Message`، ولتشغيل البرنامج نفّذ الأمر التالي من مجلد `cmd`:

```
$ go run *.go
```

نحتاج إلى إخبار المصرِّف بأنه يجب تصريف جميع ملفات `.go` الموجودة في مجلد `cmd` نظرًا لوجود العديد من ملفات جو في هذا المجلد الذي يُشكّل الحزمة الرئيسية `main`، فمن خلال كتابة `*.go` يمكننا إخبار المُصرّف أنه عليه تحميل كل الملفات التي لاحقتها `.go` في مجلد `cmd`، وإذا شغّلنا الأمر `go run main.go`، فسوف يفشل البرنامج في التصريف لأنه لن يرى ملفَي a.go و b.go، وسنحصل على الخرج التالي:

```
a ->
b -> Hello
```

وفقًا لقواعد لغة جو في تهيئة الحزم، فإنه عند مصادفة عدة ملفات في الحزمة نفسها، ستكون أفضلية المعالجة وفق الترتيب الأبجدي alphabetical، لذا في أول مرة طبعنا فيها `message.Message` من a.go كانت القيمة المطبوعة هي القيمة الفارغة، وستبقى فارغةً طالما لم تُهيئ من خلال الدالة `init()` في الملف b.go، وإذا أردنا تغيير اسم ملف a.go إلى c.go، فسنحصل على نتيجة مختلفة:

```
b -> Hello
a -> Hello
```

الآن أصبح المصّرِف يرى الملف b.go أولًا، وبالتالي أصبحت قيمة `message.Message` مُهيّئة بالقيمة "Hello" وذلك بعد تنفيذ الدالة `init()` في ملف c.go.

لاحظ أنّ هذا السلوك قد ينتج أخطاءً و مشاكل لاحقة لأنه من الشائع تغيير أسماء الملفات أثناء تطوير أيّ مشروع، وهذا يؤثر على ترتيب تنفيذ دالة التهيئة كما أوضحنا.

يفضَّل تقديم الملفات المتعددة الموجودة ضمن الحزمة نفسها ضمن ملف واحد وفق ترتيب معجمي lexical order إلى المصرّف للتحكم بنهج التهيئة المتعدد، وبالتالي ضمان أنّ كل دوال التهيئة تُحمّل من أجل التصريح عنها في ملف واحد، وبالتالي منع تغيير ترتيب تنفيذها حتى لو تغيرت الأسماء.

يجب أن تحاول أيضًا تجنب إدارة الحالة managing state في الحزمة الخاصة بك باستخدام المتغيرات العامة، أي المتغيرات التي يمكن الوصول إليها من أيّ مكان في الحزمة، ففي البرنامج السابق مثلًا كان المتغير `message.Message` مُتاحًا في أيّ مكان ضمن الحزمة مع الحفاظ على حالة البرنامج، وبسبب هذه الإمكانية كانت دوال التهيئة قادرةً على الوصول إلى هذا المتغير وتعديل قيمته، وبالتالي أصبح من الصعب التنبؤ بسلوك هذا البرنامج، ولتجنب ذلك يُفضَّل إبقاء المتغيرات ضمن مساحات يمكنك التحكم بها مع إمكانية وصول ضئيلة لها قدر الإمكان حسب حاجة البرنامج.

قد يؤدي وجود عدة تعليمات تهيئة ضمن البرنامج إلى ظهور تأثيرات غير مرغوب فيها ويجعل من الصعب قراءة برنامجك أو التنبؤ به، كما يضمن لك تجنب استخدام تعليمات التهيئة المتعددة أو الاحتفاظ بها جميعًا في ملف واحد بعدم تغير سلوك برنامجك عند نقل الملفات أو تغيير الأسماء.

19.5 استخدام دالة التهيئة لتحقيق مفهوم التأثير الجانبي

غالبًا ما تستورَد بعض الحزم في لغة جو بغية الاستفادة من تأثيرها الجانبي فقط وليس من أجل أيّ مكوِّن آخر من مكوناتها، وغالبًا ما يكون ذلك عندما تتضمن هذه الحزمة دالة تهيئة بداخلها، وبالتالي تُنفّذ قبل أيّ شيفرة أخرى بمجرد استيراد الحزمة، وبالتالي إمكانية التلاعب بالحالة التي يبدأ بها البرنامج، وتسمى هذه العملية باستيراد التأثير الجانبي.

إحدى حالات الاستخدام الشائعة لمفهوم استيراد التأثير الجانبي هي الحصول على معلومات أولية تُفيد في تحديد أي جزء من شيفرة ما يجب أن يُنفّذ، ففي حزمة `image` مثلًا تحتاج الدالة `image.Decode` إلى معرفة تنسيق الصورة التي تحاول فك تشفيرها (gif، png، jpg، ...إلخ) قبل أن تتمكن من تنفيذها، وهكذا أمر يمكن تحقيقه من خلال مفهوم التأثير الجانبي، فلنقل أنك تحاول استخدام `image.Decode` مع ملف بتنسيق png كما يلي:

```
. . .
func decode(reader io.Reader) image.Rectangle {
  m, _, err := image.Decode(reader)
  if err != nil {
        log.Fatal(err)
```

```
    }
    return m.Bounds()
}
. . .
```

هذا البرنامج سليم من ناحية التصريف، لكن إذا حاولت فك تشفير صورة بتنسيق png، فستتلقى خطأً، ولحل هذه المشكلة سنحتاج إلى تسجيل تنسيق الصورة في الدالة `image.Decode`، ولحسن الحظ تتضمن الحزمة `image/png` تعليمة التهيئة التالية:

```
func init() {
  image.RegisterFormat("png", pngHeader, Decode, DecodeConfig)
}
```

وبالتالي إذا استوردنا `image/png` إلى الشيفرة السابقة، فسوف تُنفَّذ دالة التهيئة `image.RegisterFormat()` مباشرةً وقبل أيّ شيء آخر.

```
. . .
import _ "image/png"
. . .
func decode(reader io.Reader) image.Rectangle {
  m, _, err := image.Decode(reader)
  if err != nil {
        log.Fatal(err)
  }
  return m.Bounds()
}
```

سيؤدي هذا إلى ضبط حالة البرنامج وتسجيل أننا نحتاج إلى إصدار png من `image.Decode()`، وسيحدث هذا التسجيل بوصفه أثرًا جانبيًا لاستيراد الحزمة `image/png`.

ربما لاحظت وجود الشرطة السفلية _ قبل `image/png` وهذا أمر مهم لأنّ جو لا تسمح لك باستيراد حزمة غير مستخدَمة في البرنامج، وبالتالي فإنّ وجود الشرطة السفلية سيتجاهل أيّ شيء في الحزمة المستورَدة باستثناء تعليمة التهيئة -أي التأثير الجانبي-، وهذا يعني أنه حتى إذا لم نستورِد الحزمة `image/png`، فيمكننا استيراد تأثيرها الجانبي.

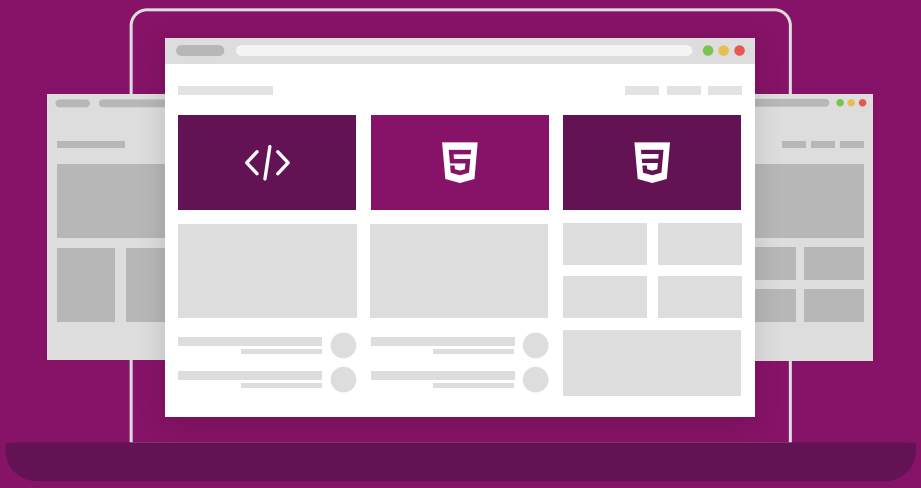
من المهم أن تعرف متى تحتاج إلى استيراد حزمة للحصول على تأثيرها الجانبي، فبدون التسجيل الصحيح، من المحتمل أن تتمكن جو من تصريف برنامجك ولكن لن يعمل كما تتوقع، وعادةً ما تتضمن مراجع (أو توثيقات documentation) المكتبة القياسية إشارةً إلى الحاجة إلى هكذا نوع من التأثيرات مع حالات معينة،

وبالتالي إذا كتبت حزمةً تتطلب استيرادًا للتأثيرات الجانبية، فيجب عليك أيضًا التأكد من توثيق دالة التهيئة التي تستخدِمها حتى يتمكن المستخدِمون الذين يستورِدون الحزمة الخاصة بك من استخدامها بطريقة صحيحة.

19.6 الخاتمة

تعلّمنا في هذا الفصل من كتاب تعلم البرمجة بلغة جو طريقة استخدام دالة التهيئة `init()` وفهمنا كيف أنها تُنفَّذ قبل أيّ شيء آخر في البرنامج وأنها تستطيع أداء بعض المهام وضبط حالة البرنامج الأولية، كما تحدّثنا أيضًا عن تعليمات التهيئة المتعددة والموجودة ضمن ملفات متعددة وكيف أنّ ترتيب تنفيذها يعتمد على على أمور محددة مثل الترتيب الأبجدي أو المعجمي لملفات الحزمة.

دورة تطوير واجهات المستخدم



ابدأ مسارك المهني كمطور واجهات المواقع والمتاجر الإلكترونية
فور انتهائك من الدورة

التحق بالدورة الآن

20. تخصيص الملفات التنفيذية بوسوم البناء

إنَّ وسم البناء Build tag أو قيد البناء Build constraint هو مُعرّف يُضاف إلى التعليمات البرمجية لتحديد متى يجب تضمين ملف ما في حزمة أثناء عملية البناء `build`، ويتيح لك إمكانية بناء إصدارات مُختلفة لتطبيقك من نفس التعليمات البرمجية المصدرية والتبديل بينها بطريقة سريعة ومنظمة.

ويستخدم العديد من المطورين وسوم البناء لتحسين سير العمل Workflow عند بناء تطبيقات متوافقة مع جميع أنظمة تشغيل الأساسية Cross-platform، مثل البرامج التي تتطلب تغييرات في التعليمات البرمجية لمراعاة الفروقات بين أنظمة التشغيل المختلفة. تُستخدم وسوم البناء أيضًا من أجل اختبار التكامل Integration testing، مما يسمح لك بالتبديل بسرعة بين الشيفرة المتكاملة والشيفرة باستخدام خادم زائف Mock server أو شيفرة اختبارية بديلة Stub، وبين المستويات المختلفة لمجموعات الميزات التي يتضمنها تطبيقك.

لنأخذ مثلًا، مشكلة اختلاف مجموعات الميزات التي تُمنح للعملاء، فعند كتابة بعض التطبيقات، قد ترغب في التحكم بالمميزات التي يجب تضمينها في الثنائي binary، مثل التطبيق الذي يوفّر مستويات مجانية واحترافية Pro ومتقدمة Enterprise. كلما رفع العميل من مستوى اشتراكه في هذه التطبيقات، توفّرت له المزيد من الميزات وأصبحت غير مقفلة.

يمكنك حل هذه المشكلة من خلال الاحتفاظ بمشاريع منفصلة ومحاولة إبقائها متزامنةً مع بعضها بعضًا من خلال استخدام تعليمات الاستيراد import، وعلى الرغم من أن هذا النهج سيعمل، لكنه سيصبح مملًا بمرور الوقت وعرضةً للخطأ، وقد يكون النهج البديل هو استخدام وسوم البناء.

ستستخدم في هذا الفصل وسوم البناء في لغة جو، لإنشاء ملفات تنفيذية مختلفة تُقدم مجموعات ميزات مجانية واحترافية ومتقدمة لتطبيقك. سيكون لكل من هذه الملفات التنفيذية مجموعةٌ مختلفةٌ من الميزات المتاحة، إضافةً للإصدار المجاني، الذي هو الخيار الافتراضي.

ملاحظات:

- التوافق مع أنظمة التشغيل الأساسية Cross-platform: هو مصطلح يستخدم في علم الحوسبة يشير إلى برامج الحاسوب أو أنظمة التشغيل أو لغات الحاسوب أو لغات البرمجة وتطبيقاتها التي يمكنها العمل على عدة منصات حاسوبية. وهناك نوعان رئيسيان من البرمجيات المتوافقة مع أنظمة التشغيل الأساسية، إذ يستلزم الأول بناءه لكل منصة يمكنه العمل عليها، والثاني يمكنه العمل مباشرةً على أي منصة تدعمه.
- اختبار التكامل Integration testing: يمثّل مرحلة اختبار البرامج التي تتكامل فيها الوحدات البرمجية وتُختبر مثل وحدة واحدة متكاملة. يُجرى اختبار التكامل لتقييم مدى امتثال نظام أو مكون برمجي لمتطلبات وظيفية محددة، وغالبًا ما تكون هذه المتطلبات مدونة في توثيق الخاصيات والمتطلبات.
- الخادم الزائف Mock server: هو إطار عمل يهدف إلى تبسيط اختبار التكامل. تعتمد هذه الأُطر على مفهوم الكائنات الزائفة، وهي كائنات محاكاة تحاكي سلوك الكائنات الحقيقية بطرق خاضعة للرقابة، وتكون غالبًا بمثابة جزء من عملية اختبار البرنامج. يُنشئ المبرمج عادةً كائنًا زائفًا لاختبار سلوك بعض الأشياء الأخرى، بنفس الطريقة التي يستخدم بها مصمم السيارة دمية اختبار التصادم لمحاكاة السلوك الديناميكي للإنسان في اصطدام السيارة. هذه التقنية قابلة للتطبيق أيضًا في البرمجة العامة.
- الشيفرة الاختبارية البديلة Stub: هي برنامج صغير يُستبدل ببرنامج أطول، ربما يُحمّل لاحقًا، أو يكون موجودًا عن بُعد في مكان ما، إذ يكون بديلًا مؤقتًا للشيفرة التي لم تُطوّر بعد، وهذه الشيفرة مفيدة جدًا في نقل البيانات والحوسبة الموزعة وكذلك تطوير البرمجيات واختبارها عمومًا.

20.1 المتطلبات الأساسية

ستحتاج لامتلاك مساحة عمل خاصة مُهيئة وجاهزة في لغة جو، فإذا لم يكن لديك واحدة، فارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبّت لغة جو Go وقم بإعداد بيئة تطوير محلية بحسب نظام تشغيلك، ويُفضّل أن تكون قد اطلعت أيضًا على فقرة التعرّف على GOPATH وفقرة استيراد الحزم في لغة جو Go قبل المتابعة في قراءة الفقرات التالية.

20.2 بناء النسخة المجانية

سنبدأ ببناء الإصدار المجاني من التطبيق، إذ سيكون هو الإصدار الافتراضي عند تنفيذ الأمر `go build` دون أي وسوم بناء. سنستخدم لاحقًا وسوم البنية لإضافة أجزاء أخرى إلى برنامجنا.

أنشئ مجلدًا باسم التطبيق الخاص بك في مجلد `src`، وسنستخدم هنا الاسم `app`:

```
$ mkdir app
```

انتقل إلى المجلد app الذي أنشأته:

```
$ cd app
```

أنشئ الآن ملف main.go داخل مجلد المشروع. وقد استخدمنا هنا محرر النصوص نانو nano لفتح وإنشاء الملف:

```
$ nano main.go
```

سنعرّف الآن الإصدار المجاني من التطبيق. انسخ المحتويات التالية إلى ملف main.go:

```
package main
import "fmt"
var features = []string{
 "Free Feature #1",
 "Free Feature #2",
}
func main() {
 for _, f := range features {
  fmt.Println(">", f)
 }
}
```

أنشأنا برنامجًا يُصرّح عن شريحة Slice باسم features، تحتوي على سلسلتين نصيتين strings تمثلان ميزات إصدار تطبيقنا المجاني. تستخدم الدالة main() حلقة for لتنتقل عبر عناصر شريحة الميزات من أجل طباعة جميع الميزات المتاحة على الشاشة.

احفظ الملف واخرج منه، وبعد حفظ الملف لن نضطر إلى تحريره مرةً أخرى خلال هذا الفصل، إذ سنستخدم وسوم البناء لتغيير ميزات الثنائيات التي سنبنيها منها.

اِبنِ وشغّل البرنامج:

```
$ go build
$ ./app
```

ستحصل على الخرج التالي:

```
\> Free Feature #1
\> Free Feature #2
```

طبع البرنامج ميزتين مجانيتين تكملان ميزات الإصدار المجاني من تطبيقنا.

أكملنا الآن الإصدار المجاني المُتمثّل بتطبيق يحتوي على مجموعة ميزات أساسية جـدًا. سـنبني بعـد ذلـك شيفرة تمكننا من إضافة مزيدٍ من الميزات إلى التطبيق عند البناء.

20.3 إضافة ميزات احترافية باستخدام go build

تجنّبنا حتى الآن إجراء تغييرات على الملف main.go، وذلك لمحاكاة بيئـة إنتـاج عامـة ينبغي فيهـا إضـافة الشيفرة دون تغيير الشيفرة الرئيسية أو كسرها.

ونظرًا لإمكانية تعديل ملف main.go، سنحتاج إلى اسـتخدام آليـة أخـرى لإدخـال المزيـد من المـيزات إلى شريحة `features` باستخدام وسوم البناء.

سننشئ ملفًـا جديـدًا باسـم pro.go، والـذي سيسـتخدم الدالـة `init()` لإضـافة المزيـد من المـيزات إلى شريحة `features`:

```
$ nano pro.go
```

أضف المحتويات التالية إلى الملف بعد فتحه:

```
package main
func init() {
 features = append(features,
  "Pro Feature #1",
  "Pro Feature #2",
 )
}
```

اسـتخدمنا الدالـة `init()` لتشـغيل الشـيفرة قبـل الدالـة `main()` في التطـبيق، ثم اسـتخدمنا الدالـة `append()` لإضافة ميزات احترافية إلى شريحة `features`.

احفظ الملف واخرج منه، ثم صرّف التطبيق وشغّله باستخدام الأمر التالي:

```
$ go build
```

نظرًا لوجود ملفين الآن في المجلد الحالي، هما pro.go و main.go، سينشئ الأمر `go build` ملفًا ثنائيًا من كليهما:

```
$ ./app
```

ستحصل على الخرج التالي:

```
\> Free Feature #1
\> Free Feature #2
\> Pro Feature #1
\> Pro Feature #2
```

يتضمن التطبيق الآن كلًا من الميزات الاحترافية والمجانية، وهذا غير مرغوب بالوضع الحـالي للتطـبيق؛ فلا يوجد تمييز بين الإصدارات، إذ يتضمن الإصدار المجاني الميزات التي من المفترض أن تكون متـوفرة فقـط في الإصدار الاحترافي. يمكنك حل المشـكلة عن طريـق إضـافة المزيـد من التعليمـات البرمجيـة لإدارة المسـتويات المختلفة للتطبيق، أو استخدام وسوم البناء لإخبار أدوات جو عن الملفـات الـتي تكـون بامتـداد go.،الـتي يجب بناؤها وتلك التي يجب تجاهلها. سنضيف في الخطوة التالية وسوم البناء.

20.4 إضافة وسوم البناء

يمكنك الآن استخدام وسوم البناء لتمييز الإصدار الاحترافي عن المجاني. يكون شكل الوسم كما يلي:

```
// +build tag_name
```

من خلال وضع هذا السطر البرمجي في بداية الحزمة الخاصة بـك (في أول سـطر) وتبـديل `tag_name` إلى اسم وسم البناء الذي تريده، ستُوسّم هذه الحزمة لتصبح شيفرةً يمكن تضمينها اختياريًّا في الثنائي النهائي. دعنا نرى هذا عمليًا عن طريق إضافة وسم البناء إلى ملف pro.go لإخبـار الأمـر `go build` بتجاهلهـا مـا لم يُحـدّد الوسم. افتح الملف في محرر النصوص الخاص بك:

```
$ nano pro.go
```

أضِف ما يلي:

```
// +build pro
package main
func init() {
 features = append(features,
  "Pro Feature #1",
  "Pro Feature #2",
 )
}
```

أضفنـا في بداية الملف pro.go السـطر `// +build pro` متبوعًا بسـطر جديد فـارغ؛ وهذا السـطر الجديد ضروري، وبدونه سيفسّر جو السطر السابق على أنه تعليق، ويجب أن تكون تصريحات وسوم البناء أيضًا في أعلى الملف ذي الامتداد go. دومًا، لا تضع أي شيء، ولا حتى التعليقات قبلها.

يُخبِر التصريح `+build` الأمر `go build` أن هذا ليس تعليقًا، بل هو وسم بناء. والجزء الثاني هو الوسم `pro`، فبإضافة هذا الوسم في الجزء العلوي من ملف pro.go، سيُضمّن الأمر `go build` ملف pro.go في حال وجود الوسم `pro` فقط.

صرّف التطبيق الآن وشغّله:

```
$ go build
$ ./app
```

ستحصل على الخرج التالي:

```
\> Free Feature #1
\> Free Feature #2
```

بما أن ملف pro.go يتطلب وجود الوسم `pro`، سيجري تجاهل الملف وسيُصرّف التطبيق دونه.

عند استخدام الأمر `go build`، يمكننا استخدام الراية `-tags` لتضمين شيفرة محددة لكي تُصرّف مع التطبيق عن طريق إضافة وسم الشيفرة مثل وسيط. سنجرّب ذلك مع الوسم `pro`:

```
$ go build -tags pro
```

ستحصل على الخرج التالي:

```
\> Free Feature #1
\> Free Feature #2
\> Pro Feature #1
\> Pro Feature #2
```

سنحصل الآن على الميزات الاحترافية فقط إذا أضفنا الوسم `pro`.

هذا جيد إذا كان هناك إصدارين فقط، ولكن الأمور تصبح معقدةً عند وجود مزيدٍ من الإصدارات وإضافة مزيدٍ من الوسوم. سنستخدم في الخطوة التالية وسوم بناء متعددة مع منطق بولياني Boolean logic، لإضافة إصدار مُتقدم.

20.5 استخدام المنطق البولياني مع وسوم البناء

عندما تكون هناك وسوم بناء متعددة في حزمة جو، تتفاعل هذه الوسوم مع بعضها بعضًا باستخدام المنطق البولياني. ومن أجل توضيح ذلك، سنضيف المستوى مُتقدم Enterprise إلى تطبيقنا باستخدام الوسم `pro` والوسم `enterprise`.

سنحتاج من أجل بناء ثنائي للإصدار المتقدم إلى تضمين كل من الميزات الافتراضية والمـيزات الاحترافيـة، ومجموعة جديدة من الميزات الخاصة بالإصدار المتقدم. افتح محررًا وأنشئ ملفًا جديدًا enterprise.go، لنضـع فيه الميزات الجديدة:

```
$ nano enterprise.go
```

ستبدو محتويات enterprise.go متطابقة تقريبًا مع pro.go ولكنها ستحتوي على مـيزات جديـدة. أضِـف الأسطر التالية إلى الملف enterprise.go:

```
package main
func init() {
 features = append(features,
  "Enterprise Feature #1",
  "Enterprise Feature #2",
 )
}
```

لا يحتوي ملف enterprise.go حاليًا على أي وسوم بناء، وكما تعلّمت عندما أضفت pro.go، فهذا يعـني أن هذه الميزات ستضاف إلى الإصدار المجاني عند تنفيذ go.build. بالنسبة إلى pro.go،

أضفت `\\ +build pro` متبوعًا بسطر فارغ في أعلى الملف لإخبار الأمر `go build` أنه يجب تضمينه فقط عند استخدام `-tags pro`. تحتاج في هذه الحالة فقـط إلى وسـم بنـاء واحـد لتحقيـق الهـدف، لكن عنـد إضافة الميزات الجديدة المتقدمة، إذ يجب أن يكون لديك أولًا ميزات احترافية Pro.

سنجعل الآن ملف enterprise.go يدعم وسم البناء pro. افتح الملف في محرر النصوص الخاص بك:

```
$ nano enterprise.go
```

أضف بعد ذلك وسم البناء قبل سطر التصريح `package main` وتأكد من إضافة سطر فارغ كما تحدثنا:

```
// +build pro
package main
func init() {
 features = append(features,
  "Enterprise Feature #1",
  "Enterprise Feature #2",
 )
}
```

احفظ الملف وأغلقه، ثم صرّف التطبيق دون إضافة أي وسوم (الإصدار المجاني):

```
$ go build
$ ./app
```

ستحصل على الخرج التالي:

```
\> Free Feature #1
\> Free Feature #2
```

لاحظ أن الميزات المتقدمة لا تظهر في الإصدار المجاني. دعنا الآن نضيف وسم الإصدار المحترف `pro` ونبني التطبيق ونشغّله مرةً أخرى:

```
$ go build -tags pro
$ ./app
```

ستحصل على الخرج التالي:

```
\> Free Feature #1
\> Free Feature #2
\> Enterprise Feature #1
\> Enterprise Feature #2
\> Pro Feature #1
\> Pro Feature #2
```

ليس هذا فعلًا ما نحتاجه حتى الآن، لأن الميزات المتقدمة تظهر عندما نبني إصدار احترافي، وسنحتاج إلى استخدام وسم بناء آخر لحل هذه المشكلة. بالنسبة للوسم `enterprise` فهو على عكس الوسم `pro`، إذ نحتاج هنا إلى التأكد من توفُّر الميزات الاحترافية `pro` والمتقدمة `enterprise` في نفس الوقت.

يراعي نظام البناء في جو هذا الموقف ويسمح باستخدام بعض المنطق البولياني في نظام وسوم البناء.

لنفتح enterprise.go مرةً أخرى:

```
$ nano enterprise.go
```

سنضيف الآن وسمًا إلى هذا الملف باسم `enterprise` كما فعلنا سابقًا عند إضافة الوسم `pro`:

```
// +build pro enterprise
package main
```

```
func init() {
 features = append(features,
  "Enterprise Feature #1",
  "Enterprise Feature #2",
 )
}
```

احفظ الملف واخرج، ثم صرّف التطبيق مع إضافة الوسم `enterprise`:

```
$ go build -tags enterprise
$ ./app
```

سيكون الخرج على النحو التالي:

```
\> Free Feature #1
\> Free Feature #2
\> Enterprise Feature #1
\> Enterprise Feature #2
```

لاحظ أننا خسرنا ميزات الإصدار الاحترافي، وسبب ذلك هو أنه عندما نضع عدة وسوم بناء ضمن نفس السطر في ملف بالامتداد go.، سيفسر `go build` أن العلاقة بينهما هي OR المنطقية. إذًا، بإضافة السطر `\\ +build pro enterprise` سوف يُبنى الملف enterprise.go في حال وجود إحدى الوسمين `pro` أو `enterprise`.

نحن الآن بحاجة إلى كتابة وسوم البناء بطريقة صحيحة مع استخدام المنطق AND، ولفعل ذلك سنكتب كلًا من الوسمين في سطر منفصل، وبهذا الشكل سيفسّر `go build` على أن العلاقة بينهما AND.

افتح الملف enterprise.go مرةً أخرى وافصل وسوم البناء في أسطر منفصلة على هذا النحو:

```
// +build pro
// +build enterprise
package main
func init() {
 features = append(features,
  "Enterprise Feature #1",
  "Enterprise Feature #2",
 )
}
```

الآن، صرّف تطبيقك مع الوسم `enterprise`:

```
$ go build -tags enterprise
$ ./app
```

ستحصل على الخرج التالي:

```
\> Free Feature #1
\> Free Feature #2
```

هذا غير كافي، لأن منطق AND يتطلب تحقق العنصرين لذا نحتاج وسمي البناء `pro` و `enterprise`:

```
$ go build -tags "enterprise pro"
$ ./app
```

ستحصل على الخرج التالي:

```
\> Free Feature #1
\> Free Feature #2
\> Enterprise Feature #1
\> Enterprise Feature #2
\> Pro Feature #1
\> Pro Feature #2
```

يمكنك الآن بناء التطبيق من نفس شجرة المصدر بعدة طرق مختلفة وفقًا للميزات التي تريد توفيرها.

وقد استخدمنا في هذا المثـال وسم بناء جديد باستخـدام وسم `// +build` للـدلالة على منطق AND، ولكن هناك طرق بديلة لتمثيل المنطق البولياني باستخدام وسوم البناء.

يحتوي الجدول التالي أمثلة على تنسيقات نحوية أخرى لوسوم البناء، جنبًا إلى جنب مع مكافئها المنطقي:

قاعدة وسم البناء	عينة عن الوسم	التعليمة المنطقية
عناصر مفصولة بفراغ	`\\ +build pro enterprise`	محترف أو متقدم `pro` OR `enterprise`
عناصر مفصولة بفاصلة	`\\ +build pro,enterprise`	محترف ومتقدم `pro` AND `enterprise`
عناصر مفصولة بعلامة تعجب	`\\ +build !pro`	ليس محترف NOT `pro`

20.6 خاتمة

تعرّفت في هذا الفصل على كيفية استخدام وسوم البناء Build tags للتحكم في الملفات التي ستُصـرّف إلى ملفات ثنائية في جو؛ إذ صرّحنا أولًا عن وسم البناء، ثم استخدمناها في `go build` ثم دمجنـا عـدة وسـوم باستخدام المنطق البولياني. أخيرًا بنينـا برنامجًـا يتضـمن مجموعـات مـيزات مختلفـة كـل منهـا يمثـل إصـدارًا (مجاني، احترافي، متقدم)، والذي أظهر قوة وسوم البناء التي تمثّل أداةً قوية جدًا بالتحكم بإصدارات المشروع.

دورة إدارة تطوير المنتجات

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

21. تعرف على المؤشرات Pointers

المؤشّر هو عنوان يشير إلى موقع في الذاكرة، وتُستخدم المؤشرات عادةً للسماح للدوالّ أو هياكـل البيانـات بالحصول على معلومات عن الـذاكرة وتعـديلها دون الحاجـة إلى نسـخ الـذاكرة المشـار إليهـا، والمؤشّـرات قابلـة للاستخدام سواءٌ مع الأنواع الأوليّة (المُضمّنة) أو الأنواع التي يعرّفها المستخدم.

تتضمن البرامج المكتوبة بلغة جو عادةً دوالًا وتوابعًا Methods نمرر لها بيانات مثل وسطاء، ونحتـاج أحيانًـا إلى إنشاء نسخة محلية من تلك البيانات بحيث تظل النسخة الأصلية من البيانات دون تغيير. مثلًا، لو كان لديك برنامج يمثّل مصرفًا bank، وتريد أن يُظهر هذا البرنامج التغيرات التي تطرأ على رصـيد المسـتخدم تبعًـا للخطـة التوفيرية التي يختارها، فهنا قد تحتاج إلى بناء دالة تنجز هـذا الأمـر من خلال تمريـر الرصـيد الحـالي للمسـتخدم إضافةً إلى الخطة التي يريدها. لا نريد هنا تغيير الرصيد الأساسي وإنما نريد فقط إظهار التعديل الذي سيطرأ على الرصيد، وبالتالي يجب أن نأخذ نسخةً من رصيد المستخدم مثل وسيط للدالة ونعدّل على هذه النسـخة. تسـمى هذه النسخة نسخةً محلية، وندعو عملية التمرير هذه "التمرير بالقيمة passing by value" لأننا لانرسل المتغير نفسه وإنما قيمته فقط.

هناك حالات أخرى قد تحتاج فيها إلى تعديل البيانات الأصلية؛ بمعنى آخر قد نحتاج إلى تغيير قيمة المتغير الأصلي مباشرةً من خلال الدالة، فمثلًا، عندما يودع المستخدم رصيدًا إضافيًّا في حسابه، فهنـا تحتـاج إلى جعـل الدالة قادرةً على تعديل قيمة الرصيد الأصلي وليس نسخةً منه (نحن نريد إضافة مال إلى رصيده السابق). ليس ضروريًا هنا تمرير البيانات الفعلية إلى الدالة، إذ يمكنك ببساطة إخبار الدالة بالمكان الذي توجـد بـه البيانـات في الذاكرة من خلال "مؤشر" يحمل عنوان البيانات الموجودة في الذاكرة. لا يحمل المؤشر القيمة، وإنما فقط عنـوان أو مكان وجود القيمة، وتتمكن الدالة من خلال هذا المؤشر من التعديل على البيانـات الأصـلية مباشـرةً. يسـمى هذا "التمرير بالمرجع passing by reference"، لأن قيمة المتغير لا تُمرّر إلى الدالة، بل إلى موقعها فقط.

سننشئ في هذا الفصل المؤشرات ونستخدمها لمشاركة الوصول إلى الذاكرة المُخصصة لمتغير ما.

21.1 تعريف واستخدام المؤشرات

يوجد عنصرا صيغة مختلفان يختصان باستخدام مؤشّر لمتغيرٍ variable ما، وهما: معامل العنونة Address-of operator وهو & الذي يعيد عنوان المتغيّر الذي يوضع أمامه في الذاكرة، ومعامل التحصيل Dereference، وهو * الذي يعيد قيمة المتغير الموجود في العنوان المحدّد بواسطة عامله.

> يُستخدم رمز النجمة * أيضًا للتصريح عن مؤشّر لمجرد التوضيح بأنه مؤشر، ولا ينبغي أن تخلط بينه وبين عامل التحصيل المُستخدم للحصول على القيمة الموجودة في عنوان محدّد، فهما شيئان مختلفان مُمثّلان بنفس الرمز.

مثال:

```
var myPointer *int32 = &someint
```

صرّحنا هنا عن متغير يُسمّى `myPointer` يُمثّل مؤشرًا لمتغير من نوع العدد الصحيح `int32`، وهيأنا المؤشر بعنوان `someint`، فالمؤشر هنا يحمل عنوان المتغير `int32` وليس قيمته.

دعنا نلقي الآن نظرةً على مؤشر لسلسلة، إذ تُصرّح الشيفرة التالية عن متغير يُمثّل سلسلة ومتغير آخر يُمثّل مؤشرًا على تلك السلسلة:

```
package main
import "fmt"
func main() {
  var creature string = "shark"
  var pointer *string = &creature
  fmt.Println("creature =", creature)
  fmt.Println("pointer =", pointer)
}
```

شغّل البرنامج بالأمر التالي:

```
$ go run main.go
```

سيطبع البرنامج عند تشغيله قيمة المتغير إضافةً إلى عنوان تخزين المتغير (عنوان المؤشر). عنوان الذاكرة هو سلسلة أرقام مكتوبة بنظام العد السداسي عشري لأسباب عتادية وبرمجية لا تهمنا الآن، وطبعنا القيم هنا للتوضيح فقط. ستلاحظ تغيُّر العنوان المطبوع في كل مرة تُشغّل فيها البرنامج، لأنه يُهيّأ من جديد وتأخذ فيه المتغيرات أماكن غير محددة في الذاكرة؛ فكل برنامج ينشئ مساحته الخاصة من الذاكرة عند تشغيله. إذًا، سيكون خرج الشيفرة السابقة مختلفًا لديك عند تشغيله:

```
creature = shark
pointer = 0xc0000721e0
```

عرّفنا المتغير الأول `creature` من النوع `string` وهيّأناه بالقيمة `shark`. أنشأنا أيضًا متغيرًا يُسمّى `pointer` يُمثّل مؤشرًا على عنوان متغير سلسلة نصية، أي يحمل عنوان متغير نوعه `string`، وهيّأناه بعنوان السلسلة النصية المُمثّلة بالمتغير `creature` وذلك من خلال وضع المعامل `&` قبل اسمه.

إذًا، سيحمل `pointer` عنوان الذاكرة التي يوجد بها `creature` وليس قيمته. هذا هو السبب وراء الحصول على القيمة `0xc0000721e0` عندما طبعنا قيمة المؤشر، وهو عنوان مكان تخزين متغير `creature` حاليًا في ذاكرة الحاسب.

يمكنك الوصول إلى قيمة المتغير مباشرةً من خلال نفس المؤشر باستخدام معامل التحصيل `*` كما يلي:

```
package main
import "fmt"
func main() {
  var creature string = "shark"
  var pointer *string = &creature
  fmt.Println("creature =", creature)
  fmt.Println("pointer =", pointer)
  fmt.Println("*pointer =", *pointer)
}
```

ويكون الخرج على النحو التالي:

```
creature = shark
pointer = 0xc000010200
*pointer = shark
```

يُمثّل السطر الإضافي المطبوع ما تحدّثنا عنه (الوصول لقيمة المتغير من خلال المؤشر). نسمي ذلك "التحصيل" إشارةً إلى الوصول لقيمة المتغير من خلال عنوانه. يمكننا استخدام هذه الخاصية أيضًا في تعديل قيمة المتغير المُشار إليه:

```
package main
import "fmt"
func main() {
  var creature string = "shark"
  var pointer *string = &creature
```

```
    fmt.Println("creature =", creature)
    fmt.Println("pointer =", pointer)
    fmt.Println("*pointer =", *pointer)
    *pointer = "jellyfish"
    fmt.Println("*pointer =", *pointer)
}
```

وسيكون الخرج على النحو التالي:

```
creature = shark
pointer = 0xc000094040
*pointer = shark
*pointer = jellyfish
```

لاحظ أننا في السطر `*pointer = "jellyfish"` وضعنا معامل التحصيل * قبل المؤشر للإشارة إلى أننا نريد تعديل القيمة التي يُشير إلى عنوانها المؤشر. أسندنا القيمة "jellyfish" إلى موقع الذاكرة التي يُشير لها `pointer`، وهذا يُكافئ تعديل قيمة المتغير `creature`. لاحظ أنه عند طباعة القيمة التي يُشير لها المؤشر سنحصل على القيمة الجديدة.

كما ذكرنا؛ فهذا يُكافئ تعديل قيمة المتغير `creature`، وبالتالي لو حاولنا طباعة قيمة المتغير `creature` سنحصل على القيمة "jellyfish" لأننا نُعدّل على الموقع الذاكري نفسه. سنضيف الآن سطرًا يطبع قيمة المتغير `creature` إلى الشيفرة السابقة:

```
package main
import "fmt"
func main() {
    var creature string = "shark"
    var pointer *string = &creature
    fmt.Println("creature =", creature)
    fmt.Println("pointer =", pointer)
    fmt.Println("*pointer =", *pointer)
    *pointer = "jellyfish"
    fmt.Println("*pointer =", *pointer)
    fmt.Println("creature =", creature)
}
```

سيكون الخرج كما يلي:

```
creature = shark
pointer = 0xc000010200
*pointer = shark
*pointer = jellyfish
creature = jellyfish
```

يهدف كل ما تعلمته حتى الآن إلى توضيح فكرة المؤشرات في لغة جو وليس حالات الاستخدام الشائعة لها، فهي تُستخدم غالبًا عند تعريف وسطاء الدوال والقيم المُعادة منها أو عند تعريف التوابع مع أنواع مخصصة. دعنا الآن نلقي نظرةً على كيفية استخدام المؤشرات مع الدوال لمشاركة الوصول إلى المتغيرات.

ضع في الحسبان أننا نطبع قيمة المؤشر لتوضيح أنه مؤشر، فلن تستخدم عمليًا قيمة المؤشر إلا للإشارة إلى القيمة الأساسية لاستردادها أو تعديلها.

21.2 مستقبلات مؤشرات الدوال

عند كتابة دالة، يمكنك تعريف بعض الوسطاء لكي تُمررهم لها إما بالقيمة أو بالمرجع؛ فعندما تُمرر وسيطًا ما بالقيمة، فهذا يعني أنك تُرسل نسخةً مُستقلة من قيمة هذا الوسيط إلى الدالة، وبالتالي فإن أي تغيير يحدث لهذه النسخة لن يؤثر على النسخة الأساسية من البيانات، لأن كل التعديلات ستجري على نُسخة من البيانات؛ أما عندما تُمرر وسيطًا بالمرجع، فهذا يعني أنك تُرسل مؤشرًا يحمل عنوان ذلك الوسيط -أي مكان تواجد البيانات في الذاكرة- إلى الدالة، وبالتالي أصبح لديك القدرة على الوصول إلى البيانات الأصلية من داخل الدالة والتعديل عليها مباشرةً.

يمكنك الاطلاع على الفصل التالي إذا أردت معرفة المزيد عن تعريف الدوال وطرق استدعائها في لغة جو.

يمكنك طبعًا تمرير أي وسيط بالطريقة التي تُريدها (بالقيمة أو بالمرجع)، فهذا يعتمد على ما تحتاجه؛ فإذا كنت تريد أن تُعدّل الدالة على البيانات الأصلية نُمرر الوسيط بالمرجع وإلا بالقيمة.

لمعرفة الفرق بدقة، دعنا أولًا نلقي نظرة على دالة تمرّر وسيطًا بالقيمة:

```
package main
import "fmt"
type Creature struct {
  Species string
}
func main() {
  var creature Creature = Creature{Species: "shark"}
  fmt.Printf("1) %+v\n", creature)
  changeCreature(creature)
```

```
    fmt.Printf("3) %+v\n", creature)
}
func changeCreature(creature Creature) {
    creature.Species = "jellyfish"
    fmt.Printf("2) %+v\n", creature)
}
```

ويكون الخرج على النحو التالي:

```
1) {Species:shark}
2) {Species:jellyfish}
3) {Species:shark}
```

أنشأنا بدايةً نوع بيانات مخصص أسميناه `Creature`، يحتوي على حقل واحد يُسمى `Species` من نوع سلسلة نصية `string`، وأنشأنا داخل الدالة الرئيسية `main` متغير من النوع `Creature` اسمه `creature` وأسندنا السلسلة `shark` إلى الحقل `Species`. بعد ذلك طبعنا المتغير `creature` لإظهار القيمة التي يتضمنها في الوقت الحالي، ثم مرّرنا المتغير `creature` (تمرير بالقيمة أي نُسخة) إلى الدالة `changeCreature` والتي بدورها تطبع قيمة المتغير المُمرر لها بعد إسناد السلسلة "jellyfish" إلى الحقل `Species` (هنا نطبعه من داخل الدالة أي محليًّا). بعد ذلك طبعنا قيمة المتغير `creature` مرةً أخرى (خارج الدالة السابقة).

لاحظ أنه يوجد لدينا ثلاث تعليمات طباعة؛ جرى السطر الأول والثالث من الخرج ضمن نطاق الدالة `main()` بينما كان السطر الثاني ضمن نطاق الدالة `changeCreature`. لاحظ أيضًا أنه في البداية كانت قيمة المتغير `creature` هي "shark" وبالتالي عند تنفيذ تعليمة الطباعة الأولى سيطبع:

```
(1) {Species:shark}
```

أما تعليمة الطباعة في السطر الثاني والموجودة ضمن نطاق الدالة `changeCreature`، فسنلاحظ أنها ستطبع القيمة:

```
(2) {Species:jellyfish}
```

لأننا عدلنا قيمة المتغير، أما في التعليمة الثالثة فقد يُخطئ البعض ويعتقد أنها ستطبع نفس القيمة التي طبعتها تعليمة السطر الثاني، لكن هذا لا يحدث لأن التعديل بقي محليًّا ضمن نطاق الدالة `changeCreature`، أي حدث التعديل على نسخة من المتغير `creature` وبالتالي لا ينتقل التعديل إلى المتغير الأساسي. إذًا سيكون خرج تعليمات الطباعة للسطرين الأول والثالث متطابق.

سنأخذ الآن نفس المثال، لكن سنغيّر عملية التمرير إلى الدالة changeCreature لتصبح تمرير بـالمرجع، وذلك من خلال تغيـير النـوع من creature إلى مؤشـر باسـتخدام المعامـل *، فبـدلًا من تمريـر creature، سنمرّر الآن مؤشرًا إلى creature أو *creature. كـان creature في المثـال السـابق من النـوع struct ويحتوي قيمة الحقل Species وهي "shark"، أما *creature فهو مؤشـر وليس struct، وبالتـالي قيمتـه هي موقع الذاكرة وهذا ما مرّرناه إلى الدالة changeCreature(). لاحظ أننا نضع المعامل & عند تمرير المتغير creature إلى الدالة.

```
package main
import "fmt"
type Creature struct {
  Species string
}
func main() {
  var creature Creature = Creature{Species: "shark"}
  fmt.Printf("1) %+v\n", creature)
  changeCreature(&creature)
  fmt.Printf("3) %+v\n", creature)
}
func changeCreature(creature *Creature) {
  creature.Species = "jellyfish"
  fmt.Printf("2) %+v\n", creature)
}
```

ستحصل عند تنفيذ الشيفرة السابقة على الخرج التالي:

```
1) {Species:shark}
2) &{Species:jellyfish}
3) {Species:jellyfish}
```

قد تبدو الأمور واضحة الآن، فعندما مرّرنا المتغير creature إلى الدالة changeCreature، كان التمريـر بالمرجع، وبالتالي أي تغيير يطرأ على المتغير creature (وهو تغيـير قيمـة الحقـل Species إلى "jellyfish") داخل هذه الدالة، سيكون مُطبّقًا على المتغير الأصلي نفسه الموجود ضـمن الدالـة main لأننـا نُعـدّل على نفس الموقع في الذاكرة، وبالتالي ستكون قيمة الخرج لتعليمات الطباعة 2 و 3 مُتطابقة.

قد لا يكون لدينا في بعض الأحيان قيمة مُعرّفة للمؤشر، وهذا قد يحدث لأسباب كثيرة منها مـا هـو متوقـع ومنها لا، وبالتالي قد يسبب لك panic حـالات انهيـار في البرنـامج. دعنـا نلقي نظـرةً على كيفيـة حـدوث ذلـك وكيفية التخطيط لتلك المشكلة المحتملة.

21.3 التأشير إلى اللاشيء Nil

القيمة الافتراضية لجميع المتغيرات في لغة جو هي الصفر، وهذا الكلام ينطبق أيضًا على المؤشرات. لدى التصريح عن مؤشر بنوعٍ ما ولكن دون أي قيمة مُسندة، ستكون القيمة الصفرية الافتراضية هي `nil`. الصفر هنا مفهوم متعلق بالنوع، أي أنه في حالة الأعداد الصحيحة هو العدد 0، وفي حالة السلاسل النصية هو السلسلة الفارغة `""`، وأخيرًا في حالة المؤشرات هو القيمة `nil` إشارةً إلى الحالة الافتراضية لقيمة أي مؤشر.

سنُعدّل في البرنامج التالي على البرنامج السابق، بحيث نعرّف مؤشرًا متغيرًا من النوع `Creature`، لكن دون استنساخ للنسخة الحقيقية من `Creature` ودون إسناد عنوانها إلى المؤشر؛ أي أن قيمة المؤشر هي `nil`، ولن نستطيع الرجوع إلى أي من الحقول أو التوابع المُعرّفة في النوع `Creature`. لنرى ماذا سيحدث:

```
package main
import "fmt"
type Creature struct {
  Species string
}
func main() {
  var creature *Creature
  fmt.Printf("1) %+v\n", creature)
  changeCreature(creature)
  fmt.Printf("3) %+v\n", creature)
}
func changeCreature(creature *Creature) {
  creature.Species = "jellyfish"
  fmt.Printf("2) %+v\n", creature)
}
```

سيكون الخرج على النحو التالي:

```
1) <nil>
panic: runtime error: invalid memory address or nil pointer
dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8
pc=0x109ac86]
goroutine 1 [running]:
```

```
main.changeCreature(0x0)
/Users/corylanou/projects/learn/src/github.com/gopherguides/learn/
_training/digital-ocean/pointers/src/nil.go:18 +0x26
  main.main()
/Users/corylanou/projects/learn/src/github.com/gopherguides/learn/
_training/digital-ocean/pointers/src/nil.go:13 +0x98
        exit status 2
```

نلاحظ عند تشغيل البرنامج أن تعليمة الطباعة الأولى 1 نجحت وطبعت قيمة المتغير `creature` وهي `<nil>`، لكن عندما وصلنا إلى استدعاء الدالة `changeCreature` ومحاولة ضبط قيمة الحقل `Species`، ظهرت حالة انهيار في البرنامج نظرًا لعدم إنشاء نسخة من هذا المتغيّر، وأدى هذا إلى محاولة الوصول إلى موقع ذاكري غير موجود أصلًا أو غير مُحدد.

هذا الأمر شائع في لغة جو، لذلك عندما تتلقى وسيطًا مثل مؤشر، لا بُد من فحصه إذا كان فارغًا أم لا قبل إجراء أي عمليات عليه، لتجنب حالات كهذه.

نتحقق عادةً من قيمة المؤشر في أي دالة تستقبل مؤشرًا مثل وسيط لها كما يلي:

```
if someVariable == nil {
  // هنا يمكن أن نطبع أي رسالة تُشير إلى هذه الحالة أو أن نخرج من الدالة
}
```

يمكنك بذلك التحقق مما إذا كان الوسيط يحمل قيمةً صفريةً أم لا. عند تمرير قيمة صفرية (هنا نقصد `nil`) قد ترغب في الخروج من الدالة باستخدام تعليمة `return` أو إعادة رسالة خطأ تخبر المستخدم أن الوسيط الممرّر إلى الدالة أو التابع غير صالح.

تتحقق الشيفرة التالية من وجود قيمة صفرية للمؤشر:

```
package main
import "fmt"
type Creature struct {
  Species string
}
func main() {
  var creature *Creature
  fmt.Printf("1) %+v\n", creature)
  changeCreature(creature)
  fmt.Printf("3) %+v\n", creature)
}
```

```
func changeCreature(creature *Creature) {
  if creature == nil {
        fmt.Println("creature is nil")
        return
  }
  creature.Species = "jellyfish"
  fmt.Printf("2) %+v\n", creature)
}
```

أضفنا إلى الدالة `changeCreature` تعليمات لفحص قيمة الوسيط `creature` فيما إذا كانت صفرية أم لا؛ ففي حال كانت صفرية نطبع "creature is nil" ونخرج من الدالة من خلال تعليمة `return`، وإلا نتابع العمل في الدالة ونُعدّل قيمة الحقل `Species`. سنحصل الآن على المخرجات التالية:

```
1) <nil>
creature is nil
3) <nil>
```

لاحظ أنه على الرغم من وجود حالة صفرية للمتغير، إلا أنه لم تحدث حالة انهيار للبرنامج لأننا عالجناها.

إذا أنشأنا نسخةً من النوع `Creature` وأُسندت للمتغير `creature`، سيتغير الخرج بالتأكيد، لأنه أصبح يُشير إلى موقع ذاكري حقيقي:

```
package main
import "fmt"
type Creature struct {
  Species string
}
func main() {
  var creature *Creature
  creature = &Creature{Species: "shark"}
  fmt.Printf("1) %+v\n", creature)
  changeCreature(creature)
  fmt.Printf("3) %+v\n", creature)
}
func changeCreature(creature *Creature) {
  if creature == nil {
        fmt.Println("creature is nil")
        return
```

```
    }
    creature.Species = "jellyfish"
    fmt.Printf("2) %+v\n", creature)
}
```

سنحصل على الناتج المتوقع التالي:

```
1) &{Species:shark}
2) &{Species:jellyfish}
3) &{Species:jellyfish}
```

عندما تتعامل مع المؤشرات، هناك احتمال أن يتعرّض البرنامج لحالة انهيار، لذلك يجب عليك التحقق لمعرفة ما إذا كانت قيمة المؤشر صفرية قبل محاولة الوصول إلى أي من الحقول أو التوابع المعرّفة ضمن نوع البيانات الذي يشير إليه.

دعنا نلقي نظرةً على كيفية استخدام المؤشرات مع التوابع.

21.4 مستقبلات مؤشرات التوابع

المُستقبل receiver في لغة جو هو الوسيط الذي يُعرّف عند التصريح عن التابع. ألقِ نظرةً على الشيفرة التالية:

```
type Creature struct {
    Species string
}
func (c Creature) String() string {
    return c.Species
}
```

المُستقبل في هذا التابع هو `c Creature`، وهو يُشير إلى أن نسخة المتغير `c` من النوع `Creature` وأنك ستستخدمه ليُشير إلى نسخة متغير من هذا النوع.

يختلف أيضًا سلوك التوابع كما هو الحال في الدوال التي يختلف فيها سلوك الدالة تبعًا لطريقة تمرير الوسيط (بالمرجع أو بالقيمة). ينبع الاختلاف الأساسي من أنه إذا صرّحت عن دالة مع مُستقبل "قيمة"، فلن تتمكن من إجراء تغييرات على نسخة هذا النوع التي عُرّف التابع عليه. عمومًا، ستكون هناك أوقات تحتاج فيها أن يكون تابعك قادرًا على تحديث نسخة المتغير الذي تستخدمه، ولإجراء هكذا تحديثات ينبغي عليك جعل المستقبل "مؤشرًا".

دعنا نضيف التابع `Reset` للنوع `Creature`، الذي يسند سلسلةً نصيةً فارغة إلى الحقل `Species`.

```
package main
import "fmt"
type Creature struct {
  Species string
}
func (c Creature) Reset() {
  c.Species = ""
}
func main() {
  var creature Creature = Creature{Species: "shark"}
  fmt.Printf("1) %+v\n", creature)
  creature.Reset()
  fmt.Printf("2) %+v\n", creature)
}
```

إذا شغّلت البرنامج ستحصل على الخرج:

```
1) {Species:shark}
2) {Species:shark}
```

لاحظ أنه على الرغم من ضبطنا قيمة الحقل `Species` على السلسلة الفارغة في التابع `Reset`، إلا أننا عندما طبعنا المتغير `creature` في الدالة `main` حصلنا على "shark". السبب في عدم انتقال التغيير هو استخدامنا مُستقبل قيمة في تعريف التابع `Reset`، وبالتالي سيكون لهذا التابع إمكانية التعديل فقط على نسخة المتغير `creature` وليس المتغير الأصلي. بالتالي، إذا أردنا تحديث هذه القيمة؛ أي التعديل على النسخة الأصلية للمتغير، فيجب علينا تعريف مُستقبل مؤشر.

```
package main
import "fmt"
type Creature struct {
  Species string
}
func (c *Creature) Reset() {
  c.Species = ""
}
func main() {
  var creature Creature = Creature{Species: "shark"}
  fmt.Printf("1) %+v\n", creature)
```

```
    creature.Reset()
    fmt.Printf("2) %+v\n", creature)
}
```

لاحظ أننا أضفنا المعامل * أمام النوع `Creature` عندما صرّحنا عن التابع `Reset`، وهذا يعني أن الوسـيط الذي نُمرره أصبح مؤشرًا، وبالتالي أصبحت كل التعديلات التي نُجريها من خلاله مُطبّقةً على المتغير الأصلي.

```
1) {Species:shark}
2) {Species:}
```

لاحظ أن التابع `Reset` عدّل قيمة الحقـل `Species` كمـا توقعنـا، وهـذا مُماثـل لفكـرة التمريـر بـالمرجع أو القيمة في الدوال.

21.5 خاتمة

تؤثر طريقة تمرير الوسطاء (بالمرجع أو القيمة) إلى التوابع أو الدوال على آليـة الوصـول إلى المتغـير المُمـرّر؛ ففي حالة التمرير بالمرجع يكون التعديل مباشرةً على المتغير الأصلي، أما في الحالـة الثانيـة فيكـون على نسـخة من المتغير. الآن بعد أن تعرفت على المؤشرات، أصبح بإمكانك التعرف على استخدامها مع الواجهات أيضًا.

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

22. البنى Structs

إنَّ بناء تجريدات abstractions لتفاصيل البرنـامج الخـاص بـك أعظم أداة يمكن أن تقـدمها لغـة البرمجـة للمطور، إذ تسمح البُنى structs لمطوري لغة جو بوصف العالم الذي يعمـل فيـه البرنـامج؛ فبـدلًا من اسـتخدام السلاسل النصية strings لوصف أشـياء، مثـل الشـارع Street والمدينـة City والرمـز البريـدي PostalCode، يمكن استخدام بنية تجمع كل هذه الأشياء تحت مفهوم "العنوان Address".

يمكن تعريف البنية على أنها هيكل بيانات يُستخدم لتعريف نوع بيانات جديد يحتـوي مجموعـةً محـددةً من القيم مختلفـة النـوع، ويمكن الوصـول لهـذه العناصـر أو القيم عن طريـق اسـمها. تسـاعد البـنى المطـورين المستقبليين (بما في ذلك نحن) بتحديد البيانات المهمة للبرامج الخاصة بنا وكيف يجب أن تستخدم الشيفرات المستقبلية هذه البيانات بالطريقة الصحيحة.

يمكن تعريف البنى واستخدامها بعدة طرق مختلفة في الشيفرات البرمجية. وسنلقي نظرةً في هـذا الفصـل على كل من هذه التقنيات.

22.1 تعريف البنى

يمكنك أن تتخيل البنى مثل نماذج جوجل التي يُطلب منك ملؤها أحيانًا. تتضمّن هذه النماذج حقولًا يُطلب منك تعبئتها، مثل الاسم، أو العنوان، أو البريد الإلكتروني، أو مربعات فارغة يمكن تحديد إحداها لوصـف حالتـك الزوجية (أعزب، متزوج، أرمل) ...إلخ. يمكن أن تتضمن البُنى أيضًـا حقـولًا يمكنـك تعبئتهـا. تهيئـة متغـير ببنيـة جديدة، أشبه باستخراج نسخة من نموذج جاهز للتعبئة.

لإنشاء بنية جديدة يجب أولاً إعطاء لغة جو مُخططًا يصف الحقول الـتي تحتـوي عليهـا البنيـة. يبـدأ تعريـف البنية بالكلمة المفتاحية `type` متبوعًا باسم البنية الذي تختـاره `Creature` ثم الكلمـة `struct` ثم قوسـين `{}`

نضع بداخلهما الحقول التي نريدها في البنية. يمكنك استخدام البنية بعد الانتهاء من التصريح عنها مع المتغيرات كما لو أنها نوع بيانات بحد ذاته.

```
package main
import "fmt"
type Creature struct {
  Name string
}
func main() {
  c := Creature{
        Name: "Sammy the Shark",
  }
  fmt.Println(c.Name)
}
```

ستحصل عند تشغيل البرنامج على الخرج التالي:

```
Sammy the Shark
```

صرّحنا في هذا المثال أولًا عن بنية اسمها `Creature` تتضمّن حقلًا اسمه `Name` من نوع سلسلة نصية `string`. نُعرّف داخل الدالة `main` مُتغيرًا `c` من النوع `Creature` ونهيئ الحقل `Name` فيه بالقيمة "Sammy the Shark"، إذ نفتح قوسين `{}` ونضع هذه المعلومات بينهما كما في الشيفرة أعلاه. أخيرًا استدعينا الدالة `fmt.Println` وطبعنا من خلالها الحقل `Name` من خلال المتغير `c` وذلك عن طريق وضع اسم المتغير متبوعًا بنقطة `.` ومتبوعًا باسم الحقل، مثل `c.Name`، الذي يعيد في هذه الحالة حقل `Name`.

عندما نأخذ نسخةً من بنية، نذكر غالبًا اسم كل حقل ونسند له قيمة (كما في المثال السابق). عمومًا، إذا كنت ستؤمن قيمة كل حقل أثناء تعريف نسخة من بنية فيمكنك عندها تجاهل كتابة أسماء الحقول، كما يلي:

```
package main
import "fmt"
type Creature struct {
  Name string
  Type string
}
func main() {
  c := Creature{"Sammy", "Shark"}
  fmt.Println(c.Name, "the", c.Type)
```

```
}
```

وسيكون الخرج على النحو التالي:

```
Sammy the Shark
```

أضفنا حقلًا جديدًا للبنية `Creature` باسم `Type` وحددنا نوعه `string`. أنشأنا داخل الدالة `main` نسخةً من هذه البنية وهيّأنا قيم الحقلين `Name` و `Type` بالقيمتين "Sammy" و "shark" على التوالي من خلال التعليمة التالية:

```
Creature{"Sammy", "Shark"}
```

واستغنينا عن كتابة أسماء الحقول صراحةً.

نُسمي هذه الطريقة بالطريقة المُختصرة للتصريح عن نسخة من بنية، وهذه الطريقة غير شائعة كثيرًا لأنها تتضمن عيوبًا مثل ضرورة تحديد قيم لجميع الحقول وبالترتيب وعدم نسيان أي حقل. نستنتج سريعًا أن استخدام هذه الطريقة لا يكون جيدًا عندما يكون لدينا عددٌ كبيرٌ من الحقول لأننا سنكون عُرضةً للخطأ والنسيان والتشتت عندما نقرأ الشيفرة مرةً أخرى. إذًا، يُفضّل استخدام هذه الطريقة فقط عندما يكون عدد الحقول قليل.

ربما لاحظت أننا نبدأ أسماء جميع الحقول بحرف كبير، وهذا مهم جدًا لأنه يلعب دورًا في تحديد إمكانية الوصول إلى هذه الحقول؛ فعندما نبدأ اسم الحقل بحرف كبير، فهذا يعني إمكانية الوصول إليه من خارج الحزمة، أما الحرف الصغير فلا يمكن الوصول إليها من خارج الحزمة.

22.2 تصدير حقول البنية

يعتمد تصدير حقول البنية على نفس قواعد تصدير المكونات الأخرى في لغة جو؛ فإذا بدأ اسم الحقل بحرف كبير، فسيكون قابلاً للقراءة والتعديل بواسطة شيفرة من خارج الحزمة التي صُرّح عنه فيها؛ أما إذا بدأ الحقل بحرف صغير، فلن تتمكن من قراءة وتعديل هذا الحقل إلا من شيفرة من داخل الحزمة التي صُرّح عنه فيها. يوضّح المثال التالي الأمر:

```
package main
import "fmt"
type Creature struct {
  Name string
  Type string
  password string
}
func main() {
```

```
    c := Creature{
          Name: "Sammy",
          Type: "Shark",
          password: "secret",
    }
    fmt.Println(c.Name, "the", c.Type)
    fmt.Println("Password is", c.password)
}
```

وسيكون الخرج على النحو التالي:

```
Sammy the Shark
Password is secret
```

أضفنا إلى البنية السابقة حقلًا جديدًا `secret`، وهو حقل من النوع `string` ويبدأ بحرف صغير؛ أي أنه غير مُصدّر، وأي حزمة أخرى تحاول إنشاء نسخة من هذه البنية `Creature` لن تتمكن من الوصول إلى حقل `secret`. عمومًا، يمكننا الوصول إلى هذا الحقل ضمن نطاق الحزمة، لذا إذا حاولنا الوصول إلى هذا الحقل من داخل الدالة `main` والتي بدورها موجودة ضمن نفس الحزمة بالتأكيد، فيمكننا الرجوع لهذا الحقل `c.password` والحصول على القيمة المُخزنة فيه. وجود حقول غير مُصدّرة أمر شائع في البنى مع إمكانية وصول بواسطة توابع مُصدّرة exported.

22.3 البنى المضمنة Inline Structs

تُسمى أيضًا البنى السريعة. إذ تُمكّنك لغة جو من تعريف بُنى في أي وقت تريده وفي أي مكان ودون الحاجة للتصريح عنها على أنها نوع بيانات جديد بحد ذاته، وهذا مفيد في الحالات التي تكون فيها بحاجة إلى استخدام بنية مرةً واحدةً فقط (أي لن تحتاج إلى إنشاء أكثر من نسخة)، فمثلًا، تستخدم الاختبارات غالبًا بنيةً لتعريف جميع المعاملات التي تُشكل حالة اختبار معينة. سيكون ابتكار أسماء جديدة مثل `CreatureNamePrintingTestCase` مرهقًا لدى استخدام هذه البنية في مكان واحد فقط.

يكون كتابة البنى المُضمّنة بنفس طريقة كتابة البنى العادية تقريبًا، إذ نكتب الكلمة المفتاحية `struct` متبوعةً بقوسين `{}` نضع بينهما الحقول وعلى يمين اسم المتغير. يجب أيضًا وضع قيم لهذه الحقول مباشرةً من خلال استخدام قوسين آخرين `{}` كما هو موضح في الشيفرة التالية:

```
package main
import "fmt"
func main() {
  c := struct {
```

```
        Name string
        Type string
    }{
        Name: "Sammy",
        Type: "Shark",
    }
    fmt.Println(c.Name, "the", c.Type)
}
```

ويكون الخرج كما يلي:

```
Sammy the Shark
```

لاحظ أننا لم نحتاج إلى تعريف نوع بيانات جديد لتمثيل البنية، وبالتالي لم نكتب الكلمة المفتاحية `type`، فكل ما نحتاجه هنا هو استخدام الكلمة المفتاحية `struct` إشارةً إلى البنية، وإلى معامل الإسناد القصير `:=`. نحتاج أيضًا إلى تعريف قيم الحقول مباشرةً كما فعلنا في الشيفرة أعلاه. الآن أصبح لدينا متغيرًا اسمه `c` يُمثّل بنيةً يمكن الوصول لحقولها من خلال النقطة `.` كما هو معتاد. سترى البنى المُضمّنة غالبًا في الاختبارات، إذ تُعرّف البنى الفردية بصورةٍ متكررة لاحتواء البيانات والتوقعات لحالة اختبار معينة.

22.4 خاتمة

تعرفنا في هذا الفصل على مفهوم البُنى وهي كتل بيانات غير متجانسة (أي أنها تضم حقولًا أو عناصر من أنواع بيانات مختلفة) يعرّفها المبرمجون لتنظيم المعلومات. تتعامل معظم البرامج مع أحجام هائلة من البيانات، وبدون البنى سيكون من الصعب تذكر أي من المتغيرات ترتبط معًا وأيّها غير مرتبطة أو أيّها من نوع `string` وأيها من نوع `int`. لذلك إذا كنت تتعامل مع مجموعة من المتغيرات، اسأل نفسك عما إذا كان تجميعها ضمن بنية سيكون أفضل، إذ من الممكن أن تصف هذه المتغيرات مفهومًا عالي المستوى، فيمكن مثلًا أن يشير أحد المتغيرات إلى عنوان شركة حسوب، وهناك متغيّر آخر يخص عنوان شركة أُخرى.

دورة تطوير التطبيقات
باستخدام لغة بايثون

احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن

23. تعريف التوابع Methods

تُمكّنك الدوال Functions من تنظيم منطـق مُحـدد ضـمن إجـراءات procedures قابلـة للتكـرار يمكنهـا استخدام وسطاء مختلفة في كل مرة تُنفّذ فيها. تعمل عدة دوال غالبًا على نفس الجزء من البيانات في كل مــرة. يُمكن لغة جو التعرّف على هذه الأنماط، وتسمح لك بتعريف دوال خاصـة تُسـمى "التوابـع Methods"، وتجـري عمليات على نوع بيانات مُحدد يُشار إليه باسم "المُستقبل Receiver".

23.1 تعريف تابع

قواعد كتابة التوابع مُشابهة لقواعد كتابة الدوال، والاختلاف الوحيـد هـو إضـافة معامـل يُوضـع بعـد الكلمـة المفتاحية `func` لتحديد مُستقبل التابع. المُستقبل هو تصريح لنوع البيانات الذي تُريد تعريـف التـابع من أجلـه. نُصرّح في المثال التالي عن تابع يُطبّق على نوع بنية struct:

```
package main
import "fmt"
type Creature struct {
  Name   string
  Greeting string
}
func (c Creature) Greet() {
  fmt.Printf("%s says %s", c.Name, c.Greeting)
}
func main() {
  sammy := Creature{
        Name:   "Sammy",
```

```
        Greeting: "Hello!",
    }
    Creature.Greet(sammy)
}
```

نحصل عند تشغيل الشيفرة على ما يلي:

```
Sammy says Hello!
```

أنشأنا بنيةً أسميناها `Creature` تمتلك حقلين من النوع `string` باسم `Name` و `Greeting`. لدى `Creature` تابع وحيد مُعرّف يُسمى `Greet`. أسندنا أثناء التصريح عن المستقبل نسخةً من `Creature` إلى المتغير `c`، بحيث يُمكننا من خلالها الوصول إلى حقول `Creature` وطباعة محتوياته باستخدام دالة `fmt.Printf`.

يُشار عادةً للمُستقبل في لغات البرمجة الأخرى بكلمات رئيسية مثل `self` كما في لغة بايثون Python أو `this` كما في لغة جافا Java. يُعد المستقبل في لغة جو مُتغيرًا كما باقي المتغيرات، وبالتالي يُمكنك تسميته كما تشاء. يُفضّل عمومًا تسميته بأول حرف من اسم نوع البيانات الذي تُريد أن يتعامل معه التابع، فقد سُمّي المعامل في المثال السابق بالاسم `c` لأن نوع بيانات المستقبل كان `Creature`.

أنشأنا داخل الدالة `main` نسخةً من البنية `Creature` وأعطينا حقولها `Name` و `Greeting` القيم "Sammy" و "Hello!" على التوالي. استدعينا التابع `Greet` والذي أصبح مُعرّفًا على نوع البيانات `Creature` من خلال وضع نقطة `Creature.Greet`، ومرّرنا له النسخة التي أنشأناه للتو مثل وسيط. قد يبدو هذا الأسلوب في الاستدعاء مُربكًا قليلًا، سنعرض فيما يلي أسلوبًا آخر لاستدعاء التوابع:

```
package main
import "fmt"
type Creature struct {
    Name   string
    Greeting string
}
func (c Creature) Greet() {
    fmt.Printf("%s says %s", c.Name, c.Greeting)
}
func main() {
    sammy := Creature{
        Name:   "Sammy",
        Greeting: "Hello!",
```

```
    }
    sammy.Greet()
}
```

ستحصل عند تشغيل الشيفرة على نفس الخرج السابق:

```
Sammy says Hello!
```

لا يختلف هذا المثال عن المثال السابق إلا في استدعاء التابع `Greet`، إذ استخدمنا هنا الأسلوب المختصر، بحيث نضع اسم المتغير الذي يُمثّل البنية `Creature` وهو `sammy` متبوعًا بنقطة ثم اسم التابع بين قوسين `sammy.Greet()`.

يُحبّذ دومًا استخدام هذا الأسلوب الذي يُسمّى التدوين النقطي dot notation، فمن النادر أن يستخدم المطورون الأسلوب السابق الذي يُسمّى أسلوب الاستدعاء الوظيفي functional invocation style. ويوضح المثال التالي أحد الأسباب التي تجعل هذا الأسلوب أكثر انتشارًا:

```
package main
import "fmt"
type Creature struct {
    Name   string
    Greeting string
}
func (c Creature) Greet() Creature {
    fmt.Printf("%s says %s!\n", c.Name, c.Greeting)
    return c
}
func (c Creature) SayGoodbye(name string) {
    fmt.Println("Farewell", name, "!")
}
func main() {
    sammy := Creature{
          Name:   "Sammy",
          Greeting: "Hello!",
    }
    sammy.Greet().SayGoodbye("gophers")
    Creature.SayGoodbye(Creature.Greet(sammy), "gophers")
}
```

عند تشغيل الشيفرة السابقة سيكون الخرج على النحو التالي:

```
Sammy says Hello!!
Farewell gophers !
Sammy says Hello!!
Farewell gophers !
```

عدّلنا الأمثلة السابقة لكي نُقدّم تابعًا جديدًا يُسمى SayGoodbye، وعدّلنا أيضًا تعريف التابع Greet بحيث يُعيد أيضًا المُعامل c الذي يُمثل Creature، وبالتالي سيكون لدينا إمكانية استخدام القيمة المُعادة من هذا التابع والمُتمثّلة بنسخة Creature. نستدعي في الدالة main التابعين SayGoodbye و Greet على المتغير sammy باستخدام أسلوبي التدوين النقطي والاستدعاء الوظيفي.

يعطي الأسلوبان نفس النتائج في الخرج، لكن التدوين النُقطي أسهل للقراءة، كما أنه يعرض لنا عملية استدعاء التوابع مثل تسلسل، إذ نستدعي التابع Greet من خلال المتغير sammy الذي يُمثّل Creature، ثم نستدعي التابع SayGoodbyeمن خلال القيمة التي يُعيدها هذا التابع والتي تمثّل Creature أيضًا.لا يعكس أسلوب الاستدعاء الوظيفي هذا الترتيب بسبب إضافة معامل إلى استدعاء SayGoodbye يؤدي إلى حجب ترتيب الاستدعاءات. وضوح التدوين النقطي هو السبب في أنه النمط المفضل لاستدعاء التوابع في لغة جو، سواءً في المكتبة القياسية أو في الحزم الخارجية.

تعريف التوابع على نوع بيانات مُحدد هو أمر مُختلف عن تعريف الدوال التي تعمل بناءً على قيمة ما، وهذا له أهمية خاصة في لغة جو لأنه مفهوم أساسي في الواجهات interfaces.

23.2 الواجهات interfaces

عندما تُعرّف أي تابع على نوع بيانات في لغة جو، يُضاف هذا التابع إلى مجموعة التوابع الخاصة بهذا النوع، وهي مجموعة من الدوال المرتبطة بهذا النوع ويستخدمها مُصرّف لغة جو لتحديد إذا كان يمكن إسناد نوع ما لمتغير من نوع "واجهة interface"؛ وهذا النوع هو نهج يعتمده المُصرّف لضمان أن مُتغيّرًا من نوع البيانات المطلوب يُحقق التوابع التي تتضمنها الواجهة.

يُعد أي نوع يمتلك توابع لها نفس الاسم ونفس المعلمات ونفس القيم المُعادة؛ مثل تلك الموجودة في تعريف الواجهة منفِّذًا لتلك الواجهة ويُسمح بإسناده إلى متغيرات من تلك الواجهة. نعرض فيما يلي تعريفًا لواجهة fmt.Stringer من المكتبة القياسية:

```
type Stringer interface {
 String() string
}
```

لاحظ هنا استخدام الكلمة المفتاحية type لتعريف نوع بيانات جديد يُمثّل واجهة.

لكي ينفّذ أي نوع الواجهة `fmt.Stringer`، يجب أن يوفر تابعًا اسمه `String()` يُعيد سلسلة نصية. سيُمكّنك تنفيذ هذه الواجهة من طباعة "نوعك" تمامًا كما تريد ويُسمى ذلك "طباعة مُرتّبة pretty-printed"، وذلك عند تمرير نسخة من النوع الخاص بك إلى دوال محددة في حزمة `fmt`.

يُعرّف المثال التالي نوعًا ينفّذ هذه الواجهة:

```
package main
import (
  "fmt"
  "strings"
)
type Ocean struct {
  Creatures []string
}
func (o Ocean) String() string {
  return strings.Join(o.Creatures, ", ")
}
func log(header string, s fmt.Stringer) {
  fmt.Println(header, ":", s)
}
func main() {
  o := Ocean{
        Creatures: []string{
              "sea urchin",
              "lobster",
              "shark",
        },
  }
  log("ocean contains", o)
}
```

ويكون الخرج على النحو التالي:

```
ocean contains : sea urchin, lobster, shark
```

صرّحنا في هذا المثال عن نوع بيانات جديد يُمثّل بنيةً اسمها `Ocean`. يُمكننا القول أن هذه البنية تنفّذ الواجهة `fmt.Stringer` لأنها تعرّف تابعًا اسمه `String` لا يأخذ أي وسطاء ويعيد سلسلةً نصية، أي تمامًا كما

في الواجهة. عرّفنا داخل الدالة `main` متغيرًا `o` يُمثّل بنيةً `Ocean` ومرّرناه إلى الدالة `log` التي تطبع سلسة نصية تُمرّر لها، متبوعةً بأي شيء تُنفّذه وليكن `fmt.Stringer`.

سيسمح لنا مُصرّف جو هنا أن نُمرّر البنية `o` لأنه يُنفّذ كل التوابع التي تطلبها الدالة `fmt.Stringer` (هنا يوجد تابع وحيد `String`). نستخدم داخل الدالة `log` دالة الطباعة `fmt.Println` التي تستدعي التابع `String` من البنية `Ocean` لأننا مررنا لها المعامل `s` من النوع `fmt.Stringer` في ترويستها (أي بمثابة أحد معاملاتها).

إذا لم تنفّذ البنية `Ocean` التابع `String` سيُعطينا جو خطأً في التصريف، لأن الدالة `log` تتطلب تمرير وسيط من النوع `fmt.Stringer`، وسيكون الخطأ كما يلي:

```
src/e4/main.go:24:6: cannot use o (type Ocean) as type fmt.Stringer in
argument to log:
    Ocean does not implement fmt.Stringer (missing String method)
```

سيتحقق مُصرّف لغة جو من مطابقة التابع `String()` للتابع المُستدعى من قِبل الواجهة `fmt.Stringer`، وإذا لم يكن مُطابقًا، سيعطي الخطأ:

```
src/e4/main.go:26:6: cannot use o (type Ocean) as type fmt.Stringer in
argument to log:
    Ocean does not implement fmt.Stringer (wrong type for String
method)
        have String()
        want String() string
```

استخدمت التوابع المُعرّفة في الأمثلة السابقة "مُستقبل قيمة"، أي إذا استخدمنا أسلوب الاستدعاء الوظيفي للتوابع، سيكون المعامل الأول الذي يشير إلى النوع الذي عُرّف التابع عليه قيمةً من هذا النوع وليس مؤشرًا؛ وهذا يعني أن أي تعديل نُجريه على هذا النوع المُمثّل بالمعامل المُحدد (مثلًا في المثال السابق كان هذا المعامل هو `o`) سيكون محليًّا وسيُنفّذ على نسخة من البيانات ولن يؤثر على النسخة الأصلية. سنرى فيما يلي أمثلة تستخدم مُستقبلات مرجعية "مؤشر".

23.3 مستقبلات مثل مؤشرات

يُشبه استخدام المؤشرات مثل مستقبلات في التوابع إلى حد كبير استخدام "مستقبل القيمة"، والفرق الوحيد هو وضع علامة `*` قبل اسم النوع. يوضح المثال التالي تعريف تابع على مستقبل مؤشر إلى نوع:

```
package main
import "fmt"
```

```
type Boat struct {
  Name string
  occupants []string
}
func (b *Boat) AddOccupant(name string) *Boat {
  b.occupants = append(b.occupants, name)
  return b
}
func (b Boat) Manifest() {
  fmt.Println("The", b.Name, "has the following occupants:")
  for _, n := range b.occupants {
        fmt.Println("\t", n)
  }
}
func main() {
  b := &Boat{
        Name: "S.S. DigitalOcean",
  }
  b.AddOccupant("Sammy the Shark")
  b.AddOccupant("Larry the Lobster")
  b.Manifest()
}
```

سيكون الخرج على النحو التالي:

```
The S.S. DigitalOcean has the following occupants:
   Sammy the Shark
   Larry the Lobster
```

يُعرّف هذا المثال نوعًا يُسمّى `Boat` يمتلك حقلين هما `Name` و `occupants`. نريد حماية الحقل `occupants` بحيث لا تتمكن الشيفرات الأخرى من خارج الحزمة أن تُعدّل عليه إلا من خلال التابع `AddOccupant`، لهذا السبب جعلناه حقلًا غير مُصدّر، وذلك بجعل أول حرف صغيرًا. نريد أيضًا جعل التعديلات التي يُجريها التابع `AddOccupant` على نفس المتغير وبالتالي نحتاج إلى تمريره بالمرجع؛ أي يجب أن نُعرّف مُستقبل مؤشر `(b *Boat)` وليس مُستقبل قيمة، إذ سيعمل مُستقبل المؤشر على إجراء التعديلات على نفس بيانات المتغير الأصلي الموجودة في الذاكرة من خلال تمرير عنوانه. تعمل المؤشرات مثل مراجع إلى

متغير من نوع محدد بدلاً من نسخة من هذا النوع، لذلك سيضمن تمرير عنوان متغير من النوع `Boat` إلى التابع `AddOccupant`تنفيذ التعديلات على المتغير نفسه وليس نسخةً منه.

نُعرّف داخل الدالة `main` متغيرًا `b` يحمل عنوان بنية من النوع `Boat`، وذلك من خلال وضع `&` قبل تعريف البنية `(*Boat)` كما في الشيفرة أعلاه. استدعينا التابع `AddOccupant` مرتين لإضافة عُنصرين.

يُعرّف التابع `Manifest` على النوع `Boat` ويستخدم مُستقبل قيمة `(b Boat)`. ما زلنا قادرين على استدعاء `Manifest` داخل الدالة `main` لأن لغة جو قادرة على تحصيل dereference قيمة المؤشر تلقائيًا من `Boat`، إذ تكافئ `b.Manifest()` هنا `(*b).Manifest()`.

بغض النظر عن نوع المستقبل الذي تُعرّفه لتابع ما؛ مستقبل مؤشر أو مستقبل القيمة، فإن له آثارًا مهمة عند محاولة إسناد قيم للمتغيرات من نوع واجهة.

23.4 المستقبلات مثل مؤشرات والواجهات

عندما تحاول إسناد قيمة متغير من نوع محدد إلى متغير نوعه واجهة، يتأكد مُصرّف جو ما إذا كان ذلك النوع يُنفّذ كل التوابع التي تتطلبها الواجهة. تختلف مجموعة التوابع لمستقبل المؤشر عن مستقبل القيمة لأن التوابع التي تتلقى مؤشرًا يمكنها التعديل على المُستقبل الخاص بها، بينما لا يمكن لتلك التي تتلقى قيمة فعل ذلك.

```
package main
import "fmt"
type Submersible interface {
  Dive()
}
type Shark struct {
  Name string
  isUnderwater bool
}
func (s Shark) String() string {
  if s.isUnderwater {
        return fmt.Sprintf("%s is underwater", s.Name)
  }
  return fmt.Sprintf("%s is on the surface", s.Name)
}
func (s *Shark) Dive() {
  s.isUnderwater = true
```

```
}
func submerge(s Submersible) {
  s.Dive()
}
func main() {
  s := &Shark{
        Name: "Sammy",
  }
  fmt.Println(s)
  submerge(s)
  fmt.Println(s)
}
```

سيكون الخرج على النحو التالي:

```
Sammy is on the surface
Sammy is underwater
```

عرّفنا واجهة تُسمى `Submersible`، تقبل أنواعًا تُنفّذ التابع `Dive()` الخاص بها. عرّفنا أيضًا النوع `Shark` مع الحقل `Name` والتابع `isUnderwater` لتتبع حالة متغيرات هذا النوع. عرّفنا أيضًا التابع `Dive()` مع مُستقبل مؤشر من النوع `Shark`، إذ يُعدّل هذا التابع قيمة التابع `isUnderwater` لتصبح `true`. عرّفنا أيضًا التابع `String()` مع مُستقبل قيمة من النوع `Shark` لطباعة حالة `Shark` بأسلوب مُرتب باستخدام `fmt.Println` ومن خلال الواجهة `fmt.Stringer` التي تعرّفنا عليها مؤخرًا. عرّفنا أيضًا الدالة `submerge` التي تأخذ معاملًا من النوع `Submersible`.

يسمح لنا استخدام الواجهة `Submersible` بدلًا من `*Shark` في الدالة `submerge` جعل هذه الدالة تعتمد فقط على السلوك الذي يوفره النوع، وبالتالي جعلها أكثر قابلية لإعادة الاستخدام، فلن تضطر إلى كتابة دوال `submerge` جديدة لحالات خاصة أخرى مثل `Submarine` أو `Whale` أو أي كائنات مائية أخرى في وقت لاحق، فطالما أنها تُعرّف التابع `Dive()` يمكنها أن تعمل مع الدالة `submerge`.

نُعرّف داخل الدالة `main` المتغير `s` الذي يمثّل مؤشرًا إلى `Shark` ونطبعه مباشرةً باستخدام الدالة `fmt.Println` مما يؤدي إلى طباعة `Sammy is on the surface` على شاشة الخرج. نمرر بعدها المتغير `s` إلى الدالة `submerge` ثم نستدعي الدالة `fmt.Println` مرةً أخرى على المتغير `s` مما يؤدي لطباعة النتيجة التالية `Sammy is underwater`.

إذا جعلنا `s` من النوع `Shark` بدلًا من `*Shark`، سيعطي المُصرّف الخطأ التالي:

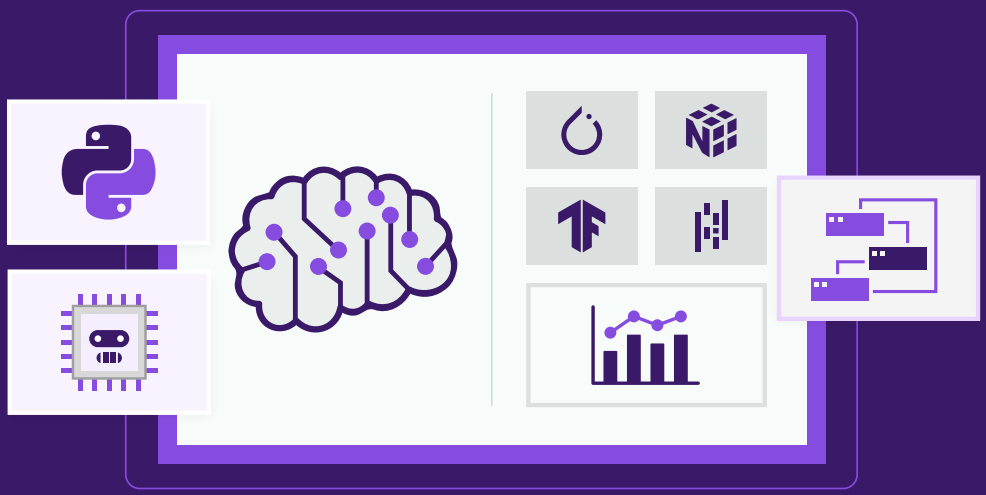
```
cannot use s (type Shark) as type Submersible in argument to submerge:

  Shark does not implement Submersible (Dive method has pointer
receiver)
```

يخبرنا مُصرّف جو أن `Shark` تمتلك التابع `Dive` وأن ذلك معرّفٌ في مستقبل المؤشر. عندما ترى رسالة الخطأ هذه، فإن الحل هو تمرير مؤشر إلى نوع الواجهة باستخدام العامل `&` قبل اسم المتغير الذي ترغب بإسناده لمتغير آخر.

23.5 خاتمة

لا يختلف التصريح عن التوابع Methods في لغة جو كثيرًا عن تعريف الدوال Functions التي تستقبل أنواعًا مختلفة من المتغيرات، إذ تنطبق نفس قواعد العمل مع المؤشرات. وتوفر لغة جو الراحة في تعريف الدوال وتسمح لك بجمعها في مجموعات من التوابع التي يمكن تفسيرها من خلال أنواع الواجهة. وسيسمح لك استخدام التوابع بطريقة فعّالة بالعمل مع الواجهات في شيفراتك البرمجية وتحسين إمكانية الاختبار، كما يجعل الشيفرة أكثر تنظيمًا وسهلة القراءة للمطورين الآخرين.

دورة الذكاء الاصطناعي



تعلم الذكاء الاصطناعي وتعلم الآلة والتعلم العميق
وتحليل البيانات، وأضفها إلى تطبيقاتك

التحق بالدورة الآن

24. بناء البرامج وتثبيتها

استخدمنا خلال الفصول السابقة من الكتاب الأمر `go run` كثيرًا، وكان الهدف منه تشغيل شيفرة البرنامج الخاص بنا، إذ أنه يُصرّف شفرة المصدر تلقائيًا ويُشغّل الملف التنفيذي أو القابل للتنفيذ executable الناتج. يُعد هذا الأمر مفيدًا عندما تحتاج إلى اختبار برنامجك من خلال سطر الأوامر، لكن عندما ترغب بنشر تطبيقك، سيتطلب منك ذلك بناء تعليماتك البرمجية في ملف تنفيذي، أو ملف واحد يحتوي على شيفرة محمولة أو تنفيذية (يُطلق عليه أيضًا الكود-باي أو كود البايت p-code، وهو شكل من أشكال مجموعة التعليمات المصممة للتنفيذ الفعّال بواسطة مُصرّف برمجي) يمكنها تشغيل تطبيقك. ولإنجاز ذلك يمكنك استخدام سلسلة أدوات جو لبناء البرنامج وتثبيته.

تسمى عملية ترجمة التعليمات البرمجية المصدر إلى ملف تنفيذي في لغة جو بالبناء؛ فعند بناء هذا الملف التنفيذي ستُضاف إليه الشيفرة اللازمة لتنفيذ البرنامج التنفيذي الثنائي binary على النظام الأساسي المُستهدف. هذا يعني أن جو التنفيذي Go binary (خادم مفتوح المصدر أو حزمة برمجية تسمح للمستخدمين الذين لا يستخدمون جو بتثبيت الأدوات المكتوبة بلغة جو بسرعة، دون تثبيت مُصرّف جو أو مدير الحزم - كل ما تحتاجه هو curl) لا يحتاج إلى اعتماديات dependencies النظام مثل أدوات جو للتشغيل على نظام جديد. سيسمح وضع هذه الملفات التنفيذية ضمن مسار ملف تنفيذي على نظامك، بتشغيل البرنامج من أي مكان في نظامك؛ أي كما لو أنك تُثبّت أي برنامج عادي على نظام التشغيل الخاص بك.

ستتعلم في هذا الفصل كيفية استخدام سلسلة أدوات لغة جو Go toolchain لتشغيل وبناء وتثبيت برنامج "Hello، World!" لفهم كيفية استخدام التطبيقات البرمجية وتوزيعها ونشرها بفعالية.

24.1 المتطلبات

ستحتاج إلى امتلاك مساحة عمل خاصة مُهيئة وجاهزة في لغة جو Go، فإذا لم تكن قد أنشأت واحدة، فارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبَّت لغة جو، وجهز بيئة تطوير محلية بحسب نظام تشغيلك.

24.2 إعداد وتشغيل جو التنفيذي Go Binary

سننشئ بدايةً تطبيقًا بسيطًا بلغة جو يطبع العبارة الترحيبية "Hello، World!"، وذلك لتوضيح سلسلة أدوات جو، وانظر فصل كتابة برنامجك الأول في جو Go إن لم تتطلع عليه مسبقًا.

أنشئ مجلد `greeter` داخل المجلد `src`:

```
$ mkdir greeter
```

بعد ذلك أنشئ ملف main.go بعد الانتقال إلى هذا المجلد الذي أنشأته للتو، ويمكنك استخدام أي محرر نصوص تختاره لإنشاء الملف، هنا استخدمنا محرر نانو nano:

```
$ cd greeter
$ nano main.go
```

بعد فتح الملف، أضف المحتويات التالية إليه:

```
package main
import "fmt"
func main() {
  fmt.Println("Hello, World!")
}
```

عند تشغيل هذا البرنامج سيطبع العبارة "Hello, World!" ثم سينتهي البرنامج. احفظ الملف الآن وأغلقه.

استخدم الأمر `go run` لاختبار البرنامج:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
Hello, World!
```

كما تحدّثنا سابقًا؛ يبني الأمر `go run` الملف المصدري الخاص بك في ملف تنفيذي، ثم يُصرّفه ويعرض الناتج. نريد هنا أن نتعلم كيفية بناء الملف التنفيذي بطريقة تمكننا من توزيع ونشر برامجنا، ولذلك سنستخدم الأمر `go build` في الخطوة التالية.

24.3 إنشاء وحدة جو من أجل Go binary

بُنيت برامج ومكتبات جو وفق المفهوم الأساسي "للوحدة module"، إذ تحتوي الوحدة على معلومات حول المكتبات التي يستخدمها برنامجك وإصدارات هذه المكتبات التي يجب استخدامها. لتخبر جو أن مجلدًا ما هو وحدة، ستحتاج إلى إنشاء هذا المجلد باستخدام الأمر `go mod`:

```
$ go mod init greeter
```

سيؤدي ذلك إلى إنشاء الملف go.mod الذي يتضمن اسم الوحدة ونسخة جو المستخدمة في إنشائها.

```
go: creating new go.mod: module greeter
go: to add module requirements and sums:
  go mod tidy
```

سيطالبك جو بتشغيل `go mod tidy` لتحديث متطلبات هذه الوحدة إذا تغيرت في المستقبل، ولن يكون لتشغيله الآن أي تأثير إضافي.

24.4 بناء الملفات التنفيذية باستخدام الأمر go build

يمكنك بناء ملف تنفيذي باستخدام الأمر `go build` من أجل تطبيق مكتوب بلغة جو، مما يسمح لك بتوزيعه ونشره في المكان الذي تريده.

سنجرب ذلك مع ملف main.go. من داخل المجلد `greeter` من خلال تنفيذ الأمر التالي:

```
$ go build
```

إذا لم تُقدم وسيطًا لهذا الأمر، سيُصرّف الأمر `go build` تلقائيًا برنامج main.go في مجلدك الحالي، وسيتضمن ذلك أيضًا كل الملفات التي يكون امتدادها `*.go` في هذا المجلد. سيبني أيضًا جميع التعليمات البرمجية الداعمة واللازمة لتكون قادرًا على تنفيذ البرنامج التنفيذي على أي جهاز حاسوب له نفس معمارية النظام، بغض النظر عما إذا كان هذا النظام يحتوي على أدوات جو أو مُصرّف جو أو ملفاته المصدرية.

إذًا، فقد بنيت تطبيق الترحيب الخاص بك في ملف تنفيذي أُضيف إلى مجلدك الحالي. تحقق من ذلك عن طريق تشغيل الأمر التالي:

```
$ ls
```

إذا كنت تستخدم نظام التشغيل ماك أو إس macOS أو لينكس Linux، فستجد ملفًا تنفيذيًا جديدًا مُسمّى على اسم المجلد الذي بنيت فيه برنامجك:

```
greeter main.go go.mod
```

> في نظام التشغيل ويندوز، سيكون الملف التنفيذي باسم greeter.exe.

سيُنشئ الأمر `go build` افتراضيًا ملفًا تنفيذيًا للنظام الأساسي والمعمارية الحاليين. على سبيل المثال، إذا بُنيَ على نظام تشغيل Linux/386، سيكون الملف التنفيذي متوافقًا مع أي نظام Linux/386 آخر، حتى إذا لم يكن جو مُثبّتًا على ذلك النظام. تدعم لغة جو إمكانية البناء على الأنظمة والمعماريات الأخرى، ويمكنك قراءة المزيد عن ذلك في فصل بناء تطبيقات جو على أنظمة التشغيل والمعماريات المختلفة.

بعد أن أنشأت ملفك التنفيذي، يمكنك تشغيله للتأكد من أنه قد بُنيَ بطريقة سليمة.

في نظام ماك أو إس أو لينكس، شغّل الأمر التالي:

```
$ ./greeter
```

أما في ويندوز فنفّذ الأمر التالي:

```
$ greeter.exe
```

سيكون الخرج على النحو التالي:

```
Hello, World!
```

بذلك تكون قد أنشأت ملفًا تنفيذيًّا يحتوي على برنامجك وعلى شيفرة النظام المطلوبة لتشغيل هذا الملف التنفيذي. يمكنك الآن توزيع هذا البرنامج على أنظمة جديدة أو نشره على خادم، مع العلم أن الملف سيعمل دائمًا على نفس البرنامج.

سنشرح فيما يلي كيفية تسمية ملف تنفيذي وتعديله بحيث يمكنك التحكم أكثر في عملية بناء البرنامج.

24.5 تغيير اسم الملف التنفيذي

بعد أن عرفت كيفية إنشاء ملف تنفيذي، ستكون الخطوة التالية هي تحديد كيفية اختيار جو اسمًا للملف التنفيذي وكيفية تخصيص هذا الاسم لمشروعك.

يقرر جو تلقائيًا اسم الملف التنفيذي الذي بُنيَ عند تشغيل الأمر `go build`، وذلك اعتمادًا على الوحدة التي أنشأتها. عندما نفّذنا الأمر `go mod init greeter` منذ قليل، أُنشئت وحدة باسم `greeter`، وهذا هو سبب تسمية الملف التنفيذي الثنائي binary الذي أُنشئ باسم greeter بدوره.

إذا فتحت ملف go.mod (الذي يُفترض أن يكون ضمن مجلد مشروعك)، وكان يتضمن التصريح التالي:

```
module github.com/sammy/shark
```

وهذا يعني أن الاسم الافتراضي للملف التنفيذي الذي بُنيَ هو shark.

لن تكون هذه التسميات الافتراضية دائمًا الخيار الأفضل لتسمية الملف التنفيذي الخاص بك في البرامج الأكثر تعقيدًا التي تتطلب تسميات اصطلاحية محددة، وسيكون من الأفضل تحديد أسماء مُخصصة من خلال الراية -o.

سنغيّر الآن اسم الملف التنفيذي الذي أنشأناه في القسم السابق إلى الاسم hello ونضعه في مجلد فرعي يسمى bin. ولن نحتاج إلى إنشاء هذا المجلد إذ ستتكفل جو بذلك أثناء عملية البناء. نفّذ الأمر go build مع الراية -o كما يلي:

```
$ go build -o bin/hello
```

تُخبر الراية -o مُصرّف جو أن عليه مُطابقة خرج الأمر go build مع الوسيط المُحدد بعدها والمتمثّل بالعبارة bin/hello. بعبارةٍ أوضح؛ تُخبر هذه الراية المُصرّف أن الملف التنفيذي السابق يجب أن يكون اسمه hello وأن يكون ضمن مجلد اسمه bin، وفي حال لم يكن هذا المجلد موجودًا فعليك إنشاؤه تلقائيًّا.

لاختبار الملف التنفيذي الجديد، انتقل إلى المجلد الجديد وشغّل الملف التنفيذي:

```
$ cd bin
$ ./hello
```

ستحصل على الخرج التالي:

```
Hello, World!
```

يمكنك الآن اختيار اسم الملف التنفيذي ليناسب احتياجات مشروعك.

إلى الآن لا تزال مقيدًا بتشغيل ملفك التنفيذي من المجلد الحالي، وإذا أردت استخدام الملفات التنفيذية التي بنيتها من أي مكان على نظامك، يجب عليك تثبيتها باستخدام الأمر go install.

24.6 تثبيت برامج جو باستخدام الأمر go install

ناقشنا حتى الآن كيفية إنشاء ملفات تنفيذية من ملفات مصدرية بامتداد go.، هذه الملفات التنفيذية مفيدة من أجل التوزيع والنشر والاختبار، ولكن لا يمكن تنفيذها من خارج المجلدات المصدرية الموجودة ضمنها. قد تكون هذه مشكلة إذا كنت تريد استخدام برنامجك باستمرار ضمن سكريبتات الصدفة shell scripts أو في

مهام أخرى. لتسهيل استخدام البرامج، يمكنك تثبيتها في نظامك والوصول إليها من أي مكـان. لتوضـيح الفكـرة سنستخدم الأمر `go install` لتثبيت البرنامج الذي نعمل عليه.

يعمل الأمر `go install` على نحوٍ مماثل تقريبًا للأمر `go build`، ولكن بدلًا من ترك الملـف التنفيـذي في المجلد الحالي أو مجلد محدد بواسطة الراية `-o`، فإنه يضعه في المجلد `$GOPATH/bin`.

لمعرفة مكان وجود مجلد `$GOPATH` الخاص بك، شغّل الأمر التالي:

```
$ go env GOPATH
```

قد يختلف الخرج الذي تتلقاه، ولكن يُفترض أن يكون ضمن مجلد `go` الموجود داخل مجلد `$HOME`:

```
$HOME/go
```

نظرًا لأن `go install` سيضع الملفات التنفيذية التي أُنشئت في مجلد فرعي للمجلد `$GOPATH` اسـمه bin، يجب إضافة هذا المجلد إلى متغير البيئة `$PATH`. تحدّثنا عن هذه المواضيع في الفصل الأول من الكتاب.

بعد إعداد المجلد `$GOPATH/bin`، ارجع إلى مجلد `greeter`:

```
$ cd ..
```

شغّل الآن أمر التثبيت:

```
$ go install
```

سيؤدي هذا إلى بناء ملفك التنفيذي ووضعه في `$GOPATH/bin`. شغّل الأمر التالي لاختبار ذلك:

```
$ ls $GOPATH/bin
```

سيسرد لك هذا الأمر محتويات المجلد `$GOPATH/bin`:

```
greeter
```

> لا يدعم الأمر `go install` الراية `-o`، لذلك سيستخدم الاسم الافتراضي الذي تحدّثنا عنه سابقًا لتسمية الملف التنفيذي.

تحقق الآن ما إذا كان البرنامج سيعمل من خارج المجلد المصدر. ارجع أولًا إلى المجلد `$HOME`:

```
$ cd $HOME
```

استخدم ما يلي لتشغيل البرنامج:

```
$ greeter
```

ستحصل على الخرج التالي:

```
Hello, World!
```

يمكنك الآن تثبيت البرامج التي تكتبها في نظامك، مما يتيح لك استخدامها من أي مكان، ومتى احتجت إليها.

24.7 خاتمة

أوضحنا في هذا الفصل كيف تسهل سلسلة أدوات جو عملية إنشاء ملفات تنفيذية ثنائية من التعليمات البرمجية المصدرية، ويمكن توزيعها لتعمل على أنظمة أخرى، حتى لو لم تحتوي على أدوات وبيئات للغة جو.

استخدمنا أيضًا الأمر `go install` لبناء برامجنا وتثبيتها تلقائيًا مثل ملفات تنفيذية ضمن متغير البيئة `$PATH` الخاص بالنظام. ستستطيع من خلال الأمرين `go install` و `go build` مشاركة واستخدام التطبيق الخاص بك كما تشاء.

الآن بعد أن تعرفت على أساسيات `go build`، يمكنك استكشاف كيفية إنشاء شيفرة مصدر معيارية من خلال الفصل التالي من الكتاب الذي يشرح طريقة استخدام وسوم البنية Struct Tags من أجل تخصيص الملفات التنفيذية في لغة جو Go.

دورة علوم الحاسوب

مميزات الدورة

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

اشترك الآن

25. استخدام وسوم البنية Struct Tags

تُستخدم البنى structs لجمع أجزاء متعددة من المعلومات معًا ضمن كتلة واحدة، وتُستخدم مجموعات المعلومات هذه لوصف المفاهيم ذات المستوى الأعلى، مثل العنوان المكوَّن من شارع ومدينة وولاية ورمز بريدي. عندما تقرأ هذه المعلومات من أنظمة، مثل قواعد البيانات، أو واجهات برمجة التطبيقات API، يمكنك استخدام وسوم البنية للتحكم في كيفية تخصيص هذه المعلومات لحقول البنية. وسوم البنية هي أجزاء صغيرة من البيانات الوصفية المرفقة بحقول البنية التي توفر إرشادات إلى شيفرة جو أُخرى تعمل مع البنية.

25.1 كيف يبدو شكل وسم البنية؟

وسم البنية في لغة جو هو عبارة عن "توضيح" يُكتب بعد نوع الحقل داخل البنية، ويتكون كل وسم من زوج `key:"value"` أي مفتاح مرتبط بقيمة مقابلة ويوضع ضمن علامتين اقتباس مائلة (`) كما يلي:

```
type User struct {
  Name string `example:"name"`
}
```

من خلال هذا التعريف، ستكون هناك شيفرة جو أخرى قادرة على فحص هذه البنية واستخراج القيم المخصصة لمفاتيح معينة، وبدون هذه الشيفرة الأخرى لن تؤثر وسوم البنية على تعليماتك البرمجية.

جرّب هذا المثال لترى كيف يبدو وسم البنية، وكيف سيكون عديم التأثير دون الشيفرة الأخرى.

```
package main
import "fmt"
type User struct {
  Name string `example:"name"`
```

```
}
func (u *User) String() string {
  return fmt.Sprintf("Hi! My name is %s", u.Name)
}
func main() {
  u := &User{
        Name: "Sammy",
  }
  fmt.Println(u)
}
```

سيكون الخرج على النحو التالي:

```
Hi! My name is Sammy
```

يعرّف هـذا المثـال نـوع بيانـات باسـم `User` يمتلـك حقلًا اسـمه `Name`، وأعطينـا هـذا الحقـل وسـم بنيـة `example:"name"` مُتمثّل بالمفتاح `example` والقيمة `"name"` للحقل `Name`. عرّفنـا التـابع `String()` في البنية `User`، والذي تحتاجه الواجهة `fmt.Stringer`، وبالتالي سيُستدعى تلقائيًـا عنـدما نُمـرّر هـذا النـوع إلى `fmt.Println` وبالتالي نحصل على طباعة مُرتبة للبنية.

نأخذ في الدالة `main` متغيرًا من النوع `User` ونمرّره إلى دالة الطباعـة `fmt.Println`. على الـرغم من أنـه لدينا وسم بنية، إلا أننا لن نلحظ أي فرق في الخرج، أي كما لـو أنهـا غـير موجـودة. لكي نحصـل على تـأثير وسـم البنية، يجب كتابة شيفرة أخرى تفحص البنية في وقت التشغيل runtime. تحتوي المكتبة القياسـية على حـزم تستخدم وسوم البنية كأنها جزء من عملها، وأكثرها شيوعًا هي حزمة `encoding/json`.

25.2 ترميز JSON

جسون JSON هي اختصار إلى "ترميز كـائن باسـتخدام جافـا سـكريبت JavaScript Object Notation"، وهي تنسيق نصي لترميز مجموعات البيانات المنظمة وفق أسماء مفاتيح مختلفة. يُشاع استخدام جسون لربط البيانات بين البرامج المختلفة، إذ أن التنسيق بسيط ويوجد مكتبات جاهزة لفك ترمـيزه في العديـد من اللغـات البرمجية. فيما يلي مثال على جسون:

```
{
 "language": "Go",
 "mascot": "Gopher"
}
```

يتضمن كائن جسون أعلاه مفتاحين؛ الأول هو `language` والثاني `mascot`، ولكل منهما قيمة مرتبطة به هما `Go` و `Gopher` على التوالي.

يستخدم مُرمّز encoder جسون في المكتبة القياسية وسوم البنية مثل "توصيفات annotations" تشير إلى الكيفية التي تريد بها تسمية الحقول الخاصة بك في خرج جسون. يمكن العثور على آليات ترميز وفك ترميز جسون في حزمة `encoding/json` من هنا.

جرب هذا المثال لترى كيف يكون ترميز جسون دون وسوم البنية:

```
package main
import (
  "encoding/json"
  "fmt"
  "log"
  "os"
  "time"
)
type User struct {
  Name     string
  Password   string
  PreferredFish []string
  CreatedAt   time.Time
}
func main() {
  u := &User{
        Name:   "Sammy the Shark",
        Password: "fisharegreat",
        CreatedAt: time.Now(),
  }
  out, err := json.MarshalIndent(u, "", " ")
  if err != nil {
        log.Println(err)
        os.Exit(1)
  }
  fmt.Println(string(out))
}
```

وسيكون الخرج على النحو التالي:

```
{
 "Name": "Sammy the Shark",
 "Password": "fisharegreat",
 "CreatedAt": "2019-09-23T15:50:01.203059-04:00"
}
```

عرّفنا بنيـة تمثّـل مُسـتخدم مـع حقـول تُـدل على اسـمه `Name` وكلمـة المـرور `Password` وتـاريخ إنشـاء الحساب `CreatedAt`. وأخـذنا داخـل الدالـة `main` متغـيرًا من هـذه البنيـة وأعطينـا قيمًـا لجميـع حقولـه عـدا `PreferredFish`. بعد ذلك، مرّرنا البنية إلى الدالة `json.MarshalIndent`، إذ تمكننا هذه الدالـة من رؤيـة خرج جسون بسهولة ودون استخدام أي أداة خارجية. يمكن استبدال الاستدعاء السابق لهـذه الدالـة بالاسـتدعاء `json.Marshal(u)` وذلك لطباعة جسون بدون أي فراغـات إضـافية، إذ يتحكم الوسـيطان الإضـافيان للدالـة `json.MarshalIndent` في بادئة الخرج، التي أهملناها مع السلسلة الفارغة، وبالمحـارف المُـراد اسـتخدامها من أجل تمثيل المسافة البادئة (هنا فراغين).

سُجّلت الأخطاء الناتجـة عن `json.MarshalIndent` وأُنهي البرنـامج باسـتخدام `os.Exit(1)`، وأخـيرًا حوّلنـا المصـفوفة `[]byte` المُعـادة من `json.MarshalIndent` إلى سلسـلة ومررنـا السلسـلة الناتجـة إلى `fmt.Println` للطباعة على الطرفية.

تظهر حقول البنية تمامًا كما تُسمّى، وهذا ليس طبعًا نمط جسون النموذجي الذي يسـتخدم تنسـيق "سـنام الجمل Camel case" لأسماء الحقول. سنحقق ذلك في المثال التالي، إذ سنرى عند تشغيله أنه لن يعمـل لأن أسماء الحقول المطلوبة تتعارض مع قواعد جو المتعلقة بأسماء الحقول المصدَّرة.

```
package main
import (
  "encoding/json"
  "fmt"
  "log"
  "os"
  "time"
)
type User struct {
  name      string
  password   string
  preferredFish []string
  createdAt   time.Time
}
```

```
func main() {
  u := &User{
        name:   "Sammy the Shark",
        password: "fisharegreat",
        createdAt: time.Now(),
  }
  out, err := json.MarshalIndent(u, "", " ")
  if err != nil {
        log.Println(err)
        os.Exit(1)
  }

  fmt.Println(string(out))
}
```

وسيكون الخرج على النحو التالي:

```
{}
```

استبدلنا هذه المرة أسماء الحقول، بحيث تتوافق مع تنسيق سنام الجمل؛ فبدلًا من `Name` وضعنا `name` وبدلًا من `Password` وضعنا `password` وبدلًا من `CreatedAt` وضعنا `createdAt`، وعدّلنا داخل متن الدالة `main` أسماء الحقول أيضًا بحيث تتوافق مع الأسماء الجديدة، ومرّرنا البنية إلى الدالة `json.MarshalIndent` كما في السابق. الخرج كان عبارة عن كائن جسون فارغ `{}`.

يفرض نمط سنام الجمل أن يكون أول حرف صغير دومًا، بينما لا تهتم جسون بذلك، ولغة جو صارمة مع حالة الأحرف؛ إذ تدل البداية بحرف كبير على أن الحقل غير مُصدّر وتدل البداية بحرف صغير على أنّه مُصدّر، وبما أن الحزمة `encoding/json` هي حزمة منفصلة عن حزمة `main` التي نستخدمها، يجب علينا كتابة الحرف الأول بأحرف كبيرة لجعله مرئيًا للحزمة `encoding/json`. حسنًا، يبدو أننا في طريق مسدود، فنحن بحاجة إلى طريقة ما لننقل إلى ترميز جسون ما نود تسمية هذا الحقل به.

25.2.1 استخدام وسوم البنية للتحكم بالترميز

يمكنك تعديل المثال السابق، بحيث تكون قادرًا على تصدير الحقول المتوافقة مع تنسيق سنام الجمل عن طريق إضافة وسم بنية لكل حقل. يجب أن يكون لدى وسوم البنية التي تتعرّف عليها الحزمة `encoding/json` مفتاحها `json` مع قيمة مُرافقة لهذا المفتاح تتحكم بالخرج.

إذًا، من خلال استخدام تنسيق سنام الجمل لقيم المفاتيح، سيستخدم المرمّز هذه القيم مثل أسماء للحقول مع الإبقاء على الحقول مُصدّرة، وبالتالي نكون أنهينا المشاكل السابقة:

```
package main
import (
  "encoding/json"
  "fmt"
  "log"
  "os"
  "time"
)
type User struct {
  Name      string   `json:"name"`
  Password   string   `json:"password"`
  PreferredFish []string `json:"preferredFish"`
  CreatedAt   time.Time `json:"createdAt"`
}
func main() {
  u := &User{
        Name:   "Sammy the Shark",
        Password: "fisharegreat",
        CreatedAt: time.Now(),
  }
  out, err := json.MarshalIndent(u, "", " ")
  if err != nil {
        log.Println(err)
        os.Exit(1)
  }
  fmt.Println(string(out))
}
```

سيكون الخرج على النحو التالي:

```
{
 "name": "Sammy the Shark",
 "password": "fisharegreat",
 "preferredFish": null,
 "createdAt": "2019-09-23T18:16:17.57739-04:00"
}
```

لاحظ أننا تركنا أسماء الحقول تبدأ بأحرف كبيرة من أجل السماح بعملية التصدير، ولحل مشكلة تنسيق سنام الجمل استخدمنا وسوم بنية من الشكل `json:"name"`، إذ أن `"name"` هي القيمة التي نريد من `json.MarshalIndent` أن تعرضها عند طباعة كائن جسون.

لاحظ أننا لم نُعطِ الحقل `PreferredFish` أي قيمة، وبالتالي قد لا نرغب بظهوره عند طباعة كائن جسون. فيما يلي سنتحدث حول هذا الموضوع.

25.2.2 حذف حقول جسون الفارغة

يُعد حذف حقول الخرج التي ليس لها قيمة في جسون أمرًا شائعًا، وبما أن جميع الأنواع في جو لها "قيمة صفرية" أو قيمة افتراضية مُهيّأة بها، تحتاج حزمة `encoding/json` إلى معلومات إضافية لتتمكن من معرفة أن بعض الحقول ينبغي عدّها غير مضبوطة، أي قيمتها صفرية في جو. هذه المعلومات هي في الحقيقة مجرد كلمة واحدة نضيفها إلى نهاية القيمة المرتبطة بمفتاح الحقل في وسم البنية؛ وهذه الكلمة هي `,omitempty`، إذ نُخبر جسون من خلال إضافة هذه الكلمة إلى الحقل بأننا لا نريد ظهوره عندما تكون قيمته صفرية. يوضح المثال التالي الأمر:

```
package main
import (
  "encoding/json"
  "fmt"
  "log"
  "os"
  "time"
)
type User struct {
  Name     string  `json:"name"`
  Password   string  `json:"password"`
  PreferredFish []string `json:"preferredFish,omitempty"`
  CreatedAt   time.Time `json:"createdAt"`
}
func main() {
  u := &User{
        Name:   "Sammy the Shark",
        Password: "fisharegreat",
        CreatedAt: time.Now(),
  }
```

```
    out, err := json.MarshalIndent(u, "", " ")
    if err != nil {
          log.Println(err)
          os.Exit(1)
    }
    fmt.Println(string(out))
}
```

سيكون الخرج على النحو التالي:

```
{
 "name": "Sammy the Shark",
 "password": "fisharegreat",
 "createdAt": "2019-09-23T18:21:53.863846-04:00"
}
```

عـــدّلنا الأمثلـــة الســـابقة بحيث أصـــبح حقـــل PreferredFish يحتـــوي على وســـم البنيـــة "json:"preferredFish,omitempty، إذ سيمنع وجود الكلمة ,omitempty ظهور هذا الحقل في خرج كائن جسون.

أصبحت الآن الأمور أفضل، لكن هناك مشكلة أخرى واضحة، وهي ظهور كلمة المرور، ولحل المشكلة تؤمن encoding/jon طريقةً لتجاهل الحقول الخاصة تمامًا.

25.2.3 منع عرض الحقول الخاصة في خرج كائنات جسون

يجب تصدير بعض الحقول من البنى حتى تتمكن الحزم الأخرى من التفاعل بطريقة صحيحة مع النـوع، لكن قد تكون المشكلة في حساسية طبيعة أحد هذه الحقول كما في حالة كلمة المرور في المثال السابق، لذا نـود أن يتجاهل مُرمّز جسون الحقل تمامًا، حتى عند تهيئته بقيمة. يكون حل هذه المشكلة باستخدام المحـرف - ليكـون قيمةً لوسيط المفتاح الخاص بوسم البنية json:.

يعمل هذا المثال على إصلاح مشكلة عرض كلمة مرور المستخدم.

```
package main
import (
  "encoding/json"
  "fmt"
  "log"
  "os"
```

```
    "time"
)
type User struct {
    Name   string  `json:"name"`
    Password string  `json:"-"`
    CreatedAt time.Time `json:"createdAt"`
}
func main() {
    u := &User{
          Name:   "Sammy the Shark",
          Password: "fisharegreat",
          CreatedAt: time.Now(),
    }
    out, err := json.MarshalIndent(u, "", " ")
    if err != nil {
          log.Println(err)
          os.Exit(1)
    }
    fmt.Println(string(out))
}
```

سيكون الخرج على النحو التالي:

```
{
 "name": "Sammy the Shark",
 "createdAt": "2019-09-23T16:08:21.124481-04:00"
}
```

الشيء الوحيد الذي تغير في هذا المثال عن المثال السابق هو أن حقل كلمة المرور يستخدم الآن القيمة الخاصة `"-"` لوسم البنية `json:`. يمكنك أن تلاحظ من الخرج السابق اختفاء كلمة المرور من الخرج.

ميزتا التجاهل والإخفاء، أو حتى باقي الخيارات في حزمة `encoding/json` التي استخدمناها مع حقل `PreferredFish` و `Password`، ليستا قياسيتين، أي ليست كل الحزم تستخدم نفس الميزات ونفس بنية القواعد، لكن حزمة `encoding/json` هي حزمة مُضمّنة عمومًا في المكتبة القياسية، وبالتالي سيكون لدى الحزم الأخرى نفس الميزات ونفس العرض convention. مع ذلك، من المهم قراءة التوثيق الخاص بأي حزمة تابعة لجهة خارجية تستخدم وسوم البنية لمعرفة ما هو مدعوم وما هو غير مدعوم.

25.3 خاتمة

توفر وسوم البنية وسيلةً قويةً لتسهيل التعامل مع دوال الشيفرات التي تعمل مع البـنى، كمـا تـوفر العديـد من الحزم القياسية والخارجية طرقًا لتخصيص عملياتها من خلال استخدام هذه الوسوم. يـوفر اسـتخدام الوسـوم بفعالية في التعليمات البرمجية الخاصة بك سلوك تخصيص ممتاز ويوثّق بإيجاز كيفيـة اسـتخدام هـذه الحقـول للمطوّرين المستقبليين.

أكبر موقع توظيف عن بعد في العالم العربي

ابحث عن الوظيفة التي تحقق أهدافك وطموحاتك
المهنية في أكبر موقع توظيف عن بعد

تصفح الوظائف الآن

26. استخدام الواجهات Interfaces

من أهم الصفات التي يجب أن تتمتع بها البرامج التي نكتبها، هي المرونة وإمكانية إعادة الاستخدام، إضافةً إلى الصفة التركيبية modular، إذ تُعد هذه الصفات الثلاثة أمرًا ضروريًّا لتطوير برامج متعددة الاستخدامات، كما أنها تُسهّل عمليات التعديل والصيانة على البرامج، فمثلًا إذا احتجنا لتعديل بسيط على البرنامج، سيكون بالإمكان إجراء هذا التعديل في مكانٍ واحد بدلًا من إجراء نفس التعديل في أماكن متعددة من البرنامج.

تختلف كيفية تحقيق تلك الصفات من لغة إلى أخرى، فعلى سبيل المثال، يُستخدم مفهوم الوراثة في لغات مثل جافا Java و C++ و C# لتحقيق ذلك. يمكن للمطورين أيضًا تحقيق أهداف التصميم هذه من خلال التركيب Composition، وهي طريقة لدمج الكائنات أو أنواع البيانات في أنواع أكثر تعقيدًا، وهو النهج المُستخدم في لغة جو لتحقيق الصفات السابقة. توفِّر الواجهات في لغة جو توابعًا لتنظيم التراكيب المعقدة، وسيتيح لك تعلم كيفية استخدامها إنشاء شيفرة مشتركة وقابلة لإعادة الاستخدام.

ستتعلم في هذا الفصل تركيب أنواع مخصصة لها سلوكيات مشتركة، وإعادة استخدام الشيفرة التي نكتبها، وسنتعلم كيفية تحقيق الواجهات للأنواع المخصصة التي تتوافق مع الواجهات المُعرّفة في حزم أخرى.

26.1 تعريف السلوك Behavior

تُعد الواجهات من العناصر الأساسية للتركيب، فهي تعرّف سلوك نوعٍ ما، وتُعد الواجهة `fmt.Stringer` من أكثر الواجهات شيوعًا في مكتبة جو القياسية:

```
type Stringer interface {
  String() string
}
```

نُعرّف في السطر الأول من الشيفرة السابقة نوعًا جديدًا يُدعى `Stringer`، ونحدد أنه واجهة. بعد ذلك، نكتب محتويات هذه البنية بين قوسين `{}`، إذ ستُعرِّف هذه المحتويات سلوك الواجهة، أي ما الذي تفعله الواجهة؛ فبالنسبة للواجهة السابقة `Stringer`، من الواضح أن السلوك الوحيد فيها هو تابع `String()` لا يأخذ أي وسطاء ويعيد سلسلةً نصية.

سنرى الآن بعض المقتطفات البرمجية التي تمتلك سلوك الواجهة `fmt.Stringer`:

```
package main
import "fmt"
type Article struct {
  Title string
  Author string
}
func (a Article) String() string {
  return fmt.Sprintf("The %q article was written by %s.", a.Title,
a.Author)
}
func main() {
  a := Article{
        Title: "Understanding Interfaces in Go",
        Author: "Sammy Shark",
  }
  fmt.Println(a.String())
}
```

هنا نُنشئ نوعًا جديدًا اسمه `Article`، ويمتلك حقلين هما `Title` و `Author`، وكلاهما من نوع سلاسل نصية.

```
...
type Article struct {
  Title string
  Author string
}
…
```

نُعرّف بعد ذلك تابعًا يسمى `String` على النوع `Article`، بحيث يعيد هذا التابع سلسلةً تمثل هذا النوع، أي محتوياته عمليًّا:

```
...
func (a Article) String() string {
  return fmt.Sprintf("The %q article was written by %s.", a.Title,
a.Author)
}
...
```

نُعـرّف بعـد ذلـك في الدالـة `main` متغـيرًا من النـوع `Article` ونسـميه `a`، ونسـند السلسـلة "Understanding Interfaces in Go" إلى الحقل `Title` والسلسلة "Sammy Shark" للحقل `Author`:

```
...
a := Article{
  Title: "Understanding Interfaces in Go",
  Author: "Sammy Shark",
}
...
```

نطبع بعد ذلك نتيجة التابع `String` من خلال استدعاء الدالة `fmt.Println` وتمرير نتيجة استدعاء التابع `a.String()` إليها:

```
...
fmt.Println(a.String())
```

ستحصل عند تشغيل البرنامج على:

```
The "Understanding Interfaces in Go" article was written by Sammy
Shark.
```

لم نستخدم واجهةً حتى الآن، لكننا أنشأنا نوعًا يمتلك سلوكًا يطابق سلوك الواجهـة `fmt.Stringer`. دعنـا نرى كيف يمكننا استخدام هذا السلوك لجعل الشيفرة الخاصة بنا أكثر قابلية لإعادة الاستخدام.

26.2 تعريف الواجهة Interface

بعد أن عرّفنا نوعًا جديدًا مع سلوك مُحدد، سنرى كيف يمكننا استخدام هذا السلوك، لكن قبل ذلك سـنلقي نظرةً على ما سنحتاج إلى فعله إذا أردنا استدعاء التابع `String` من نوع `Article` داخل دالة:

```
package main
import "fmt"
type Article struct {
```

```
  Title string
  Author string
}
func (a Article) String() string {
  return fmt.Sprintf("The %q article was written by %s.", a.Title,
a.Author)
}
func main() {
  a := Article{
        Title: "Understanding Interfaces in Go",
        Author: "Sammy Shark",
  }
  Print(a)
}
func Print(a Article) {
  fmt.Println(a.String())
}
```

أضفنا في هذه الشيفرة دالةً جديدةً تسمى `Print`، تأخذ وسيطًا من النوع `Article`. لاحظ أن كل ما تفعله هذه الدالة هو أنها تستدعي التابع `String`، لذا يمكننا بدلًا من ذلك تعريف واجهة للتمرير إلى الدالة:

```
package main
import "fmt"
type Article struct {
  Title string
  Author string
}
func (a Article) String() string {
  return fmt.Sprintf("The %q article was written by %s.", a.Title,
a.Author)
}
type Stringer interface {
  String() string
}
func main() {
  a := Article{
        Title: "Understanding Interfaces in Go",
```

```
        Author: "Sammy Shark",
    }
    Print(a)
}
func Print(s Stringer) {
    fmt.Println(s.String())
}
```

ننشئ هنا واجهةً اسمها `Stringer`:

```
...
type Stringer interface {
  String() string
}
...
```

تتضمّن هذه الواجهة تابعًا وحيدًا يسمى `String()` يعيد سلسلةً، ويُعرّف هذا التابع على نوع بيانات مُحدد، وعلى عكس الدوال فلا يمكن له أن يُستدعى إلا من متغير من هذا النوع.

نعدّل بعد ذلك بصمة الدالة `Print` بحيث تستقبل وسيطًا من النوع `Stringer` الذي يُمثّل واجهةً، وليس نوعًا مُحددًا مثل `Article`. بما أن المصرّف يعرف أن `Stringer` هي واجهة تمتلك التابع `String`، فلن يقبل إلا الأنواع التي تُحقق هذا التابع.

يمكننا الآن استخدام الدالة `Print` مع أي نوع يتوافق مع الواجهة `Stringer`. لتوضيح ذلك دعنا ننشئ نوعًا آخر كما يلي:

```
package main
import "fmt"
type Article struct {
    Title string
    Author string
}
func (a Article) String() string {
    return fmt.Sprintf("The %q article was written by %s.", a.Title,
a.Author)
}
type Book struct {
    Title string
```

```
    Author string
    Pages int
}
func (b Book) String() string {
    return fmt.Sprintf("The %q book was written by %s.", b.Title,
.Author)
}
type Stringer interface {
    String() string
}
func main() {
    a := Article{
        Title: "Understanding Interfaces in Go",
        Author: "Sammy Shark",
    }
    Print(a)
    b := Book{
        Title: "All About Go",
        Author: "Jenny Dolphin",
        Pages: 25,
    }
    Print(b)
}
func Print(s Stringer) {
    fmt.Println(s.String())
}
```

عرّفنا هنا نوع بيانات جديد يُسمى Book يمتلك التابع String، وبالتالي يُحقق الواجهة Stringer، وبالتالي يمكننا استخدامه مثل وسيط للدالة Print.

```
The "Understanding Interfaces in Go" article was written by Sammy
Shark.
The "All About Go" book was written by Jenny Dolphin. It has 25 pages.
```

أوضحنا إلى الآن كيفية استخدام واجهة واحدة فقط، ويمكن عمومًا أن يكون للواجهة أكثر من سلوك معـرّف.

وسنرى في الفقرات التالية يلي كيف يمكننا جعل واجهاتنا أكثر تنوعًا من خلال التصريح عن المزيد من التوابع.

26.3 تعدد السلوكيات في الواجهة

تُعد كتابة أنواع صغيرة وموجزة وتركيبها في أنواع أكبر وأكثر تعقيدًا من الأمور الجيدة عند كتابة الشيفرات في لغة جو، وينطبق الشيء نفسه عند إنشاء واجهات. لمعرفة كيفية إنشاء الواجهة، سنبدأ أولًا بتعريف واجهة واحدة فقط. سنحدد شكلين؛ دائرة `Circle` ومربع `Square`، وسيُعرّف كلاهما تابعًا يُسمى المساحة `Area` يعيد المساحة الهندسية للشكل:

```
package main
import (
  "fmt"
  "math"
)
type Circle struct {
  Radius float64
}
func (c Circle) Area() float64 {
  return math.Pi * math.Pow(c.Radius, 2)
}
type Square struct {
  Width float64
  Height float64
}
func (s Square) Area() float64 {
  return s.Width * s.Height
}
type Sizer interface {
  Area() float64
}
func main() {
  c := Circle{Radius: 10}
  s := Square{Height: 10, Width: 5}
  l := Less(c, s)
  fmt.Printf("%+v is the smallest\n", l)
}
func Less(s1, s2 Sizer) Sizer {
  if s1.Area() < s2.Area() {
```

```
        return s1
    }
    return s2
}
```

بمـا أن النـوعين يُصـرحان عن تـابع Area، فيمكننـا إنشـاء واجهـة تحـدد هـذا السـلوك. لنُنشـئ واجهـة Sizer التالية:

```
...
type Sizer interface {
  Area() float64
}
…
```

نعرّف بعد ذلك دالةً تسمى Less تأخذ واجهتين Sizer مثل وسيطين وتعيد أصغر واحدة:

```
...
func Less(s1, s2 Sizer) Sizer {
  if s1.Area() < s2.Area() {
   return s1
  }
  return s2
}
...
```

لاحظ أن معاملات الدالة وكذلك القيمة المُعادة هي من النوع Sizer، وهذا يعني أننا لا نعيد مربعًا أو دائرة، بل نعيد واجهة Sizer.

والآن نطبع الوسيط الذي لديه أصغر مساحة:

```
{Width:5 Height:10} is the smallest
```

سنضيف الآن سلوكًا آخر لكل نوع، وسنضيف التـابع String() الـذي يعيـد سلسـلةً نصـيةً، وهـذا بـدوره سيؤدي إلى تحقيق الواجهة fmt.Stringer:

```
package main
import (
  "fmt"
  "math"
```

```
)
type Circle struct {
  Radius float64
}
func (c Circle) Area() float64 {
  return math.Pi * math.Pow(c.Radius, 2)
}
func (c Circle) String() string {
  return fmt.Sprintf("Circle {Radius: %.2f}", c.Radius)
}
type Square struct {
  Width float64
  Height float64
}
func (s Square) Area() float64 {
  return s.Width * s.Height
}
func (s Square) String() string {
  return fmt.Sprintf("Square {Width: %.2f, Height: %.2f}", s.Width,
s.Height)
}
type Sizer interface {
  Area() float64
}
type Shaper interface {
  Sizer
  fmt.Stringer
}
func main() {
  c := Circle{Radius: 10}
  PrintArea(c)
  s := Square{Height: 10, Width: 5}
  PrintArea(s)
  l := Less(c, s)
  fmt.Printf("%v is the smallest\n", l)
}
func Less(s1, s2 Sizer) Sizer {
```

```
    if s1.Area() < s2.Area() {
        return s1
    }
    return s2
}
func PrintArea(s Shaper) {
    fmt.Printf("area of %s is %.2f\n", s.String(), s.Area())
}
```

بما أن النـوعين `Circle` و `Square` ينفـذّان التـابعين `Area` و `String`، سـيكون بالإمكـان إنشـاء واجهـة أخرى لوصف تلك المجموعة الأوسع من السلوك. لأجل ذلك سننشئ واجهةً تسمى `Shaper` من واجهة `Sizer` وواجهة `fmt.Stringer`:

```
...
type Shaper interface {
    Sizer
    fmt.Stringer
}
...
```

> حبذا أن ينتهي اسم الواجهة بالحرفين `er` مثل `fmt.Stringer` و `io.Writer`، إلخ. ولهذا السبب أطلقنا على واجهتنا اسم `Shaper`، وليس `Shape`.

يمكننا الآن إنشاء دالة تسمى `PrintArea` تأخذ وسيطًا من النوع `Shaper`. هذا يعني أنه يمكننـا اسـتدعاء التابعين على القيمة التي تُمرّر لكل من `Area` و `String`:

```
...
func PrintArea(s Shaper) {
    fmt.Printf("area of %s is %.2f\n", s.String(), s.Area())
}
```

ستحصل عند تشغيل البرنامج على الخرج التالي:

```
area of Circle {Radius: 10.00} is 314.16
area of Square {Width: 5.00, Height: 10.00} is 50.00
Square {Width: 5.00, Height: 10.00} is the smallest
```

رأينا كيف يمكننا إنشاء واجهات أصغر وبناء واجهات أكبر حسب الحاجة. وكان بإمكاننا البدء بالواجهـة الأكـبر وتمريرها إلى جميع الدوال، إلا أن الممارسات الجيدة تقتضي إرسال أصغر واجهة فقط إلى الدالة المطلوبة، إذ أن

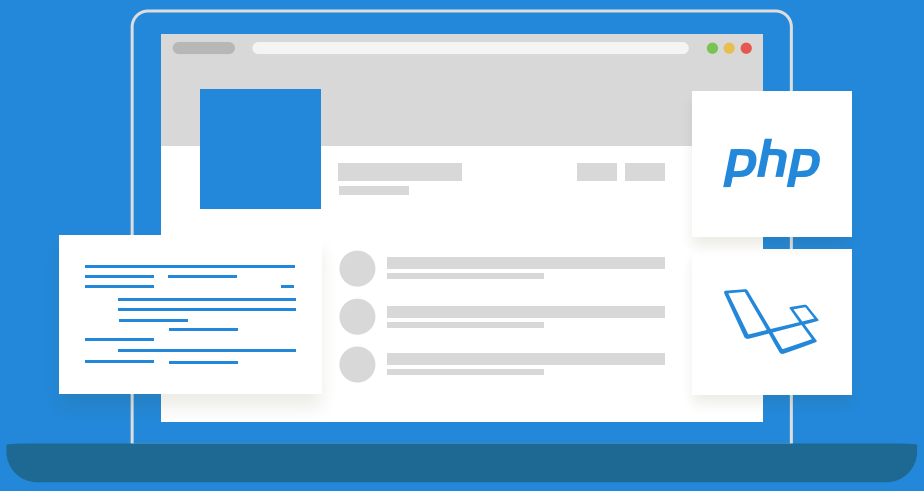
ذلك ضروري لجعل الشيفرة واضحة أكثر، فإذا كانت الدالة تهدف لإجراء سلوك محدد، وهذا السلوك مُعرّف في واجهة أصغر، فحبذا أن تُمرر الواجهة الأصغر وليس الواجهة الأكبر (التي تحتوي الواجهة الأصغر).

إذا مررنا مثلًا الواجهة `Shaper` إلى الدالة `Less`، فهنا نفترض أنها ستستدعي كلًا من التابع `Area` والتابع `String`، لكنها لا تستدعي إلا التابع `Area`، وهذا سيجعل الدالة أقل وضوحًا، كما أننا نعلم أنه يمكننا فقط استدعاء التابع `Area` لأي وسيط يُمرّر إليه.

26.4 خاتمة

تعلمنا في هذا الفصل كيفية إنشاء واجهات صغيرة وكيفية توسيعها لتصبح واجهات أكبر، وكيفية مشاركة الأشياء التي نريدها فقط مع دالة أو تابع من خلال تلك الواجهات. تعلمنا أيضًا كيفية تركيب واجهة من واجهات أصغر أو من واجهات موجودة في حزم أخرى وليس فقط الحزمة التي نعمل ضمنها.

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

27. بناء تطبيقات لمختلف أنظمة التشغيل

عند تطوير البرمجيات من المهم الأخذ بالحسبان نوع نظام التشغيل الـذي تبـني التطـبيق عليـه والمعماريـة التي ستُصرّف تطبيقك من أجلها إلى ملف ثنائي تنفيـذي Binary، وتكـون عمليـة تشـغيل التطـبيق على نظـام تشغيل مختلف أو معمارية مختلفة غالبًا عمليةً بطيئة أو مستحيلة أحيانًا، لذا من الممارسـات العمليّـة الشـائعة بناء ملف تنفيذي يعمل على العديد من المنصات المختلفة، وذلك لزيادة شعبية وعدد مستخدمي تطبيقـك، إلا أن ذلك غالبًا ما يكون صعبًا عندما تختلف المنصة التي تطوّر تطبيقك عليها عن المنصة التي تريد نشـره عليهـا؛ إذ كان يتطلب مثلًا تطوير برنامج على ويندوز ونشره على لينكس Linux أو ماك أو إس MacOS سابقًا إعـدادات بناء مُحددة من أجل كل بيئة تُريد تصريف البرنامج من أجلها، ويتطلب الأمر أيضًا الحفـاظ على مزامنـة الأدوات، إضافةً إلى الاعتبارات الأخرى التي قد تضيف تكلفةً وتجعل الاختبار التعاوني Collaborative Testing والنشـر أكثر صعوبة.

تحل لغة جو هذه المشكلة عن طريق بناء دعم لمنصّات متعددة مباشرةً من خلال الأداة `go build` وبقيـة أدوات اللغة. يمكنك باستخدام متغيرات البيئة ووسوم البنيـة التحكم في نظـام التشـغيل والمعماريـة الـتي بُـني الملف التنفيذي النهائي من أجلها، إضافةً إلى وضع مُخطط لسير العمل يمكن من خلاله التبديل إلى التعليمـات البرمجيـة المُضـمّنة والمتوافقـة مـع المنصـة المطلـوب تشـغيل التطـبيق عليهـا دون تغيـير الشـيفرة البرمجية الأساسية.

ستُنشئ في هذا الفصـل تطبيقًـا يربـط السلاسـل النصـية مـع بعضـها في مسـار ملـف، إضـافةً إلى كتابـة شيفرات برمجية يمكن تضمينها اختياريًّا، يعتمد كلٌ منها على منصة مُحددة. ستُنشـئ ملفـات تنفيذيـة لأنظمـة تشغيل ومعماريات مختلفة على نظامك الخاص، وسيُبيّن لك ذلك كم أن لغة جو قوية في هذا الجانب.

> في تكنولوجيا المعلومات، المنصة هي أي عتاد Hardware أو برمجية Software تُستخدم لاستضافة تطبيق أو خدمة. على سبيل المثال قد تتكون من عتاديات ونظام تشغيل وبرامج أخرى تستخدم مجموعة التعليمات الخاصة بالمعالج.

27.1 المتطلبات

- يفترض هذا الفصل أنك على دراية بوسوم البنية في لغة جو، وإذا لم يكن لديك معرفةً بها، راجع فصل استخدام وسوم البنية لتخصيص الملفات التنفيذية Binaries.
- ستحتاج أيضًا إلى امتلاك مساحة عمل خاصة مُهيئة وجاهزة في لغة جو Go، فإذا لم تكن قد أنشأت واحدة، فارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبَّت لغة جو Go وقم بإعداد بيئة تطوير محلية بحسب نظام تشغيلك.

27.2 المنصات التي يمكن أن تبني لها تطبيقك

قبل أن نعرض كيف يمكننا التحكم بعملية البناء من أجل بناء ملفات تنفيذية تتوافق مع منصات مختلفة، سنستعرض أولًا أنواع المنصات التي يمكن لجو البناء من أجلها، وكيف تشير جو إلى هذه المنصات باستخدام متغيرات البيئة GOOS و GOARCH.

يمكن عرض قائمة بأسماء المنصات التي يمكن لجو أن تبني تطبيقًا من أجلها، وتختلف هذه القائمة من إصدار لآخر، لذا قد تكون القائمة التي سنعرضها عليك مختلفةً عن القائمة التي تظهر عندك وذلك تبعًا لإصدار جو الذي تعمل عليه (الإصدار الحالي 1.13).

نفّذ الأمر التالي لعرض القائمة:

```
$ go tool dist list
```

ستحصل على الخرج التالي:

```
aix/ppc64    freebsd/amd64  linux/mipsle  openbsd/386
android/386   freebsd/arm   linux/ppc64  openbsd/amd64
android/amd64  illumos/amd64  linux/ppc64le openbsd/arm
android/arm   js/wasm     linux/s390x  openbsd/arm64
android/arm64  linux/386    nacl/386    plan9/386
darwin/386    linux/amd64   nacl/amd64p32 plan9/amd64
darwin/amd64   linux/arm    nacl/arm    plan9/arm
darwin/arm    linux/arm64   netbsd/386   solaris/amd64
darwin/arm64   linux/mips   netbsd/amd64  windows/386
```

```
dragonfly/amd64 linux/mips64  netbsd/arm   windows/amd64
freebsd/386   linux/mips64le netbsd/arm64  windows/arm
```

نلاحظ أن الخرج هو مجموعة من أزواج المفتاح والقيمة key-value مفصولة بالرمز / إذ يمثّل المفتـاح key (على اليسار) نظام التشغيل. تُعد هذه الأنظمة قيمًا محتملة لمتغير البيئة GOOS (يُنطق "goose")، اختصارًا إلى Go Operating System؛ بينمـا تشـير القيمـة value (على اليمين) إلى المعماريـة، وتمثّـل القيم المُحتملـة لمتغير البيئة GOARCH (تُنطق "gore-ch") وهي اختصار Go Architecture.

دعنا نأخذ مثال linux/386 لفهم ما تعنيـه وكيـف يجـري الأمـر: تُمثّـل linux المفتـاح وهي قيمـة المتغـير GOOS، بينما تمثّل 386 المعالج Intel 80386 والتي ستكون هي القيمة، وتُمثّل قيمة المتغير GOARCH.

هناك العديد من المنصات الـتي يمكن أن تتعامـل معهـا من خلال الأداة go build، لكن غالبًـا سـيكون تعاملـك مـع منصـة لينكس Linux أو ويـنـدوز Windows أو داروين darwin في قيم المتغـير GOOS. إذًا، هـذا يُغطي المنصات الثلاثة الأكبر: ويندوز ولينكس وماك، إذ يعتمد الأخير على نظـام التشـغيل داروين. ويمكن للغـة جو عمومًا تغطية المنصات الأقل شهرة مثل nacl.

عند تشغيل أمر مثـل go build، يسـتخدم جـو مُتغـيرات البيئـة GOOS و GOARCH المـرتبطين بالمنصـة الحالية، لتحديد كيفية بناء الملف التنفيذي. لمعرفة تركيبة المفتـاح-قيمـة للمنصـة الـتي تعمـل عليهـا، يمكنـك استخدام الأمر go env وتمرير GOOS و GOARCH مثل وسيطين:

```
$ go env GOOS GOARCH
```

يعمل الجهاز الذي نستخدمه بنظام ماك ومعمارية AMD64 لذا سيكون الخرج:

```
darwin
amd64
```

أي أن منصتنا لديها القيم التالية لمتغيرات البيئة GOOS=darwin و GOARCH=amd64.

أنت الآن تعرف ما هي GOOS و GOARCH، إضافةً إلى قيمهما المحتملـة. سـنكتب الآن برنامجًـا لاسـتخدامه مثالًا على كيفية استخدام متغيرات البيئة هذه ووسوم البنية، بهدف بناء ملفات تنفيذية لمنصات أخرى.

27.3 بناء تطبيق يعتمد على المنصة

سنبدأ أولًا ببناء برنامج بسيط، ومن الأمثلة الجيدة لهذا الغرض الدالة Join من الحزمـة path/filepath من مكتبة جو القياسية، إذ تأخذ هذه الدالة عدة سلاسـل وتُرجـع سلسـلةً مكونـةً من تلـك السلاسـل بعـد ربطهـا اعتمادًا على فاصل مسار الملف filepath.

يُعد هذا المثال التوضيحي مناسبًا لأن تشغيل البرنامج يعتمد على نظام التشغيل الذي يعمل عليه، ففي نظام التشغيل ويندوز، يكون فاصل المسار \ ، بينما تستخدم الأنظمة المستندة إلى يونكس Unix الفاصل /.

سنبدأ ببناء تطبيق يستخدم `filepath.Join()`، وسنكتب لاحقًا تنفيذًا خاصًا لهذه الدالة يُخصص الشيفرة للملفات التنفيذية التي تتبع لمنصة محددة.

أنشئ مجلدًا داخل المجلد `src` باسم تطبيقك:

```
$ mkdir app
```

انتقل إلى المجلد:

```
$ cd app
```

أنشئ ملفًا باسم main.go من خلال محرر النصوص نانو nano أو أي محرر آخر:

```
$ nano main.go
```

ضع في الملف الشيفرة التالية:

```
package main
import (
 "fmt"
 "path/filepath"
)
func main() {
 s := filepath.Join("a", "b", "c")
 fmt.Println(s)
}
```

تستخدم الدالة الرئيسية `main()` هنا الدالة `filepath.Join()` لربط ثلاث سلاسل مع فاصل المسار الصحيح المعتمد على المنصة.

احفظ الملف واخرج منه، ثم نفّذه من خلال الأمر التالي:

```
$ go run main.go
```

عند تشغيل هذا البرنامج، ستتلقى مخرجات مختلفة بناءً على المنصة التي تستخدمها، ففي نظام التشغيل ويندوز، سترى السلاسل مفصولة بالفاصل \ :

```
a\b\c
```

أما في أنظمة يونِكس مثل ماك ولِنُكس:

```
a/b/c
```

يوضح هـذا أن اختلاف بروتوكـولات نظـام الملفـات المسـتخدمة في أنظمـة التشـغيل هـذه، يقتضـي على البرنامج بناء شيفرات مختلفة للمنصـات المختلفـة. نحن نعلم أن الاختلاف هنـا سـيكون بفاصـل الملفـات كمـا تحدثنا، وبما أننا نستخدم الدالة `filepath.Join()` فلا خوف من ذلك، لأنها ستأخذ بالحسـبان اختلاف نظـام التشغيل الذي تُستخدم ضمنه. تفحص سلسلة أدوات جو تلقائيًا `GOOS` و `GOARCH` في جهـازك وتسـتخدم هـذه المعلومات لاستخدام الشيفرة المناسبة مع وسوم البنية الصحيحة وفاصل الملفات المناسب.

سنرى الآن من أين تحصل الدالة `filepath.Join()` على الفاصل المناسب. شغّل الأمـر التـالي لفحص المقتطف ذي الصلة من مكتبة جو القياسية:

```
$ less /usr/local/go/src/os/path_unix.go
```

سـيُظهر هـذا الأمـر محتويـات الملـف path_unix.go. ألـقِ نظـرةً على السـطر الأول منـه، والـذي يُمثّـل وسوم البنية.

```
. . .
// +build aix darwin dragonfly freebsd js,wasm linux nacl netbsd
openbsd solaris
package os
const (
 PathSeparator      = '/' // فاصل المسار الخاص بنظام التشغيل
 PathListSeparator = ':' // فاصل قائمة من المسارات الخاص بنظام التشغيل
)
. . .
```

تعرّف الشيفرة الفاصل `PathSeparator` المستخدم مع مختلف أنـواع الأنظمـة الـتي تسـتند على UNIX والتي تدعمها جو. لاحظ في السطر الأول وسوم البناء التي تمثل كل واحـدة منهـا قيمـةً محتملـةً للمتغـير `GOOS` وجميعها تمثل أنظمة تستند إلى يونكس. يأخذ `GOOS` أحد هذه القيم ويُنتج الفاصل المناسب وفق نوع النظام.

اضـغط `q` للعـودة لسـطر الأوامـر. افتح الآن ملـف path_windows الـذي يُعـبر عن سـلوك الدالـة `filepath.Join()` على نظام ويندوز:

```
. . .
package os
const (
  PathSeparator     = '\\' // فاصل المسار الخاص بنظام التشغيل
```

```
    PathListSeparator = ';' // فاصل قائمة من المسارات الخاص بنظام التشغيل
)
. . .
```

على الرغم من أن قيمة PathSeparator هنا هي \\، إلا أن الشيفرة ستعرض الشرطة المائلة الخلفية المفردة \ اللازمة لمسارات ملفات ويندوز، إذ تُستخدم الشرطة المائلة الأولى بمثابة مفتاح هروب Escape character.

لاحظ أنه في الملف الخاص بنظام يونكس كان لدينا وسوم بناء، أما في ملف ويندوز فلا يوجد وسوم بناء، وذلك لأن GOOS و GOARCH يمكن تمريرهما أيضًا إلى go build عن طريق إضافة شرطة سفلية _ وقيمة متغير البيئة مثل لاحقة suffix لاسم الملف (سنتحدث عن ذلك أكثر بعد قليل).

يجعل الجزء _windows من path_windows.go الملف يعمل كما لو كان يحتوي على وسم البناء // +build windows في أعلى الملف، لذلك عند تشغيل البرنامج على ويندوز، سيستخدم الثوابت PathSeparator و PathListSeparator من الشيفرة الموجودة في الملف path_windows.go.
اضغط q للعودة لسطر الأوامر.

أنشأت في هذه الخطوة برنامجًا يوضّح كيف يمكن لجو أن يجري التبديل تلقائيًا بين الشيفرات من خلال متغيرات البيئة GOOS و GOARCH ووسوم البنية. يمكنك الآن تحديث برنامجك وكتابة تنفيذك الخاص للدالة filepath.Join()، والاعتماد على وسوم البنية لتحديد الفاصل PathSeparator المناسب لمنصات ويندوز ويونكس يدويًا.

27.4 تنفيذ دالة خاصة بالمنصة

بعد أن تعرّفت على كيفية تحقيق مكتبة جو القياسية للتعليمات البرمجية الخاصة بالمنصة، يمكنك استخدام وسوم البنية لأجل ذلك في تطبيقك. ستكتب الآن تعريفًا خاصًا للدالة filepath.Join(). افتح ملف main.go الخاص بتطبيقك:

```
$ nano main.go
```

استبدل محتويات main.go وضع فيه الشيفرة التالية التي تتضمن دالة خاصة اسميناها Join:

```
package main
import (
 "fmt"
 "strings"
)
```

```
func Join(parts ...string) string {
 return strings.Join(parts, PathSeparator)
}
func main() {
 s := Join("a", "b", "c")
 fmt.Println(s)
}
```

تأخذ الدالة `Join` عدة سلاسل نصية من خلال المعامل `parts` وتربطهما معًا باستخدام الفاصل `PathSeparator`، وذلك من خلال الدالة `strings.Join()` من حزمة `strings`. لم نُعرّف `PathSeparator` بعد، لذا سنُعرّفه الآن في ملف آخر. احفظ main.go واخرج منه، وافتح المحرر المفضل لديك، وأنشئ ملفًا جديدًا باسم path.go:

```
nano path.go
```

صرّح عن الثابت `PathSeparator` وأسند له فاصل المسارات الخاص بملفات يونكس `/`:

```
package main
const PathSeparator = "/"
```

صرّف التطبيق وشغّله:

```
$ go build
$ ./app
```

ستحصل على الخرج التالي:

```
a/b/c
```

هذا جيد بالنسبة لأنظمة يونكس، لكنه ليس ما نريده تمامًا، فهو يُعطي دومًا `a/b/c` بغض النظر عن المنصة، وهذا لا يتناسب مع ويندوز. إذًا، نحن بحاجة إلى نسخة خاصة من `PathSeparator` لنظام ويندوز، وإخبار `go build` أي من هذه النسخ يجب استخدامها وفقًا للمنصة المطلوبة. هنا يأتي دور وسوم البنية.

27.5 استخدام وسوم البنية مع متغيرات البيئة

لكي نعالج حالة كون النظام هو ويندوز، سنُنشئ الآن ملفًا بديلًا للملف path.go وسنضيف وسوم بناء Build Tags مهمتها المطابقة مع المتغيرات `GOOS` و `GOARCH`، وذلك لضمان أن الشيفرة المستخدمة هي الشيفرة التي تعمل على المنصة المحددة.

أضف بدايةً وسم بناء إلى الملف path.go لإخباره أن يبني لكل شيء باستثناء ويندوز، افتح الملف:

```
$ nano path.go
```

أضِف وسم البناء التالي إلى الملف:

```
// +build !windows
package main
const PathSeparator = "/"
```

تُقدّم وسوم البنية في لغة جو إمكانية "العكس inverting" مما يعني أنه يمكنك توجيه جو لبناء هذا الملف من أجل أي منصة باستثناء ويندوز. لعكس وسم بناء، ضع "!" قبل الوسم كما فعلنا أعلاه واحفظ الملف واخرج منه.

والآن إذا حاولت تشغيل هذا البرنامج على ويندوز، ستتلقى الخطأ التالي:

```
./main.go:9:29: undefined: PathSeparator
```

في هذه الحالة لن تكون جو قادرةً على تضمين path.go لتعريف فاصل المسار PathSeparator.

الآن بعد أن تأكدت من أن path.go لن يعمل عندما يكون GOOS هو ويندوز. أنشئ ملفًا جديدًا باسم windows.go:

```
$ nano windows.go
```

أضِف ضمن هذا الملف PathSeparator ووسم بناء أيضًا لإخبار الأمر go build أن هذا الملف هو التحقيق المقابل للويندوز:

```
// +build windows
package main
const PathSeparator = "\\"
```

احفظ الملف واخرج من محرر النصوص. يمكن للتطبيق الآن تصريف نسخة لنظام ويندوز ونسخة أخرى لباقي الأنظمة. ستُبنى الآن ملفات تنفيذية بطريقة صحيحة وفقًا للمنصة المطلوبة، إلا أنه هناك المزيد من التغييرات التي يجب عليك إجراؤها من أجل التصريف على المنصة التي لا يمكنك الوصول إليها.

سنُعدّل في الخطوة التالية متغيرات البيئة المحلية GOOS و GOARCH.

27.6 استخدام متغيرات البيئة المحلية GOOS و GOARCH

استخدمنا سابقًا الأمر `go env GOOS GOARCH` لمعرفة النظام والمعمارية التي تعمل عليها. عند استخدام الأمر `go env`، سيبحث عن المتغيرين `GOOS` و `GOARCH`، فإذا وجدهما سيستخدم قيمهما وإلا سيفترض أن قيمهما هي معلومات المنصة الحالية (نظام ومعمارية المنصة). نستنتج مما سبق أنه بإمكاننا تحديد نظام ومعمارية لمتغيرات البيئة غير نظام ومعمارية المنصة الحالية. يعمل الأمر `go build` بطريقة مشابهة للأمر السابق، إذ يمكنك تحديد قيم لمتغيرات البيئة `GOOS` و `GOARCH` مختلفة عن المنصة الحالية.

إذا كنت لا تستخدم نظام ويندوز، ابنِ نسخةً تنفيذية من تطبيقك لنظام ويندوز عن طريق تعيين قيمة متغير البيئة `GOOS` على `windows` عند تشغيل الأمر `go build`:

```
$ GOOS=windows go build
```

اسرد الآن محتويات مجلدك الحالي:

```
$ ls
```

ستجد في الخرج ملفًا باسم app.exe، إذ يكون امتداده exe والذي يشير إلى ملف ثنائي تنفيذي في نظام ويندوز.

```
app app.exe main.go path.go windows.go
```

يمكنك باستخدام الأمر `file` الحصول على مزيد من المعلومات حول هذا الملف، للتأكد من بنائه:

```
$ file app.exe
```

ستحصل على:

```
app.exe: PE32+ executable (console) x86-64 (stripped to external PDB),
for MS Windows
```

يمكنك أيضًا إعداد واحد أو اثنين من متغيرات البيئة أثناء وقت البناء. نفّذ الأمر التالي:

```
$ GOOS=linux GOARCH=ppc64 go build
```

بذلك يكون قد استُبدل ملفك التنفيذي app بملف لمعمارية مختلفة. شغّل الأمر `file` على تطبيقك:

```
$ file app
```

ستحصل على الخرج التالي:

```
app: ELF 64-bit MSB executable, 64-bit PowerPC or cisco 7500, version
1 (SYSV), statically linked, not stripped
```

من خلال متغيرات البيئة GOOS و GOARCH يمكنك الآن إنشاء تطبيق قابل للتنفيذ على منصات مختلفة، دون الحاجة إلى إعدادات ضبط مُعقدة. سنستخدم في الخطوة التالية بعض الاصطلاحات لأسماء الملفات لكي نجعل تنسيقها دقيقًا، إضافةً إلى البناء تلقائيًّا من دون الحاجة إلى وسوم البنية.

27.7 استخدام لواحق اسم الملف مثل دليل إلى المنصة المطلوبة

كما لاحظت سابقًا، تعتمد جو على استخدام وسوم البنية كثيرًا لفصل الإصدارات أو النسخ المختلفة من التطبيق إلى ملفات مختلفة بغية تبسيط التعليمات البرمجية، فعندما فتحت ملف os/path_unix.go كان هناك وسم بناء يعرض جميع التركيبات المحتملة التي تُعبر عن منصات شبيهة أو تستند إلى يونكس، إلا أن ملف os/path_windows.go لا يحتوي على وسوم بناء، لأن لاحقة اسم الملف كانت كافية لإخبار جو بالمنصة المطلوبة.

دعونا نلقي نظرةً على تركيب أو قاعدة هذه الميزة، فعند تسمية ملف امتداده go.، يمكنك إضافة GOOS و GOARCH مثل لواحق إلى اسم الملف بالترتيب، مع فصل القيم عن طريق الشرطات السفلية _ فإذا كان لديك ملف جو يُسمى filename.go، يمكنك تحديد نظام التشغيل والمعمارية عن طريق تغيير اسم الملف إلى filename_GOOS_GOARCH.go. على سبيل المثال، إذا كنت ترغب في تصريفه لنظام التشغيل ويندوز باستخدام معمارية ARM 64-bit، فيمكنك كتابة اسم الملف filename_windows_arm64.go، إذ يساعد اصطلاح التسمية هذا في الحفاظ على الشيفرة منظمةً بدقة.

حدّث البرنامج الآن بحيث نستخدم لاحقات اسم الملف بدلًا من وسوم البنية، وأعد تسمية ملف path.go و windows.go لاستخدام الاصطلاح المستخدم في حزمة os:

```
$ mv path.go path_unix.go
$ mv windows.go path_windows.go
```

بعد تعديل اسم الملف أصبح بإمكانك حذف وسم البناء من ملف path_windows.go:

```
$ nano path_windows.go
```

بعد حذف // +build windows سيكون ملفك كما يلي:

```
package main
const PathSeparator = "\\"
```

احفظ الملف واخرج منه.

بما أن `unix` هي قيمة غير صالحة للمتغير `GOOS`، فإن اللاحقة `_unix.go` ليس لها أي معنى لمُصرّف جو، وبالرغم من ذلك، فإنه ينقل الغاية المرجوة من الملف. لا يزال مثلًا ملف `os/path_unix.go` بحاجة إلى استخدام وسوم البنية، لذا يجب الاحتفاظ بهذا الملف دون تغيير.

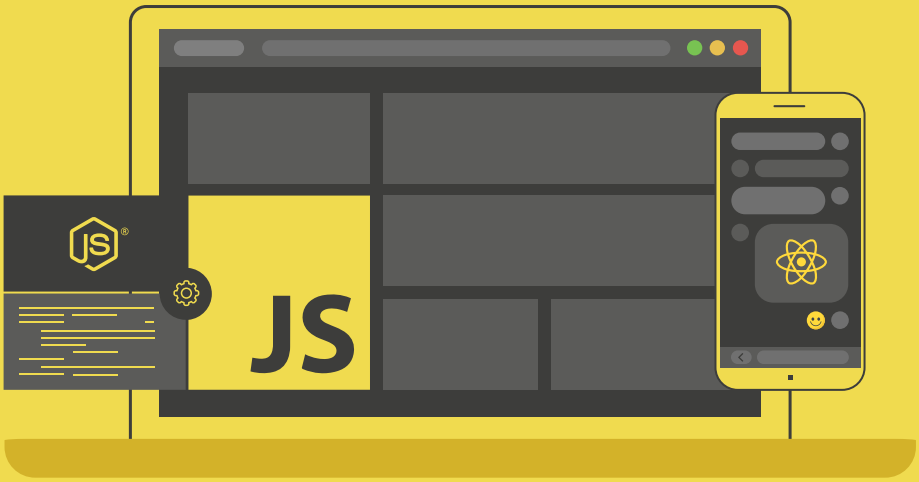
نستنتج أنه من خلال الاصطلاحات استطعنا التخلص من وسوم البنية غير الضرورية التي أضفناها إلى شيفرات التطبيق، كما جعلنا نظام الملفات أكثر تنظيمًا ووضوحًا.

27.8 الخاتمة

لا تحتاج لغة جو إلى أدوات إضافية لدعم فكرة المنصات المتعددة، وهذه ميزة قوية في جو. وقد تعلمنا في هذا الفصل استخدام هذه الإمكانية عن طريق إضافة وسوم البنية واللواحق لاسم الملف، وذلك لتحديد الشيفرات البرمجية التي يجب تنفيذها وفقًا للمنصة المطلوب العمل عليه.

أنشأنا تطبيقًا متعدد المنصات وتعلمنا كيفية التعامل مع متغيرات البيئة `GOOS` و `GOARCH` لبناء ملفات تنفيذية لمنصات أخرى تختلف عن المنصة الحالية. وختامًا ننوه لأن تعدد المنصات أمر مهم جدًا، فهو يُضيف ميزةً مهمة لتطبيقك، إذ تُمكنه من التصريف وفقًا للمنصة المطلوبة من خلال متغيرات البيئة هذه.

دورة تطوير التطبيقات
باستخدام لغة JavaScript



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

التحق بالدورة الآن

28. ضبط إصدار التطبيقات بالراية ldflags

يؤدي بناء الثنائيات أو الملفات التنفيذية Binaries جنبًا إلى جنب مع إنشاء البيانات الوصفية Metadata والمعلومات الأخرى المتعلقة بالإصدار Version عند نشر التطبيقات إلى تحسين عمليات المراقبة Monitoring والتسجيل Logging وتصحيح الأخطاء، وذلك من خلال إضافة معلومات تعريف تساعد في تتبع عمليات البناء التي تُجريها بمرور الوقت. يمكن أن تتضمن معلومات الإصدار العديد من الأشياء التي تتسم بالديناميكية، مثل وقت البناء والجهاز أو المستخدم الذي أجرى عملية بناء الملف التنفيذي ورقم المعرّف ID للإيداع Commit على نظام إدارة الإصدار VCS الذي تستخدمه Git مثلًا. بما أن هذه المعلومات تتغير باستمرار، ستكون كتابة هذه المعلومات ضمن الشيفرة المصدر مباشرةً، وتعديلها في كل مرة نرغب فيها بإجراء تعديل أو بناء جديد أمرًا مملًا، وقد يُعرّض التطبيق لأخطاء. يمكن للملفات المصدرية التنقل وقد تُبدّل المتغيرات أو الثوابت الملفات خلال عملية التطوير، مما يؤدي إلى كسر عملية البناء.

إحدى الطرق المستخدمة لحل هذه المشكلة في لغة جو هي استخدام الراية `-ldflags` مع الأمر `go build` لإدراج معلومات ديناميكية في الملف الثنائي التنفيذي في وقت البناء دون الحاجة إلى تعديل التعليمات البرمجية المصدرية. تُشير `Id` ضمن الراية السابقة إلى الرابط linker، الذي يُمثّل البرنامج الذي يربط الأجزاء المختلفة من الشيفرة المصدرية المُصرّفة مع الملف التنفيذي النهائي. إذًا، تعني `ldflags` رايات الرابط linker flags؛ لأنها تمرر إشارة إلى الأداة `cmd/link` الخاصة بلغة جو، والتي تسمح لك بتغيير قيم الحزم المستوردة في وقت البناء من سطر الأوامر.

سنستخدم في هذا الفصل الراية `-ldflags` لتغيير قيمة المتغيرات في وقت البناء وإدخال المعلومات الديناميكية ضمن ملف ثنائي تنفيذي، من خلال تطبيق يطبع معلومات الإصدار على الشاشة.

28.1 المتطلبات

ستحتاج إلى امتلاك مساحة عمل خاصة مُهيئة وجاهزة في لغة جو Go، فإذا لم تكن قد أنشأت واحدة، فارجع للتعليمات الواردة في الفصل الأول من الكتاب، وقم بتثبيت لغة جو Go وإعداد بيئة تطوير محلية بحسب نظام تشغيلك.

28.2 بناء تطبيق تجريبي

يجب أن يكون لدينا تطبيق حتى نستطيع تجريب عملية إدراج المعلومات إليه ديناميكيًا من خلال الراية `-ldflags`. سنبني في هذه الخطوة تطبيقًا بسيطًا للتجريب عليه، بحيث يطبع المعلومات الخاصة بالإصدار.

بدايةً أنشئ مجلدًا باسم `app` يُمثّل اسم التطبيق داخل المجلد `src`:

```
$ mkdir app
```

انتقل إلى هذا المجلد:

```
$ cd app
```

أنشئ باستخدام محرر النصوص الذي تُفضّله وليكن نانو nano الملف main.go:

```
$ nano main.go
```

ضع الشيفرة التالية بداخل هذا الملف، والتي تؤدي إلى طباعة معلومات الإصدار الحالي من التطبيق:

```
package main
import (
  "fmt"
)
var Version = "development"
func main() {
  fmt.Println("Version:\t", Version)
}
```

صرّحنا داخل الدالة `main` عن متغير يُدعى `Version`، ثم طبعنا السلسلة النصية `Version:` متبوعةً بإزاحة جدول واحدة tab كما يلي: `\t`، ثم قيمة المتغير `Version`. هنا أعطينا متغير الإصدار القيمة `development`، والتي ستكون إشارةً إلى الإصدار الافتراضي من التطبيق. سنُعدّل لاحقًا قيمة هذا المتغير، بحيث تُشير إلى الإصدار الرسمي من التطبيق، ووفقًا للتنسيق المُتبع في تسمية الإصدارات.

احفظ واغلق الملف، ثم ابنِ الملف وشغّله للتأكد من أنه يعمل:

```
$ go build
$ ./app
```

ستحصل على الخرج التالي:

```
Version: development
```

لديك الآن تطبيق يطبع معلومات الإصدار الافتراضي، ولكن ليس لديك حتى الآن طريقةً لتمرير معلومات الإصدار الحالي في وقت البناء. ستستخدم في الخطوة التالية الراية `-ldflags` لحل هذه المشكلة.

28.3 استخدام ldflags مع go build

تحدّثنا سابقًا أن رايات الربط تُستخدم لتمرير الرايات إلى أداة الربط الأساسية الخاصة بلغة جو. يحدث ذلك وفقًا للصيغة التالية:

```
$ go build -ldflags="-flag"
```

هنا مرّرنا `flag` إلى الأداة الأساسية `go tool link` التي تعمل بمثابة جزء من الأمر `go build`. هنا وضعنا علامتي اقتباس حول القيمة التي نمررها إلى `ldflags`، وذلك لكي نمنع حدوث التباس لدى سطر الأوامر (كي لا يفسرها بطريقة خاطئة أو يعدها عدة محارف كل منها لغرض مختلف). يمكنك تمرير العديد من رايات الروابط، وفي هذا الفصل سنحتاج إلى استخدام الراية `-X` لضبط معلومات متغير الإصدار في وقت الربط link time، وسيتبعها مسار المتغير (هنا اسم الحزمة متبوعة باسم المتغير) مع قيمته الجديدة.

```
$ go build -ldflags="-X 'package_path.variable_name=new_value'"
```

لاحظ أننا وضعنا الخيار `-X` ضمن علامتي اقتباس وبجانبها وضعنا اسم الحزمة متبوعةً بنقطة "." متبوعةً باسم المتغير والقيمة الجديدة وأحطناهم بعلامات اقتباس مفردة لكي يُفسرها سطر الأوامر على أنها كتلة واحدة. إذًا، سنستخدم الصيغة السابقة لاستبدال قيمة متغير الإصدار `Version` في تطبيقنا:

```
$ go build -ldflags="-X 'main.Version=v1.0.0'"
```

تمثّل `main` هنا مسار الحزمة للمتغير `Version`، لأنه يتواجد داخل الملف main.go. هنا `Version` هو المتغير المطلوب تعديله، والقيمة `v1.0.0` هي القيمة الجديدة التي نريد ضبطه عليها.

عندما نستخدم الراية `ldflags`، يجب أن تكون القيمة التي تريد تغييرها موجودة وأن يكون المتغير موجودًا ضمن مستوى الحزمة ومن نوع `string`. لا يُسمح بأن يكون المتغير ثابتًا `const`، أو أن تُضبط قيمته من خلال استدعاء دالة. يتوافق كل شيء هنا مع المتطلبات، لذا ستعمل الأمور كما هو متوقع؛ فالمتغير موجود ضمن الملف main.go والمتغير والقيمة `v1.0.0` التي نريد ضبط المتغير عليها كلاهما من النوع `string`.

شغّل التطبيق بعد بنائه:

```
$./app
```

ستحصل على الخرج التالي:

```
Version: v1.0.0
```

إذًا، استطعنا من خلال الراية `-ldflags` تعديل قيمة متغير الإصدار من `development` إلى `v1.0.0` في وقت البناء. يمكنك تضمين تفاصيل الإصدار ومعلومات الترخيص وغيرهم من المعلومات باستخدام `-ldflags` جنبًا إلى جنب مع الملف التنفيذي النهائي الذي ترغب بنشره وذلك فقط من خلال سطر الأوامر.

في هذا المثال: كان المتغير موجود في مسار واضح، لكن في أمثلة أخرى قد يكون العثور على مسار المتغير أمرًا معقدًا. سنناقش في الخطوة التالية هذا الموضوع، وسنرى ماهي أفضل الطرق لتحديد مسارات المتغيرات الموجودة في حزم فرعية في الحزم ذات الهيكلية الأكثر تعقيدًا.

28.4 تحديد مسار الحزمة للمتغيرات

كنا قد وضعنا في المثال السابق متغير الإصدار `Version` ضمن المستوى الأعلى من الحزمة في الملف main.go، لذا كان أمر الوصول إليه بسيطًا. هذه حالة مثالية، لكن في الواقع هذا لايحدث دائمًا، فأحيانًا تكون المتغيرات ضمن حزمة فرعية أخرى. عمومًا، لا يُحبذ وضع هكذا متغيرات ضمن `main`، لأنه حزمة غير قابلة للاستيراد، ويُفضّل وضع هكذا متغيرات ضمن حزمة أخرى. لذا سنعدل بعض الأمور في تطبيقنا، إذ سنُنشئ الحزمة `app/build` ونضع فيها معلومات حول وقت بناء الملف التنفيذي واسم المستخدم الذي بناه.

أنشئ مجلدًا جديدًا باسم الحزمة الجديدة:

```
$ mkdir -p build
```

أنشئ ملفًا جديدًا باسم build.go من أجل وضع المتغيرات ضمنه:

```
$ nano build/build.go
```

ضع بداخله المتغيرات التالية بعد فتحه باستخدام محرر النصوص الذي تريده:

```
package build
var Time string # سيُخزّن وقت بناء التطبيق
var User string # سيُخزّن اسم المستخدم الذي بناه
```

لا يمكن لهذين المتغيرين أن يتواجدا بدون قيم، لذا لا داعٍ لوضع قيم افتراضية لهما كما فعلنا مع متغير الإصدار. احفظ وأغلق الملف.

افتح ملف main.go لوضع المتغيرات داخله:

```
$ nano main.go
```

ضع فيه المحتويات التالية:

```
package main
import (
  "app/build"
  "fmt"
)
var Version = "development"
func main() {
  fmt.Println("Version:\t", Version)
  fmt.Println("build.Time:\t", build.Time)
  fmt.Println("build.User:\t", build.User)
}
```

استوردنا الحزمة app/build، ثم طبعنا قيمة build.Time و build.User بنفس الطريقة التي طبعنا فيها Version سابقًا. احفظ وأغلِق الملف.

إذا أردت الآن الوصول إلى هذه المتغيرات عند استخدام الراية -ldflags، يمكنك استخدام اسم الحزمة app/build يتبعها .Time أو .User كوننا نعرف مسار الحزمة. سنستخدم الأمر nm بدلًا من ذلك من أجل محاكاة موقف أكثر تعقيدًا، والذي يكون فيه مسار الحزمة غير واضح.

يُنتج الأمر go tool nm الرموز المتضمنة في ملف تنفيذي أو ملف كائن أو أرشيف، إذ يشير الرمز إلى كائن موجود في الشيفرة (متغير أو دالة مُعرّفة أو مستوردة). يمكنك العثور بسرعة على معلومات المسار من خلال إنشاء جدول رموز باستخدام nm واستخدام grep للبحث عن متغير.

> لن يساعدك الأمر nm في العثور على مسار المتغير إذا كان اسم الحزمة يحتوي على أي محارف ليست ASCII، أو " أو ٪ (هذه قيود خاصة بالأداة).

ابنِ التطبيق أولًا لاستخدام هذا الأمر:

```
$ go build
```

وجّه الأداة nm إلى التطبيق بعد بنائه وابحث في الخرج:

```
$ go tool nm ./app | grep app
```

عند تشغيل الأمر `nm` ستحصل على العديد من البيانات، لذا وضعنا `|` لتوجيه الخرج للأمر `grep` الذي يبحث عن المسارات التي تحتوي على الاسم `app` في المستوى الأعلى منها، وتحصل على الخرج التالي:

```
55d2c0 D app/build.Time
55d2d0 D app/build.User
4069a0 T runtime.appendIntStr
462580 T strconv.appendEscapedRune
. . .
```

يظهر في أول سطرين مسارات المتغيرات التي تبحث عنها: `app/build.Time` و `app/build.User`.

ابنِ التطبيق الآن بعد أن تعرفت على المسارات، وعدّل متغير الإصدار إضافةً إلى المتغيرات الجديدة التي أضفناها والتي تُمثّل وقت بناء التطبيق واسم المستخدم (تذكر أنك تُعدّل هذه المتغيرات في وقت البناء). لأجل ذلك ستحتاج إلى تمرير عدة رايات `-X` إلى الراية `-ldflags`:

```
$ go build -v -ldflags="-X 'main.Version=v1.0.0' -X 'app/build.User=$
(id -u -n)' -X 'app/build.Time=$(date)'"
```

هنا مررنا الأمر `id -u -n` لعرض المستخدم الحالي، والأمر `date` لعرض التاريخ الحالي.

شغّل التطبيق بعد بنائه:

```
$ ./app
```

ستحصل على الخرج التالي في حال كنت تعمل على نظام يستند إلى يونكس Unix:

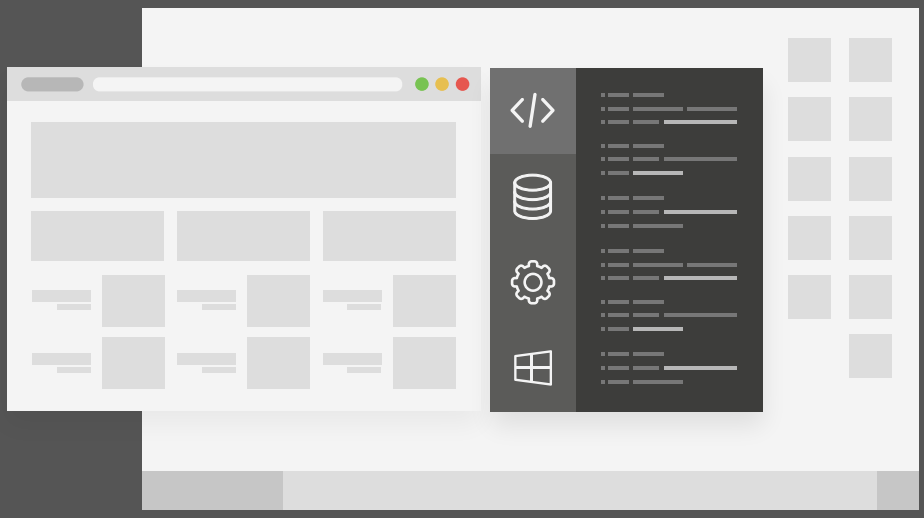
```
Version: v1.0.0
build.Time:    Fri Oct 4 19:49:19 UTC 2019
build.User:    sammy
```

لديك الآن ملف تنفيذي يتضمن معلومات الإصدار والبناء، يساعدك في مرحلة الإنتاج وحل المشكلات.

28.5 الخاتمة

وضّح لك هذا الفصل من الكتاب مدى قوة استخدام `ldflags` لإدخال معلومات في وقت البناء إذا طُبقت بطريقة سليمة. يمكنك بهذه الطريقة التحكم في رايات الميزة feature flags (هي تقنية برمجية تُمكّن الفريق البرمجي من إجراء تغييرات بدون استخدام المزيد من التعليمات البرمجية) ومعلومات البيئة ومعلومات الإصدار والأمور الأخرى دون إدخال تغييرات على الشيفرة المصدر. يمكنك الاستفادة من الخصائص التي تمنحك إياها لغة جو لعمليات النشر من خلال استخدام `ldflags` في عمليات البناء.

دورة علوم الحاسوب



دورة تدريبية متكاملة تضعك على بوابة الاحتراف
في تعلم أساسيات البرمجة وعلوم الحاسوب

التحق بالدورة الآن

29. استخدام الحزمة Flag

نـادرًا مـا تكـون الأدوات المسـاعدة لسـطر الأوامـر Command-line utilities مفيـدةً دون ضـبط configuration إضافي وذلك عندما يتطلب الأمر إجراء عمليات خـارج الصـندوق. تُعـد الإعـدادات الافتراضـية للأوامر أمرًا جيدًا ومهمًا، لكن ينبغي أن تتميز بإمكانية قبول إعدادات ضبط محددة من المسـتخدمين أيضًـا. في معظم أنظمة التشغيل يمكن تخصـيص أوامـر سـطر الأوامـر من خلال اسـتخدام الرايـات Flags؛ وهي سلاسـل نصية تُضاف إلى أمر ما، بحيث تؤدي إلى سلوك خاص لهذا الأمر حسب قيمتها. تتيح لك لغة جـو إنشـاء أدوات مساعدة لسطر الأوامر تقبل رايات يمكن من خلالها تخصـيص سـلوك الأوامـر، وذلـك باسـتخدام حزمـة `flag` من المكتبة القياسية.

ستتعلم في هـذا الفصـل طرقًـا مختلفـة لاسـتخدام حزمـة `flag` بهـدف إنشـاء أنـواع مختلفـة من الأدوات المسـاعدة لسـطر الأوامـر. سنسـتخدم رايـةً للتحكم في خـرج برنـامج وتقـديم وسـطاء موضـعية Positional arguments (وهي وسطاء يجب وضعها في الموضع أو الـترتيب المناسـب المُحـدد مسـبقًا)، إذ يمكننـا مـزج الرايات والبيانات الأخرى وتنفيذ أوامر فرعية.

29.1 استخدام الرايات لتغيير سلوك البرنامج

يتضمن استخدام حزمة الراية ثلاث خطوات: تبدأ بتعريف متغيرات تلتقـط قيم الرايـات، ثم تحديـد الرايـات التي سيستخدمها التطبيق، وتنتهي بتحليل الرايات المُقدمـة للتطـبيق عنـد التنفيـذ. تُركّـز معظم الـدوال داخـل حزمة الراية على تعريف رايات وربطها بالمتغيرات التي تُعرّفها، وتنجز الدالة `Parse()` مرحلة التحليل.

لتوضيح الأمور سنُنشئ برنامجًا بسـيطًا يتضـمن رايـةً بوليانيـة تُغيّـر الرسـالة المطبوعـة؛ فـإذا كـانت الرايـة `-color` موجودة، سيطبع البرنامج الرسالة باللون الأزرق؛ وإذا لم تُقدّم أي راية، فلن يكون للرسالة أي لون.

أنشئ ملفًا باسم boolean.go:

```
$ nano boolean.go
```

أضِف ما يلي إلى الملف:

```
package main
import (
  "flag"
  "fmt"
)
type Color string
const (
  ColorBlack Color = "\u001b[30m"
  ColorRed     = "\u001b[31m"
  ColorGreen    = "\u001b[32m"
  ColorYellow    = "\u001b[33m"
  ColorBlue     = "\u001b[34m"
  ColorReset    = "\u001b[0m"
)
func colorize(color Color, message string) {
  fmt.Println(string(color), message, string(ColorReset))
}
func main() {
  useColor := flag.Bool("color", false, "display colorized output")
  flag.Parse()
  if *useColor {
        colorize(ColorBlue, "Hello, DigitalOcean!")
        return
  }
  fmt.Println("Hello, DigitalOcean!")
}
```

يستخدم هذا المثال سلاسل الهروب ANSI Escape Sequences لجعل الطرفية تُعطي خرجًا ملونًا؛ وهي سلاسل خاصة من المحارف، لذا من المنطقي أن نُعرّف نوع خاص بها. في مثالنا نسمي هذا النوع `Color` ونعرّفه على أنه `string`، ثم نُعرّف لوحة ألوان لاستخدامها بكتلة نسميها `const` تتضمن عدة خيارات لونية اعتمادًا على النوع السابق. تستقبل الدالة `colorize` المُعرّفة بعد الكتلة `const` قيمةً لونيةً من اللوحة السابقة (أي عمليًّا متغير من النوع `Color`) إضافةً إلى الرسالة المطلوب تلوينها. بعد ذلك توجّه الطرفية Terminal لتغيير اللون عن طريق طباعة تسلسل الهروب للون المطلوب، ثم طباعة الرسالة. أخيرًا، يُطلب من الطرفية إعادة

ضبط اللون الأساسي لها من خلال طباعة `ColorReset`، أي نطبع سلسلة الهروب للون المطلوب ثم رسالتنا، فتظهر باللون المطلوب، ثم نعيد ضبط اللون إلى حالته الأصلي.

نستخدم داخل الدالة `main` الدالة `flag.Bool` لتعريف راية بوليانية اسمها `color`. المعامل الثاني لهذه الدالة `false` هو القيمة الافتراضية للراية، أي عندما نُشغل البرنامج بدونها. على عكس ما قد تتوقعه، فإن ضبط هذا المعامل على `true` هو أمر غير صحيح، لأننا لا نريد أن يُطبق سلوك التلوين إلا عندما نُمرر الراية كما سترى بعد قليل. إذًا، تكون قيمة هذا المعامل تكون غالبًا `false` مع الرايات البوليانية.

المعامل الأخير هو نص توضيحي لهذه الراية، أي كأنه توثيق استخدام أو وصف. القيمة التي تعيدها الدالة هي مؤشر إلى `bool`، وتُضبط قيمة هذا المؤشر من خلال الدالة `flag.Parse` بناءً على الراية التي يُمررها المستخدم. يمكننا بعد ذلك التحقق من قيمة هذا المؤشر البولياني عن طريق تحصيل قيمته باستخدام المعامل *. إذًا باستخدام هذه القيمة المنطقية، يمكننا استدعاء `colorize` عند تمرير `-color` أو استدعاء دالة الطباعة العادية `fmt.Println` دون تلوين إذا لم تُمرر الراية. احفظ وأغلق الملف وشغله بدون تقديم أي راية:

```
$ go run boolean.go
```

ستحصل على الخرج التالي:

```
Hello, DigitalOcean!
```

أعد تشغيله مع تمرير الراية `-color`:

```
$ go run boolean.go -color
```

سيكون الناتج هو نفس النص، ولكن هذه المرة باللون الأزرق.

يمكنك أيضًا إرسال أسماء ملفات أو بيانات أخرى إلى البرنامج، وليس فقط رايات.

29.2 التعامل مع الوسطاء الموضعية

تمتلك الأوامر غالبًا بعض الوسطاء التي تحدد سلوكها. لنأخذ مثلًا الأمر `head` الذي يطبع الأسطر الأولى من ملف ما. غالبًا ما يُستدعى هذا الأمر بالشكل التالي `head example.txt`، إذ يُمثل الملف example.txt وسيطًا موضعيًّا هنا.

تستمر الدالة `Parse()` في تحليل الرايات التي تصادفها ريثما تجد شيئًا آخر لا يُمثّل راية (سنرى بعد قليل أنها ستتوقف عند مصادفة وسيط موضعي). لذا سنتعرف الآن على دوال أخرى توفرها الحزمة `flag` هي الدالة `Args()` والدالة `Arg()`. لتوضيح ذلك سنعيد تنفيذ الأمر `head` الذي يعرض أول عدة أسطر من ملف معين.

أنشئ ملفًا جديدًا يسمى head.go وأضف الشيفرة التالية:

```
package main
import (
  "bufio"
  "flag"
  "fmt"
  "io"
  "os"
)
func main() {
  var count int
  flag.IntVar(&count, "n", 5, "number of lines to read from the file")
  flag.Parse()
  var in io.Reader
  if filename := flag.Arg(0); filename != "" {
        f, err := os.Open(filename)
        if err != nil {
              fmt.Println("error opening file: err:", err)
              os.Exit(1)
        }
        defer f.Close()
        in = f
  } else {
        in = os.Stdin
  }
  buf := bufio.NewScanner(in)
  for i := 0; i < count; i++ {
        if !buf.Scan() {
              break
        }
        fmt.Println(buf.Text())
  }
  if err := buf.Err(); err != nil {
        fmt.Fprintln(os.Stderr, "error reading: err:", err)
  }
}
```

نُعرّف بدايةً المتغير `count` الذي سيُخزّن عدد الأسطر التي يجب أن يقرأها البرنامج من الملف، كما نُعرّف أيضًا الراية `-n` باستخدام `flag.IntVar` لكي نعكس سلوك البرنامج `head`. تسمح لنا هذه الدالة بتمرير المؤشر الخاص بنا إلى متغير على عكس دوال حزمة الراية `flag` الأخرى، التي لا تحتوي على اللاحقة `Var`. بغض النظر عن هذا الفرق، ستكون بقية المعاملات للدالة `flag.IntVar` مثل نظيراتها في دوال `flag.Int` الأخرى التي لا تتضمن هذه اللاحقة، أي ستكون المعاملات كما يلي: اسم الراية ثم القيمة الافتراضية ثم الوصف. بعد ذلك، نستدعي دالة `flag.Parse()` لتفسير دخل المستخدم.

يقرأ القسم التالي الملف، إذ نعرّف متغيرًا باسم `io.Reader` الذي سيُضبط إما على الملف الذي يطلبه المستخدم، أو الدخل القياسي الذي يُمرر إلى البرنامج. نستخدم الدالة `flag.Arg` داخل التعليمة `if` للوصول إلى أول وسيط موضعي يأتي بعد الرايات. ستكون قيمة `filename`، إما اسم ملف يُقدمه المستخدم أو سلسلة فارغة `""`؛ ففي حال تقديم اسم ملف، تُستخدم الدالة `os.Open` لفتح الملف وضبط المتغير `io.Reader` سالف الذكر؛ وإذا لم يُقدم اسم ملف (أي سلسلة فارغة) فإننا نستخدم `os.Stdin` للقراءة من الدخل القياسي.

يستخدم القسم الأخير `*bufio.Scanner` الذي أنشئ باستخدام `bufio.NewScanner` لقراءة الأسطر من متغير `io.Reader` الذي يُمثله `in`. بعد ذلك، نكرّر حلقة عدة مرات حسب قيمة `count`. تُستدعى `break` إذا كان ناتج قراءة سطر باستخدام `buf.Scan` هو القيمة `false`، إذ يشير ذلك إلى أن عدد الأسطر أقل من الرقم الذي يطلبه المستخدم.

شغّل هذا البرنامج واعرض محتويات الملف البرمجي نفسه الذي كتبته للتو من خلال استخدام `head.go` مثل وسيط، أي سنشغّل البرنامج head.go ونقرأ محتوياته أيضًا:

```
$ go run head.go -- head.go
```

الفاصل `--` هو راية، إذ يُفسَّر وجودها من قِبل حزمة الراية `flag` على أنه لن يكون هناك رايات بدءًا منها.

سيكون الخرج كما يلي:

```
package main
import (
    "bufio"
    "flag"
```

دعنا نستخدم الراية `-n` التي عرّفناها لتحديد عدد الأسطر المقروءة:

```
$ go run head.go -n 1 head.go
```

سيكون الخرج هو أول سطر فقط من الملف:

```
package main
```

أخيرًا، عندما يكتشف البرنامج أنه لم تُقدّم أية وسطاء موضعية، سيقرأ من الدخل القياسي، تمامًا مثل الأمر head. جرّب تشغيل هذا الأمر:

```
$ echo "fish\nlobsters\nsharks\nminnows" | go run head.go -n 3
```

سيكون الخرج على النحو التالي:

```
fish
lobsters
sharks
```

يقتصر سلوك دوال حزمة الراية التي رأيناها حتى الآن على فحص الأمر المُستدعى كاملًا. لا نريد دائمًا هذا السلوك، خاصةً إذا كانت الأداة التي نريدها تتضمن أوامر فرعية Sub-commands.

29.3 استخدام FlagSet لدعم إمكانية تحقيق الأوامر الفرعية

تتضمن تطبيقات سطر الأوامر الحديثة غالبًا وجود أوامر فرعية، بهدف تجميع مجموعة من الأدوات تحت أمر واحد. الأداة الأكثر شهرة التي تستخدم هذا النمط هي git، فمثلًا في `git init`، لدينا `git` هو الأمر الأساسي و `init` هو الأمر الفرعي التابع له. إحدى السمات البارزة للأوامر الفرعية هي أن كل أمر فرعي يمكن أن يكون له مجموعته الخاصة من الرايات.

يمكن لتطبيقات لغة جو أن تدعم الأوامر الفرعية من خلال نوع خاص يُدعى `flag.(*FlagSet)`. سننشئ برنامجًا يُنفذ أمرًا باستخدام أمرين فرعيين وبرايات مختلفة، وذلك لكي نجعل الأمور واضحة.

نُنشئ ملفًا جديدًا يسمى subcommand.go بالمحتويات التالية:

```
package main
import (
  "errors"
  "flag"
  "fmt"
  "os"
)
func NewGreetCommand() *GreetCommand {
  gc := &GreetCommand{
        fs: flag.NewFlagSet("greet", flag.ContinueOnError),
  }
```

```
    gc.fs.StringVar(&gc.name, "name", "World", "name of the person to be
greeted")
    return gc
}
type GreetCommand struct {
    fs *flag.FlagSet
    name string
}
func (g *GreetCommand) Name() string {
    return g.fs.Name()
}
func (g *GreetCommand) Init(args []string) error {
    return g.fs.Parse(args)
}
func (g *GreetCommand) Run() error {
    fmt.Println("Hello", g.name, "!")
    return nil
}
type Runner interface {
    Init([]string) error
    Run() error
    Name() string
}
func root(args []string) error {
    if len(args) < 1 {
        return errors.New("You must pass a sub-command")
    }
    cmds := []Runner{
        NewGreetCommand(),
    }
    subcommand := os.Args[1]
    for _, cmd := range cmds {
        if cmd.Name() == subcommand {
            cmd.Init(os.Args[2:])
            return cmd.Run()
        }
    }
```

```
    return fmt.Errorf("Unknown subcommand: %s", subcommand)
}
func main() {
    if err := root(os.Args[1:]); err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}
```

ينقسم هذا البرنامج إلى عدة أجزاء: الدالة `main`، والدالة `root`، وبعض الدوال لتحقيق الأمر الفرعي.

تعالج الدالة `main` الأخطاء التي تُعاد من الأوامر، بينما تلتقط تعليمة `if` الأخطاء التي قد تُعيدها الدوال وتطبع الخطأ ويُنهى البرنامج وتعاد القيمة 1 إشارةً إلى حدوث خطأ إلى نظام التشغيل. نمرر جميع المعطيات التي استُدعي البرنامج معها داخل الدالة`main` إلى الدالة `root` مع حذف الوسيط الأول الذي يُمثل اسم البرنامج (في الأمثلة السابقة `.\subcommand`) من خلال استقطاع `os.Args`.

تعرّف الدالة الدالة `root` الأمر الفرعي`[]Runner` حيثما تُعرّف جميع الأوامر الفرعية. `Runner` هي واجهة الأوامر الفرعية التي تسمح لدالة `root` باسترداد اسم الأمر الفرعي باستخدام `Name()` ومقارنته بمحتويات المتغير `subcommand`. بمجرد تحديد الأمر الفرعي الصحيح بعد التكرار على متغير `cmds`، نُهيئ الأمر الفرعي مع بقية الوسطاء ونستدعي التابع `Run()` الخاص بهذا الأمر.

على الرغم من أننا نُعرّف أمرًا فرعيًّا واحدًا، إلا أن هذا الإطار سيسمح لنا بسهولة بإنشاء أوامر أخرى. نُعرّف نسخةً من النوع `GreetCommand` باستخدام `NewGreetCommand`، إذ نُنشئ `*flag.FlagSet` جديد باستخدام `flag.NewFlagSet`. تأخذ `flag.NewFlagSet` وسيطين، هما اسم مجموعة الرايات واستراتيجية للإبلاغ عن أخطاء التحليل. يمكن الوصول إلى اسم `*flag.FlagSet` باستخدام التابع `flag.(*FlagSet).Name`. نستخدم هذا في الطريقة `(*GreetCommand).Name()` بحيث يتطابق اسم الأمر الفرعي مع الاسم الذي قدمناه إلى `*flag.FlagSet`.

يُعرّف `NewGreetCommand` أيضًا الراية `-name` بطريقة مشابهة للأمثلة السابقة، ولكنه بدلًا من ذلك يستدعيها مثل تابع للحقل `*flag.FlagSet` في `*GreetCommand` (نكتب `gc.fs`).

عندما تستدعي `root` التابع `Init()` الذي يخص `*GreetCommand`، فإننا نمرر المعطيات المقدمة إلى التابع `Parse` الخاص بالحقل `*flag.FlagSet`.

سيكون من الأسهل أن ترى الأوامر الفرعية إذا بنيت هذا البرنامج ثم شغلته. إبنِ البرنامج كما يلي:

```
$ go build subcommand.go
```

الآن، شغّل البرنامج دون وسطاء:

```
$ ./subcommand
```

سيكون الخرج على النحو التالي:

```
You must pass a sub-command
```

شغّل الآن البرنامج مع استخدام الأمر الفرعي `greet`:

```
$ ./subcommand greet
```

سيكون الخرج على النحو التالي:

```
Hello World !
```

استخدم الآن الراية `name` مع `greet` لتحديد اسم:

```
$ ./subcommand greet -name Sammy
```

سيكون الخرج كما يلي:

```
Hello Sammy !
```

يوضح هذا المثال بعض المبادئ الكامنة وراء كيفية هيكلة تطبيقات سطر الأوامر الأكبر حجمًا أو الأكثر تعقيدًا في لغة جو. صُمم `FlagSets` لمنح المطورين مزيدًا من التحكم وكيفية معالجة الرايات من خلال منطق تحليل الراية.

29.4 الخاتمة

تمنح الرايات مستخدمي برنامجك التحكم في كيفية تنفيذ البرامج، أو التحكم بسلوكه، وبالتالي تجعله أكثر مرونةً في التعامل مع سياقات التنفيذ المحتملة. من المهم أن تمنح المستخدمين إعدادات افتراضية مفيدة، ولكن يجب أن تمنحهم الفرصة لتجاوز الإعدادات التي لا تناسب حالتهم. لقد رأينا أن حزمة الراية `flag` توفر خيارات مرنة لتعزيز إمكانية إضافة خيارات ضبط خاصة للمستخدمين. يمكنك اختيار بعض الرايات البسيطة، أو إنشاء مجموعة من الأوامر الفرعية القابلة للتوسيع. سيساعدك استخدام حزمة الراية في كلتا الحالتين على بناء أدوات مرنة لسطر الأوامر.

هل تريد كتابة سيرة ذاتية احترافية؟

نساعدك في إنشاء سيرة ذاتية احترافية عبر خبراء توظيف
مختصين في أكبر منصة توظيف عربية عن بعد

أنشئ سيرتك الذاتية الآن

30. استخدام الوحدات Modules

أضاف مؤلفو لغة جو في النسخة 1.13 طريقةً جديدةً لإدارة المكتبات الـتي يعتمـد عليهـا مشـروع مكتـوب باستخدام هذه اللغة، تسمى **وحدات Go**، وقد أتت استجابةً إلى حاجة المطورين إلى تسهيل عملية الحفـاظ على الإصدارات المختلفة من الاعتماديات dependencies وإضافة المزيد من المرونـة إلى الطريقـة الـتي ينظم بهـا المطورون مشاريعهم على أجهزتهم.

تتكون الوحدات عادةً في لغة جو من مشروع أو مكتبة واحدة إلى جانب مجموعة من حزم لغة جو التي تُطلق معًا بعد ذلك. تعمل الوحدات في هذه اللغة على حل العديد من المشكلات بالاستعانة بمتغـير البيئـة GOPATH الخاص بنظام التشغيل، من خلال السماح للمسـتخدمين بوضـع شـيفرة مشـروعهم في المجلـد الـذي يختارونـه وتحديد إصدارات الاعتماديات لكل وحدة.

سننشئ في هذا الفصل وحدة جو عامة public خاصة بنـا وسنضـيف إليهـا حزمـة، وسـنتعلم أيضًـا كيفيـة إضافة وحدة عامة أنشأها آخرون إلى مشروعنا، وإضافة إصدار محدد من هذه الوحدة إلى المشروع.

30.1 المتطلبات

تحتاج لامتلاك مساحة عمل خاصة مُهيئة وجاهزة في لغة جو (النسخة 1.13 فما فـوق)، فـإذا لم يكن لـديك واحدة، فارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبت لغة جو Go وقم بتجهيز بيئة تطـوير محليـة بحسب نظام تشغيلك، ويُفضّـل أن تطّلـع على فقـرة اسـتيراد الحـزم في لغـة جـو Go قبـل المتابعـة في قـراءة الفقرات التالية.

30.2 إنشاء وحدة جديدة

قد تبدو الوحدة module مشابهة للحزمة package للوهلة الأولى في لغة جو، لكن تحتوي الوحدة على عدد من الشيفرات التي تُحقق وظيفة الحزمة، إضافةً إلى ملفين مهمين في المجلد الجذر `root` هما الملف go.mod و go.sum. تحتوي هذه الملفات على معلومات تستخدمها أداة `go` لتتبع إعدادات ضبط Configuration الوحدة الخاصة بك، وتجري عادةً صيانتها بواسطة الأداة نفسها، لذا لن تكون مضطرًا لفعل ذلك بنفسك.

أول شيء عليك التفكير فيه هو المكان الذي ستضع فيه الوحدة، فمفهوم الوحدات يجعل بالإمكان وضع المشاريع في أي مكان من نظام الملفات، وليس فقط ضمن مجلد حُدد مسبقًا في جو.

ربما يكون لديك مجلد خاص لإنشاء المشاريع وترغب في استخدامه، لكن هنا سننشئ مجلدًا باسم `projects`، ونسمي الوحدة الجديدة `mymodule`. يمكنك إنشاء المجلد `projects` إما من خلال بيئة تطوير متكاملة IDE أو من خلال سطر الأوامر.

في حال استخدامك لسطر الأوامر، ابدأ بإنشاء المجلد وانتقل إليه من خلال التعليمات التالية:

```
$ mkdir projects
$ cd projects
```

سننشئ بعد ذلك مجلد الوحدة، إذ يكون عادةً اسم المجلد المتواجد في المستوى الأعلى من الوحدة هو نفس اسم الوحدة لتسهيل تتبع الأشياء.

أنشئ ضمن مجلد `projects` مجلدًا باسم `mymodule`:

```
$ mkdir mymodule
```

بعد إنشاء مجلد الوحدة، ستصبح لدينا البنية التالية:

```
└── projects
    └── mymodule
```

الخطوة التالية هي إنشاء ملف go.mod داخل مجلد `mymodule` كي نُعّرف الوحدة، وسنستخدم لأجل ذلك الأمر `go mod init` ونحدد له اسم الوحدة `mymodule` كما يلي:

```
$ go mod init mymodule
```

يعطي هذا الأمر الخرج التالي عند إنشاء الوحدة:

```
go: creating new go.mod: module mymodule
```

سيصبح لديك الآن بنية المجلد التالية:

```
└── projects
  └── mymodule
    └── go.mod
```

دعنا نلقي نظرةً على ملف go.mod لمعرفة ما فعله الأمر `go mod init`.

30.3 الملف go.mod

يلعب الملف go.mod دورًا مهمًا جدًا عند تشغيل الأوامر باستخدام الأداة `go`، إذ يحتوي على اسم الوحدة وإصدارات الوحدات الأخرى التي تعتمد عليها الوحدة الخاصة بك، ويمكن أن يحتوي أيضًا على موجّهات directives أخرى مثل `replace`، التي يمكن أن تكون مفيدة لإجراء عمليات تطوير على وحدات متعددة في وقت واحد.

افتح الملف go.mod الموجود ضمن المجلد `mymodule` باستخدام محرر نانو nano أو محرر النصوص المفضل لديك:

```
$ nano go.mod
```

ستكون محتوياته هكذا (قليلة أليس كذلك؟):

```
module mymodule
go 1.16
```

في السطر الأول لدينا موجّه يُدعى `module`، مهمته إخبار مُصرّف اللغة عن اسم الوحدة الخاصة بك، بحيث عندما يبحث في مسارات الاستيراد `import` ضمن حزمة فإنه يعرف أين عليه أن يبحث عن `mymodule`. تأتي القيمة `mymodule` من المعامل الذي مررته إلى `go mod init`:

```
module mymodule
```

لدينا في السطر الثاني والأخير من الملف موجّه آخر هو `go`، مهمته إخبار المصرّف عن إصدار اللغة التي تستهدفها الوحدة. بما أننا ننشئ الوحدة باستخدام النسخة 1.16 من لغة جو، فإننا نكتب:

```
go 1.16
```

سيتوسع هذا الملف مع إضافة المزيد من المعلومات إلى الوحدة، ولكن من الجيد إلقاء نظرة عليه الآن لمعرفة كيف يتغير عند إضافة اعتماديات في أوقات لاحقة.

لقد أنشأت الآن وحدة جو باستخدام `go mod init` واطلعت على ما يحتويه ملف go.mod، لكن ليس لهذه الوحدة أي وظيفة تفعلها حتى الآن.

سنبدأ في الخطوة التالية بتطوير هذه الوحدة وإضافة بعض الوظائف إليها.

30.4 إضافة شيفرات برمجية إلى الوحدة

ستنشئ ملف main.go داخل المجلد `mymodule` لنضع فيه شيفرة برمجية ونشغلها ونختبر صحة عمل الوحدة. يُعد استخدام ملف main.go في برامج هذه اللغة للإشارة إلى نقطة بداية البرنامج أمرًا شائعًا.

الأهم من هذا الملف هو الدالة `main()` التي سنكتبها بداخله، أما اسم الملف بحد ذاته فهو اختياري، لكن من الأفضل تسميته بهذا الاسم لجعل عملية العثور عليه أسهل. سنجعل الدالة `main` تطبع رسالةً ترحيبيةً عند تشغيل البرنامج.

نُنشئ الآن ملفًا باسم main.go باستخدام محرّر نانو أو أي محرر نصوص مفضل لديك:

```
$ nano main.go
```

ضع التعليمات البرمجية التالية في ملف main.go، إذ تصرّح هذه التعليمات أولًا عن الحزمة، ثم تستورد الحزمة `fmt`، وتطبع رسالةً ترحيبية:

```
package main
import "fmt"
func main() {
  fmt.Println("Hello, Modules!")
}
```

تُمثّل الحزمة في لغة جو بالمجلد الذي تُنشأ فيه، ولكل ملف داخل مجلد الحزمة سطر يُصرّح فيه عن الحزمة التي ينتمي إليها هذا الملف. أعطينا الحزمة الاسم `main` في ملف main.go الذي أنشأته للتو، ويمكنك تسمية الحزمة بالطريقة التي تريدها، ولكن حزمة `main` خاصة في لغة جو، فعندما يرى مُصرّف اللغة أن الحزمة تحمل اسم `main`، فإنه يعلم أن الحزمة ثنائية binary، ويجب تصريفها إلى ملف تنفيذي، وليست مجرد مكتبة مصممة لاستخدامها في برنامج آخر.

بعد تحديد الحزمة التي يتبع لها الملف، يجب أن نستورد الحزمة `fmt` لكي نستطيع استخدام الدالة `Println` منها، وذلك لطباعة الرسالة الترحيبية "Hello, Modules!" على شاشة الخرج.

أخيرًا، نُعرّف الدالة `main()` التي بدورها لها معنى خاص في لغة جو فهي مرتبطة بحزمة `main`، وعندما يرى المُصرّف دالة اسمها `main()` داخل حزمة باسم `main` سيعرف أن الدالة `main()` هي الدالة الأولى التي يجب تشغيلها، ويُعرف هذا بنقطة دخول البرنامج.

بمجرد إنشاء ملف main.go، ستصبح بنية مجلد الوحدة مشابهة لما يلي:

```
└── projects
  └── mymodule
    └── go.mod
    └── main.go
```

إذا كنت معتادًا على استخدام لغة جو ومتغير البيئة GOPATH، فإن تشغيل شيفرة ضمن وحدة مشابهٌ لكيفية تشغيل شيفرة من مجلد في GOPATH. لا تقلق إذا لم تكن معتادًا على GOPATH، لأن استخدام الوحدات يحل محل استخدامها.

هناك طريقتان شائعتان لتشغيل برنامج تنفيذي في لغة جو، هما: إنشاء ملف ثنائي باستخدام go build وتشغيله، أو تشغيل الملف مباشرةً باستخدام go run. ستستخدم هنا go run لتشغيل الوحدة مباشرةً، إذ أن الأخير يجب أن يُشغّل لوحده.

شغّل ملف main.go الذي أنشأته باستخدام الأمر go run:

```
$ go run main.go
```

ستحصل على الخرج التالي:

```
Hello, Modules!
```

أضفنا في هذا القسم الملف main.go إلى الوحدة التي أنشأناها وعرّفنا فيها نقطة دخول متمثلة بالدالة main وجعلناها تطبع عبارة ترحيبية.

لم نستفد حتى الآن من خصائص أو فوائد الوحدات في لغة جو؛ إذ لا تختلف الوحدة التي أنشأناها للتو عن أي ملف يُشغّل باستخدام go run، وأول فائدة حقيقية لوحدات جو هي القدرة على إضافة الاعتماديات إلى مشروعك في أي مجلد وليس فقط ضمن مجلد GOPATH، كما يمكنك أيضًا إضافة حزم إلى وحدتك.

سنوسّع في القسم التالي هذه الوحدة عن طريق إنشاء حزمة إضافية داخلها.

30.5 إضافة حزمة إلى الوحدة

قد تحتوي الوحدة على أي عدد من الحزم والحزم الفرعية sub-packages على غرار حزمة جو القياسية، وقد لا تحتوي على أية حزم. سننشئ الآن حزمة باسم mypackage داخل المجلد mymodule. سنستخدم كما جرت العادة الأمر mkdir ونمرر له الوسيط mypackage لإنشاء مجلد جديد داخل المجلد mymodule:

```
$ mkdir mypackage
```

سيؤدي هذا إلى إنشاء مجلد جديد باسم `mypackage` مثل حزمة فرعية جديدة من المجلد `mymodule`:

```
└── projects
  └── mymodule
    └── mypackage
    └── main.go
    └── go.mod
```

استخدم الأمر `cd` للانتقال إلى المجلد `mypackage`، ثم استخدم محرر نانو أو محرر النصوص المفضل لديك لإنشاء ملف mypackage.go. يمكن أن يكون لهذا الملف أي اسم، ولكن استخدم نفس اسم الحزمة لتسهيل عملية العثور على الملف الأساسي للحزمة:

```
$ cd mypackage
$ nano mypackage.go
```

أضِف داخل الملف mypackage.go الشيفرة التالية التي تتضمن دالةً تدعى `PrintHello` تطبع عبارة "Hello, Modules! This is mypackage speaking!" عند استدعائها.

```
package mypackage
import "fmt"
func PrintHello() {
  fmt.Println("Hello, Modules! This is mypackage speaking!")
}
```

نحن نريد أن تكون الدالة `PrintHello` متاحةً للحزم والبرامج الأخرى، لذا جعلناها دالةً مُصدّرةً exported function بكتابتنا لأول حرف منها بالحالة الكبيرة P.

بعد أن أنشأنا الحزمة `mypackage` ووضعنا فيها دالةً مُصدَّرةً، سنحتاج الآن إلى استيرادها من حزمة `mymodule` لاستخدامها. هذا مشابه لكيفية استيراد حزم أخرى مثل حزمة `fmt`، لكن هنا يجب أن نضع اسم الوحدة قبل اسم الحزمة `mymodule/mypackage`.

افتح ملف main.go من مجلد `mymodule` واستدعِ الدالة `PrintHello` كما يلي:

```
package main
import (
  "fmt"
  "mymodule/mypackage"
)
func main() {
```

```
    fmt.Println("Hello, Modules!")
    mypackage.PrintHello()
}
```

كما ذكرنا قبل قليل، عند استيراد الحزمة `mypackage` نضع اسم الوحدة قبلها مع وضع الفاصل `/` بينهما، وهو نفسه اسم الوحدة الذي وضعته في ملف go.mod (أي `mymodule`).

```
"mymodule/mypackage"
```

إذا أضفت لاحقًا حزمًا أخرى داخل `mypackage`، يمكنك استيرادها بطريقة مماثلة. على سبيل المثال، إذا كان لديك حزمة أخرى تسمى `extrapackage` داخل `mypackage`، فسيكون مسار استيراد هذه الحزمة هو `mymodule/mypackage/extrapackage`.

شغّل الوحدة مرةً أخرى بعد إجراء التعديلات عليها باستخدام الأمر `go run` ومرر اسم الملف main.go الموجود ضمن المجلد `mymodule` كما في السابق:

```
$ go run main.go
```

عند التشغيل سترى نفس الرسالة التي حصلنا عليها سابقًا والمتمثلة بالرسالة `Hello, Modules!` إضافةً إلى الرسالة المطبوعة من دالة `PrintHello` والموجودة في الحزمة الجديدة التي أضفناها:

```
Hello, Modules!
Hello, Modules! This is mypackage speaking!
```

لقد أضفت الآن حزمةً جديدة إلى وحدتك عن طريق إنشاء مجلد يسمى `mypackage` مع دالة `PrintHello`. يمكنك لاحقًا توسيع هذه الوحدة بإضافة حزم ودوال جديدة إليها، كما سيكون من الجيد تضمين وحدات أنشأها أشخاص آخرون في وحدتك.

سنضيف في القسم التالي وحدةً بعيدة (من جيت Git) مثل اعتمادية لوحدتك.

30.6 تضمين وحدة بعيدة أنشأها آخرون في وحدتك

تُوزَّع وحدات لغة جو من مستودعات التحكم بالإصدار Version Control Repositories -أو اختصارًا VCSs- وأكثرها شيوعًا جيت git. عندما ترغب بإضافة وحدة جديدة مثل اعتمادية لوحدتك، تستخدم مسار المستودع للإشارة إلى الوحدة التي ترغب باستخدامها. عندما يرى مُصرّف اللغة مسار الاستيراد لهذه الوحدة، سيكون قادرًا على استنتاج مكان وجود هذه الوحدة اعتمادًا على مسار المستودع.

سنضيف في المثال التالي المكتبة cobra مثل اعتمادية للوحدة التي أنشأناها، وهي مكتبة شائعة لإنشاء تطبيقات الطرفية Console.

بطريقة مشابهة لما فعلناه عند إنشاء الوحدة `mymodule` سنستخدم الأداة `go` مرةً أخرى، لكن سنضيف لها `get` أي سنستخدم الأمر `go get`، وسنضع بعده مسار المستودع.

شغّل الأمر `go get` من داخل مجلد `mymodule`:

```
$ go get github.com/spf13/cobra
```

عند تشغيل هذا الأمر ستبحث أداة `go` عن مستودع Cobra من المسار الذي حددته، وستبحث عن أحدث إصدار منها من خلال البحث في الفروع ووسوم المستودع. بعد ذلك، سيُحمّل هذا الإصدار ويبدأ في تعقبها، وذلك من خلال إضافة اسم الوحدة والإصدار إلى ملف go.mod للرجوع إليه مستقبلًا.

افتح ملف go.mod الموجود في المجلد `mymodule` لترى كيف حدّثت أداة `go` ملف go.mod عند إضافة الاعتمادية الجديدة. قد يتغير المثال أدناه اعتمادًا على الإصدار الحالي من Cobra أو إصدار أداة `go` التي تستخدمها، ولكن يجب أن تكون البنية العامة للتغييرات متشابهة:

```
module mymodule
go 1.16
require (
  github.com/inconshreveable/mousetrap v1.0.0 // indirect
  github.com/spf13/cobra v1.2.1 // indirect
  github.com/spf13/pflag v1.0.5 // indirect
)
```

لاحظ إضافة قسم جديد مُتمثّل بالموجّه `require`، ومهمته إخبار مُصرّف اللغة عن الوحدة التي تريدها (في حالتنا هي `github.com/spf13/cobra`) وعن إصدارها أيضًا. نلاحظ أيضًا أن هناك تعليق مكتوب فيه `indirect`، ليوضح أنه في الوقت الذي أُضيف فيه الموجّه `require`، لم تكن هناك إشارة مباشرة إلى الوحدة في أي من ملفات المصدر الخاصة بالوحدة. نلاحظ أيضًا وجود بعض الأسطر في `require` تشير إلى وحدات أخرى تتطلبها المكتبة Cobra والتي يجب على أداة جو أن تشير إليها أيضًا.

نلاحظ أيضًا أنه عند تشغيل الأمر `go run` أُنشئ ملف جديد باسم go.sum في مجلد `mymodule`، وهو ملفٌ آخر مهم لوحدات جو يحتوي على معلومات تستخدمها جو لتسجيل قيم معمّاة hashes وإصدارات معينة من الاعتماديات. يضمن هذا تناسق الاعتماديات، حتى لو ثُبتت على جهاز مختلف.

ستحتاج أيضًا إلى تحديث الملف main.go بعد تنزيل الاعتمادية، وذلك من خلال إضافة بعض التعليمات البرمجية الخاصة بها، لكي تتمكن من استخدامها.

افتح الملف main.go الموجود في المجلد `mymodule` وضع فيه المحتويات التالية:

```
package main
import (
  "fmt"
  "github.com/spf13/cobra"
  "mymodule/mypackage"
)
func main() {
  cmd := &cobra.Command{
        Run: func(cmd *cobra.Command, args []string) {
              fmt.Println("Hello, Modules!")
              mypackage.PrintHello()
        },
  }
  fmt.Println("Calling cmd.Execute()!")
  cmd.Execute()
}
```

تُنشئ هذه الشيفرة بنية `cobra.Command` مع دالة `Run` تطبع عبارة ترحيب، والتي ستُنفّذ بعد ذلك باستدعاء `cmd.Execute()`.

شغّل الشيفرة الآن بعد أن عدلناها:

```
$ go run main.go
```

سترى الخرج التالي الذي يبدو مشابهًا لما رأيته من قبل، لكننا استخدمنا هنا الاعتمادية الجديدة التي أضفناها كما هو موضح في السطر الأول:

```
Calling cmd.Execute()!
Hello, Modules!
Hello, Modules! This is mypackage speaking!
```

يؤدي استخدام الأمر `go get` إلى إضافة اعتماديات إلى وحدتك إلى تنزيل أحدث إصدار من هذه الاعتمادية، وهو أمرٌ جيد لأن أحدث إصدار يتضمن كل التعديلات الجديدة والإصلاحات. قد نرغب أحيانًا في استخدام إصدار أقدم أو فرع مُحدد من مستودع هذه الاعتمادية. وسنستخدم في القسم التالي الأداة `go get` لمعالجة هذه الحالة.

30.7 استخدام إصدار محدد من وحدة

بما أن وحدات لغة جو توزّع من مستودع التحكم في الإصدار، بالتالي يمكنها استخدام ميزاته، مثل الوسوم والفروع والإيداعات. يمكنك الإشارة إلى هذه الأمور في اعتمادياتك باستخدام الرمز @ في نهاية مسار الوحدة جنبًا إلى جنب مع الإصدار الذي ترغب في استخدامه. لاحظ أنه كان بإمكانك استخدام هذه الميزة مباشرةً في المثال السابق، فهي لا تتطلب شيئًا إلا وضع الرمز والإصدار. إذًا يمكننا استنتاج أنه في حال عدم استخدام هذا الرمز، يفترض جو أننا نريد أحدث إصدار، وهذا يُقابل وضع `latest` بعد هذا الرمز. `latest` هي ميزة خاصة لأداة جو، وليست جزءًا من الوحدة أو مستودع الوحدة التي تريد استخدامها مثل `my-tag` أو `my-branch`. إذًا، تُكافئ الصيغة التالية تنزيل أحدث إصدار من الوحدة سالفة الذكر:

```
$ go get github.com/spf13/cobra@latest
```

تخيل الآن أن هناك وحدة تستخدمها، وهي قيد التطوير حاليًا، ولنفترض اسمها `your_domain/sammy/awesome`. افترض أن هناك ميزة أضيفت إلى هذه الوحدة في فرع اسمه `new-feature`. لإضافة هذا الفرع مثل اعتمادية للوحدة الخاصة بك، يمكنك ببساطة استخدام مسار الوحدة متبوعًا بالرمز @ متبوعًا باسم الفرع:

```
$ go get your_domain/sammy/awesome@new-feature
```

عند تشغيل هذا الأمر ستتصل أداة `go` بالمستودع `your_domain/sammy/awesome`، وستنزّل الفرع `new-features` من آخر إيداع، وتضيف هذه المعلومات إلى الملف go.mod.

ليست الفروع الطريقة الوحيدة التي يمكنك من خلالها استخدام الرمز @، ولكن يمكنك استخدامه مع الوسوم أو الإيداعات أيضًا، فقد يكون أحيانًا أحدث إصدار من المكتبة التي تستخدمها إيداعًا معطلًا، وفي هذه الحالة يمكنك الرجوع إلى إيداع سابق واستخدامه.

بالعودة إلى نفس المثال السابق، افترض أنك بحاجة إلى الإشارة إلى الإيداع `07445ea` من `github.com/spf13/cobra` لأنه يحتوي على بعض التغييرات التي تحتاجها ولا يمكنك استخدام إصدار آخر لسبب ما. في هذه الحالة، يمكنك وضع قيمة معمّاة hash بعد الرمز @.

شغّل الآن الأمر `go get` من داخل مجلد `mymodule` لتنزيل الإصدار الجديد:

```
$ go get github.com/spf13/cobra@07445ea
```

إذا فتحت الآن ملف go.mod الخاص بالوحدة، فسترى أن `go get` قد حدّث سطر `require` للإشارة إلى الإيداع الذي تستخدمه:

```
module mymodule
go 1.16
require (
  github.com/inconshreveable/mousetrap v1.0.0 // indirect
  github.com/spf13/cobra v1.1.2-0.20210209210842-07445ea179fc //
indirect
  github.com/spf13/pflag v1.0.5 // indirect
)
```

على عكس الوسوم أو الفروع، ونظرًا لأن الإيداع يمثّل نقطةً زمنيةً معينة، تتضمن جو معلومات إضـافية في موجّه `require` للتأكد من استخدام الإصدار الصحيح مستقبلًا؛ فـإذا نظـرت إلى الإصـدار، سـترى أنـه يتضـمن إيداعًا معمّى hash قد أضفته، أي `v1.1.2-0.20210209210842-07445ea179fc`.

تستخدم جو هذه الوظيفة لدعم إطلاق عدة إصدارات من الوحدة، فعنـدما تطلـق وحـدة جـو إصـدارًا جديـدًا، سيُضاف وسمًا جديدًا إلى المستودع مع وسم لرقم الإصدار. إذا كنت تريد استخدام إصدار محدد، يمكنـك النظـر إلى قائمة الوسوم في المستودع حـتى تجـد الإصـدار الـذي تبحث عنـه، أمـا إذا كنت تعـرف الإصـدار، لن تكـون مضطرًا للبحث ضمن قائمة الوسوم لأنها مرتبة بانتظام.

بالعودة إلى نفس المثال Cobra السابق، ولنفترض أنك بحاجـة إلى اسـتخدام الإصـدار 1.1.1، الـذي يملـك وسمًا يدعى `v1.1.1` في مستودع Cobra.

سنستخدم الأمر `go get` مع الرمز `@` حتى نتمكن من استخدام هذا الإصدار الموسوم تمامًا كمـا فعلنـا مـع الفروع أو الوسوم التي لا تشير لإصدار non-version tag. الآن، حدّث الوحدة الخاصة بك حتى تستخدم Cobra 1.1.1 من خلال تشغيل الأمر `go get` مع رقم الإصدار `v1.1.1`:

```
$ go get github.com/spf13/cobra@v1.1.1
```

إذا فتحت الآن ملف go.mod الخاص بالوحدة، فسترى أن `go get` قد حدّث سطر`require` للإشارة إلى الإيداع الذي تستخدمه:

```
module mymodule
go 1.16
require (
  github.com/inconshreveable/mousetrap v1.0.0 // indirect
  github.com/spf13/cobra v1.1.1 // indirect
  github.com/spf13/pflag v1.0.5 // indirect
)
```

أخيرًا، إذا كنت تستخدم إصدارًا محددًا من المكتبة، مثل الإيداع `07445ea` أو الإصدار `v1.1.1` الذين تعرفنا عليهما منذ قليل، ثم قررت استخدام أحدث إصدار من المكتبة، فمن الممكن إنجاز ذلك باستخدام `latest` كما ذكرنا سابقًا.

لتحديث الوحدة الخاصة بك إلى أحدث إصدار من Cobra، استخدم الأمر `go get` مرةً أخرى مع مسار الوحدة وتحديد الإصدار `latest` بعد الرمز `@`:

```
$ go get github.com/spf13/cobra@latest
```

بعد تنفيذ هذا الأمر، يُحدّث ملف go.mod ليبدو كما كان عليه عند تنزيل المكتبة Cobra أول مرة. قد يبدو الخرج مختلفًا قليلًا بحسب إصدار جو الذي تستخدمه والإصدار الأخير من Cobra ولكن يُفترض أن تلاحظ أن السطر `github.com/spf13/cobra` في قسم `require` قد تحدّث إلى آخر إصدار:

```
module mymodule
go 1.16
require (
  github.com/inconshreveable/mousetrap v1.0.0 // indirect
  github.com/spf13/cobra v1.2.1 // indirect
  github.com/spf13/pflag v1.0.5 // indirect
)
```

يُعد الأمر `go get` أداةً قوية يمكنك استخدامها لإدارة الاعتماديات في ملف go.mod دون الحاجة إلى تحريره يدويًا. يتيح لك استخدام الرمز `@` مع اسم الوحدة، أن تستخدم إصدارات معينة لوحدة ما، أو حتى إيداعات، أو فروع، أو الرجوع إلى أحدث إصدار من اعتمادياتك كما رأينا للتو. سيسمح استخدام مجموعة الأدوات التي تعرفنا عليها في هذا الفصل بضمان استقرار البرنامج مستقبلًا.

30.8 الخاتمة

تعلمت في هذا الفصل كيفية إنشاء وحدةً برمجية باستخدام لغة جو مع حزمة فرعية، ثم تعرفت على طريقة استخدام تلك الحزمة داخل الوحدة التي أنشأتها. ثم تعرفت على طريقة إضافة وحدةً خارجية إليها مثل اعتمادية، وكيفية الإشارة إلى إصدارات الوحدة بطرق مختلفة، وستتعلم في الفصل التالي كيفية توزيع مكتبات جاهزة وإتاحتها للآخرين لاستخدامها في برامجهم المكتوبة بلغة البرمجة جو.

دورة تطوير تطبيقات الويب
باستخدام لغة Ruby

دورة تدريبية متكاملة من الصفر وحتى الاحتراف
تمكنك من التخصص في هندسة الويب ودخول سوق العمل

التحق بالدورة الآن

31. توزيع الوحدات Modules المكتوبة

تسمح العديد من لغات البرمجة الحديثة -بما في ذلك لغة جو- للمطورين بتوزيع مكتبات جاهزة للآخرين لاستخدامها في برامجهم. تستخدم بعض اللغات مستودعًا مركزيًا لتثبيت هذه المكتبات، بينما توزعها لغة جو من نفس مستودع التحكم في الإصدار version control repository المستخدم لإنشاء المكتبات. تستخدم لغة جو أيضًا نظام إصدار يسمى الإصدار الدلالي Semantic Versioning، ليوضح للمستخدمين متى وما هو نوع التغييرات التي أُجريت. يساعد ذلك المستخدمين على معرفة ما إذا كان الإصدار الأحدث من الوحدة آمنًا للترقية، وما إذا كان يساعد في ضمان استمرار عمل برامجهم مع الوحدة.

سننشئ في هذا الفصل وحدةً برمجيةً جديدة باستخدام لغة جو وسننشرها، وسنتعلم استخدام الإصدار الدلالي، وسننشر إصدارًا دلاليًا من الوحدة التي أنشأناها.

31.1 المتطلبات

- ستحتاج لامتلاك مساحة عمل خاصة مُهيئة وجاهزة في لغة جو، فإذا لم يكن لديك واحدة، فارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبت لغة جو Go وقم بإعداد بيئة تطوير محلية بحسب نظام تشغيلك.
- يُفضّل أن تكون على دراية بكيفية إنشاء الوحدات في لغو جو. يمكنك مراجعة فصل استخدام الوحدات Modules في لغة جو Go.
- لديك معرفة مسبقة بنظام التحكم بالإصدار جيت Git. يمكنك مراجعة مقالة ما هو جيت Git؟ على أكاديمية حسوب.
- إنشاء مستودع فارغ باسم pubmodule للوحدة التي ستنشرها.

31.2 إنشاء وحدة للتحضير لنشرها

على عكس العديد من لغات البرمجة الأخرى، توزّع الوحدة المُنشأة باستخدام لغة جو مباشـرةً من مسـتودع الشيفرة المصدر الذي يتضـمنها بـدلًا من مسـتودع حـزم مسـتقل. يسـهل هـذا على المسـتخدمين العثـور على الوحدات المشار إليها في التعليمات البرمجية الخاصة بهم وعلى المشرفين على الوحدة لنشـر إصـدارات جديـدة من الوحدة الخاصة بهم. سننشئ في هذا القسم وحدةً جديدة، وسننشرها بعد ذلك لجعلها متاحـةً للمسـتخدمين الآخرين. لنبدأ بإنشاء الوحدة الآن.

سنستخدم بدايةً الأمر `git clone` لأخذ نسخة محلية من المستودع الفارغ الـذي لا بُـد أن نكـون أنشـأناه لأنه جزء من المتطلبات الأساسية. يمكن نسخ هذا المستودع إلى أي مكان نريده على جهاز الحاسب، لكن يميـل العديد من المطورين إلى إنشاء مجلد خاص يتضمن جميع مشاريعهم. سنستخدم هنا مجلدًا باسم `projects`.

أنشئ مجلد `projects` وانتقل إليه:

```
$ mkdir projects
$ cd projects
```

من مجلد `projects`، شغل الأمر `git clone` لنسخ المستودع إلى جهاز الحاسب:

```
$ git clone git@github.com:your_github_username/pubmodule.git
```

ستكون نتيجة تنفيذ هذا الأمر هي نسخ الوحدة إلى مجلد `pubmodule` داخل مجلد `projects`. قـد تتلقى تحذيرًا بأنك نسخت مستودعًا فارغًا، ولكن لا داعٍ للقلق بشأن ذلك:

```
Cloning into 'pubmodule'...
warning: You appear to have cloned an empty repository.
```

انتقل الآن إلى المجلد `pubmodule`:

```
$ cd pubmodule
```

سنستخدم الآن الأمر `go mod init` لإنشاء الوحدة الجديدة وتمرير موقع المستودع اسـمًا للوحـدة. يُعـد التأكد من تطابق اسم الوحدة مع موقع المستودع أمرًا مهمًا، لأن هذه هي الطريقة التي تعـثر بهـا أداة `go mod init` على مكان تنزيل الوحدة عند استخدامها في مشاريع أخرى:

```
$ go mod init github.com/your_github_username/pubmodule
```

ستظهر رسالة تؤكد عملية إنشاء الوحدة من خلال إعلامنا بأن الملف go.mod قد أُنشئ:

```
go: creating new go.mod: module
github.com/your_github_username/pubmodule
```

سنستخدم الآن محرر نصوص برمجية مثل نانو nano،من أجل إنشاء وفتح ملف يحمل نفس اسم المستودع pubmodule.go.

```
$ nano pubmodule.go
```

يمكن تسمية هذا الملف بأي اسم، ولكن يُفضّل استخدام نفس اسم الحزمة لتسهيل معرفة مكان البدء عند العمل مع حزمة غير مألوفة، كما ينبغي أن يكون اسم الحزمة هو نفس اسم المستودع. بالتالي، عندما يُشير شخص ما إلى تابع أو نوع من تلك الحزمة، فإنها تتطابق مع اسم المستودع، مثل `pubmodule.MyFunction`. سيسهل ذلك عليهم معرفة مصدر الحزمة في حال احتاجوا للرجوع إليها لاحقًا.

نُضيف الآن الدالة `Hello()` إلى الحزمة، والتي ستعيد السلسلة `Hello, You!`. ستكون هذه الدالة متاحةً لأي شخص يستورد الحزمة:

```
package pubmodule
func Hello() string {
 return "Hello, You!"
}
```

لقد أنشأنا الآن وحدةً جديدةً باستخدام `go mod init` مع اسم وحدة يتطابق مع اسم المستودع البعيد `github.com/your_github_username/pubmodule`، وأضفنا أيضًا ملفًا باسم pubmodule.go إلى الوحدة، مع دالة تُسمى `Hello` يمكن استدعاؤها من قِبل مستخدمي هذه الوحدة. سننشر في الخطوة التالية هذه الوحدة بهدف إتاحتها للآخرين.

31.3 نشر الوحدة

لقد حان الوقت لنشر الوحدة التي أنشأناها في الخطوة السابقة. نظرًا لأن وحدات لغة جو تُوزّع من نفس مستودعات الشيفرة التي تُخزّن فيها، فسوف نُودع commit الشيفرة في مستودع جيت المحلي وندفعه push إلى المستودع الخاص بنا على `github.com/your_github_username/pubmodule`، ولكن قبل ذلك من الجيد التأكد من عدم إيداع ملفات لا نريد إيداعها عن طريق الخطأ أو عدم الانتباه، لأنها ستُنشر علنًا عند دفع الشيفرة إلى جيت هب. يمكن إظهار جميع الملفات داخل مجلد `pubmodule` والتغييرات التي ستُنفّذ باستخدام الأمر التالي:

```
$ git status
```

سيبدو الخرج كما يلي:

```
On branch main
No commits yet
Untracked files:
 (use "git add <file>..." to include in what will be committed)
  go.mod
  pubmodule.go
```

يجب أن نرى ملف go.mod الذي أُنشئ بواسطة الأمر `go mod init`، وملـف pubmodule.go الـذي أنشأنا فيه دالة `Hello()`. قد يكون اسم الفرع مختلفًا عن الخرج السابق اعتمادًا على كيفية إنشـاء المسـتودع، وستكون الأسماء غالبًا إما `main` أو `master`.

عندما نتأكد من أن الملفات التي نريدها فقط موجودة، يمكننا إدراج stage الملفات باسـتخدام `git add` وإيداعها بالمستودع باستخدام الأمر `git commit`:

```
$ git add .
$ git commit -m "Initial Commit"
```

سيكون الخرج كما يلي:

```
[main (root-commit) 931071d] Initial Commit

 2 files changed, 8 insertions(+)
 create mode 100644 go.mod
 create mode 100644 pubmodule.go
```

نستخدم الأمر `git push` لدفع الوحدة إلى مستودع جيت هب:

```
$ git push
```

سيكون الخرج كما يلي:

```
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 367 bytes | 367.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:your_github_username/pubmodule.git
 \* [new branch]   main -> main
```

ستُدفع الوحدة إلى المستودع بعد تشغيل الأمر `git push`، وستكون متاحةً الآن لأي شخص آخر لاستخدامها. ستستخدم جو الشيفرة الموجودة في الفرع الافتراضي للمستودع على أنها شيفرة للوحدة إذا لم يكن لديك أيّة إصدارات منشورة. لا يهم ما إذا كان اسم الفرع الافتراضي هو `main` أو `master` أو أي شيء آخر، المهم هو الفرع الذي ضُبط على أنه فرع افتراضي.

تعلّمنا في هذا القسم كيفية نشر وحدة في لغة جو على مستودع جيت هب من المستودع المحلي لإتاحتها للآخرين. واحدة من الأمور الأساسية التي يجب أخذها بالحسبان، هي التأكد من أن مستخدمي الوحدة يمكنهم استخدام إصدار مستقر منها.

قد نرغب في إجراء تغييرات وإضافة ميزات إلى الوحدة من الآن فصاعدًا، ولكن إذا أجرينا هذه التغييرات دون استخدام مفهوم الإصدارات، فقد تتعطل شيفرة شخص ما يستخدم هذه الوحدة عن طريق الخطأ.

لحل هذه المشكلة يمكننا إضافة إصدارات إلى الوحدة عندما نصل إلى مرحلة جديدة في عملية التطوير، ومع ذلك، عند إضافة إصدارات جديدة يجب التأكد من اختيار رقم إصدار مفيد حتى يعرف المستخدمون ما إذا كان من الآمن لهم الترقية فورًا أم لا.

31.4 الإصدار الدلالي

يعطي رقم الإصدار التعبيري meaningful version number للمستخدمين فكرةً عن مدى تغير الواجهة العامة أو واجهة برمجة التطبيقات التي يتفاعلون معها. تنقل جو هذه التغييرات من خلال نظام أو مخطط إصدار versioning scheme يُعرف باسم الإصدار الدلالي Semantic Versioning أو SemVer اختصارًا. يستخدم الإصدار الدلالي سلسلة الإصدار لنقل المعنى حول تغييرات الشيفرة، ومن هنا يأتي اسم الإصدار الدلالي. يُستخدم الإصدار الدلالي لتحديد الإصدارات الأحدث من الإصدار الحالي المُستخدم، وما إذا كان بالإمكان الترقية للإصدار الأحدث آليًا بأمان.

يعطي الإصدار الدلالي كل رقم في سلسلة الإصدار معنى meaning. ويحتوي الإصدار النموذجي في SemVer على ثلاثة أرقام أساسية: الإصدار الرئيسي major والإصدار الثانوي minor وإصدار التصحيح patch version. تُدمج كل من هذه الأرقام مع بعضها بواسطة النقطة لتشكيل سلسلة الإصدار، مثلًا `1.2.3`، وتُرتّب فيه الأرقام وفقًا للآلية التالية: أول رقم للإصدار الرئيسي ثم الإصدار الثانوي ثم إصدار التصحيح أخيرًا.

يمكنك بهذه الطريقة معرفة أي إصدار هو الأحدث لأن الرقم الموجود في مكان معين أعلى من الإصدارات السابقة. على سبيل المثال، الإصدار `2.2.3` أحدث من `1.2.3` لأن الإصدار الرئيسي أعلى، وبالمثل، فإن الإصدار `1.4.3` أحدث من `1.2.10` لأن الإصدار الثانوي أعلى، وعلى الرغم من أن `10` أعلى من `3` في إصدار التصحيح، فإن الإصدار الثانوي `4` أعلى من `2`، لذا فإن هذا الإصدار له الأسبقية.

وعندما يزداد رقم في سلسلة الإصدار، سوف يُعاد ضبط جميع الأجزاء الأخرى من الإصدار التي تليهـا إلى 0. على سبيل المثال، ستؤدي زيادة الإصدار الثانوي من 1.3.10 إلى 1.4.0 وزيادة الإصدار الرئيسـي من 2.4.1 إلى 3.0.0.

يسمح استخدام هذه القواعد للغة جو بتحديد إصدار الوحدة التي يجب اسـتخدامها عنـد تشـغيل go get. على سـبيل المثـال، إذا افترضـنا أن لـدينا مشـروعًا يسـتخدم الإصـدار 1.4.3 من الوحـدة التاليـة github.com/your_github_username/pubmodule. وكنـا نعتمـد على كـون الوحـدة pubmodule مستقرة، فقد نرغب فقط في ترقية إصدار التصحيح 3. تلقائيًا .

إذا شغّلت الأمر:

```
go get -u=patch github.com/your_github_username/pubmodule
```

ستستنتج لغو جو أننا تريد ترقية إصدار التصحيح للوحدة، وستبحث فقط عن الإصدارات الجديـدة مـع 1.4 للجزء الرئيسي والثانوي من الإصدار.

من المهم أن نضع في الحسبان كيف تغيرت واجهة برمجة التطبيقات العامة للوحدة عند إنشاء إصدار جديد منها، إذ ينقل لنـا كـل جـزء من سلسـلة الإصـدار الدلاليـة نطـاق التغيـير في واجهـة برمجـة التطبيقـات وكـذلك للمستخدمين. تنقسم هذه الأنـواع من التغيـيرات عـادةً إلى ثلاث فئـات مختلفـة، تتماشـى مـع كـل مكـون من مكونات الإصدار؛ إذ تؤدي التغيرات الأصغر إلى زيـادة إصـدار التصـحيح، وتزيـد التغيـيرات متوسـطة الحجم من الإصدار الثانوي، وتزيد التغييرات الأكبر في الإصدار الرئيسي. سيساعدنا اسـتخدام هـذه الفئـات في تحديـد رقم الإصدار المراد زيادته وتجنب تعطل break الشيفرة الخاصة بنا أو شيفرة أي شخص آخر يعتمد على هذه الوحدة.

31.5 أرقام الإصدارات الرئيسية

الرقم الأول في SemVer هو رقم الإصدار الرئيسي (1.4.3)، وهو أهم رقم يجب مراعاته عند إطلاق إصـدار جديد من الوحدة. في هذا النوع من الإصدارات يكون هناك تغيّر كبير في الإصدار يُشير إلى تغيرات غير متوافقـة backward-incompatible changes مع الإصدارات السابقة لواجهة برمجة التطبيقـات العامـة المسـتخدمة API؛ وقد يكون التغير غير المتوافق مع الإصدارات السابقة هو أي تغير يطـرأ على الوحـدة من شـأنه أن يتسـبب في تعطل breaking برنامج شخص ما إذا أجرى الترقية دون إجراء أي تغييرات أخـرى. يمكن أن يمثّـل التعطّـل أي حالة فشل في البناء بسبب تغيير اسم دالة أو تغيير في كيفية عمل المكتبة، بحيث أن تابعًا أصبح يُعيد "v1" بدلًا من "1". هذا فقط من أجل واجهـة برمجـة التطبيقـات العامـة الخاصـة بنـا، ومـع ذلـك يمكن لشـخص آخـر استخدام أي أنواع أو توابع قد صُدّرت.

إذا كان الإصدار يتضمن فقط تحسينات لن يلاحظها مستخدم المكتبة، فلا يحتاج إلى تغيير كبير في الإصدار. قد تكون إحدى الطرق لتذكر التغييرات التي تناسب هذه الفئة هي أخذ كل "تحديث" أو "حذف" بمثابة زيادة كبيرة في الإصدار.

> على عكس الأنواع الأخرى من الأرقام في SemVer، فإن الإصدار الرئيسي 0 له أهمية خاصة إضافية، إذ يُعد إصدارًا "قيد التطوير in development". لا يُعد أي SemVer بإصدار رئيسي 0 مستقرًا، وأي شيء يمكن أن يتغير في واجهة برمجة التطبيقات في أي وقت. يُفضّل أن نبدأ دومًا بالإصدار الرئيسي 0 عند إنشاء وحدة جديدة، وتحديث الإصدارات الثانوية والإصدارات التصحيحية فقط حتى ننتهي من التطوير الأولي للوحدة. بعد الانتهاء من تغيير واجهة برمجة التطبيقات العامة للوحدة وعدّها مستقرةً للمستخدمين، حان الوقت لبدء الإصدار 1.0.0.

لنأخذ الشيفرة التالية مثالًا على الشكل الذي قد يبدو عليه تغيير الإصدار الرئيسي. لدينا دالة تسمى `UserAddress` تقبل حاليًا `string` معاملًا وتعيد `string`:

```
func UserAddress(username string) string {
  // تُعيد عنوان المستخدم مثل سلسلة
}
```

تُعيد الدالة حاليًا سلسلة، وقد يكون من الأفضل لنا وللمستخدمين إذا أعادت الدالة السابقة بنيةً struct مثل `*Address`، إذ يمكن بهذه الطريقة تضمين بيانات إضافية (مثل الرمز البريدي) وبطريقة منظمة:

```
type Address struct {
  Address  string
  PostalCode string
}
func UserAddress(username string) *Address {
  // تُعيد عنوان المستخدم والرمز البريدي مثل بنية
}
```

قد يكون هذا مثالًا على تغيير رئيسي في الإصدار لأنه سيتطلب من المستخدمين إجراء تغييرات على التعليمات البرمجية الخاصة بهم من أجل استخدامها. سيكون الأمر نفسه صحيحًا إذا قررنا حذف الدالة `UserAddress`، لأن المستخدمين سيحتاجون إلى تحديث التعليمات البرمجية الخاصة بهم لاستخدام البديل.

مثال آخر على تغيير الإصدار الرئيسي هو إضافة معامل جديد إلى دالة `UserAddress` على الرغم من أنها ما زالت تُعيد `string`:

```
func UserAddress(username string, uppercase bool) string {
  // تُعيد عنوان المستخدم مثل سلسلة بأحرف كبيرة إذا كان المعامل المنطقي قيمته true
}
```

نظرًا لأن هذا التغيير يتطلب أيضًا من المستخدمين تحديث التعليمات البرمجية الخاصة بهم إذا كانوا يستخدمون دالة `UserAddress`، فسيتطلب ذلك أيضًا زيادة كبيرة في الإصدار.

لن تكون كل التغييرات التي نجريها على الشيفرة جذرية، إذ سنُجري أحيانًا تغييرات على واجهة برمجة التطبيقات العامة API، بحيث نُضيف دوالًا أو قيمًا جديدة، ولكن هذا لا يغير أيًا من الدوال أو القيم الحالية.

31.6 أرقام الإصدارات الثانوية

الرقم الثاني في إصدار SemVer هو رقم الإصدار الثانوي (`1.4.3`)، وهنا يطرأ تغيير بسيط في الإصدار للإشارة إلى التغييرات المتوافقة مع الإصدارات السابقة لواجهة برمجة التطبيقات العامة الخاصة بنا. سيكون التغيير المتوافق مع الإصدارات السابقة أي تغيير لا يؤثر على التعليمات البرمجية أو المشاريع التي تستخدم الوحدة الحالية. على غرار رقم الإصدار الرئيسي؛ يؤثر هذا فقط على واجهة برمجة التطبيقات العامة. قد تكون إحدى الطرق لتذكر التغييرات التي تناسب هذه الفئة، هي أي شيء يعد "إضافة"، ولكن ليس "تحديثًا".

باستخدام نفس المثال السابق الذي شرحنا فيه رقم الإصدار الرئيسي، تخيل أن لديك تابع باسم `UserAddress` يُعيد سلسلة `string`:

```
func UserAddress(username string) string {
  //تُعيد عنوان المستخدم مثل سلسلة
}
```

هنا بدل أن نقوم بتحديث `UserAddress` بجعله يُعيد `*Address`، سوف نضيف تابعًا جديدًا تمامًا يسمى `UserAddressDetail`:

```
type Address struct {
  Address     string
  PostalCode string
}
func UserAddress(username string) string {
  // تُعيد عنوان المستخدم مثل سلسلة
}
func UserAddressDetail(username string) *Address {
  // تُعيد عنوان المستخدم والرمز البريدي مثل بنية
}
```

لا تتطلب إضافة الدالة الجديدة `UserAddressDetail` إجراء تغييرات من قِبل المستخدمين إذا حدّثوا إلى هذا الإصدار من الوحدة، لذلك ستُعد زيادة بسيطة في رقم الإصدار. يمكن للمستخدمين الاستمرار في

استخدام `UserAddress` وسيحتاجون فقط إلى تحديث التعليمات البرمجية الخاصة بهم إذا كانوا يفضلون تضمين المعلومات الإضافية من `UserAddressDetail`.

من المحتمل ألا تكون تغييرات واجهة برمجة التطبيقات العامة هي المرة الوحيدة التي نُطلق فيها إصدارًا جديدًا من الوحدة. تُعد الأخطاء bugs جزءًا لا مفر منه من تطوير البرامج، ورقم إصدار التصحيح موجود للتستر على تلك الثغرات.

31.7 أرقام إصدارات التصحيح

الرقم الأخير في صيغة SemVer هي إصدار التصحيح، (`1.4.3`). تغيير إصدار التصحيح هو أي تغيير **لا يؤثر** على واجهة برمجة التطبيقات العامة API للوحدة. تكون غالبًا التغييرات التي لا تؤثر على واجهة برمجة التطبيقات العامة للوحدة، أشياءً مثل إصلاحات الأخطاء أو إصلاحات الأمان.

بالعودة إلى الدالة `UserAddress` من الأمثلة السابقة، لنفترض أن إصدارًا من الوحدة يفتقد إلى جزء من العنوان في السلسلة التي تُعيدها الدالة. إذا أطلقنا إصدارًا جديدًا من الوحدة لإصلاح هذا الخطأ، سيؤدي ذلك إلى زيادة إصدار التصحيح فقط، ولن يتضمن الإصدار أيّة تغييرات في كيفية استخدام المستخدم لواجهة برمجة التطبيقات العامة `UserAddress`، وإنما مُجرد تعديلات لضمان صحة البيانات المُعادة.

يُعد اختيار رقم إصدار جديد بعناية طريقة مهمة لكسب ثقة المستخدمين، إذ يُظهر استخدام الإصدار الدلالي للمستخدمين مقدار العمل المطلوب للتحديث إلى إصدار جديد، وبالتأكيد لن تفاجئهم عن طريق الخطأ بتحديث يكسر برنامجهم. بعد التفكير في التغييرات التي أجريناها على الوحدة، وتحديد رقم الإصدار التالي المطلوب استخدامه، يمكننا الآن نشر الإصدار الجديد وإتاحته للمستخدمين.

31.8 نشر إصدار جديد من الوحدة

سنحتاج إلى تحديث الوحدة بالتغييرات التي نخطط لإجرائها قبل نشر أي إصدار جديد من الوحدة. لن نكون قادرين على تحديد أي جزء من الإصدار الدلالي يجب أن يزداد دون أن نُجري تغييرات. بالنسبة للوحدة التي أنشأناها، سنضيف التابع `Goodbye` لاستكمال التابع `Hello`، وبعد ذلك ستنشر هذا الإصدار الجديد للاستخدام.

افتح ملف pubmodule.go وضِف التابع الجديد `Goodbye` إلى واجهة برمجة التطبيقات العامة API:

```
package pubmodule
func Hello() string {
 return "Hello, You!"
}
func Goodbye() string {
```

```
    return "Goodbye for now!"
}
```

سنحتاج الآن إلى التحقق من التغييرات التي يُتوقع أن تُنفّذ عن طريق تنفيذ الأمر التالي:

```
$ git status
```

يوضّح الخرج أن التغيير الوحيد في الوحدة الخاصة بنا، هو التابع الذي أضفناه إلى ملف pubmodule.go:

```
On branch main
Your branch is up to date with 'origin/main'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   pubmodule.go
no changes added to commit (use "git add" and/or "git commit -a")
```

نضيف بعد ذلك التغيير إلى الملفات المُدرجة ونجري التغيير في المستودع المحلي باستخدام `git add` و`git commit`:

```
$ git add .
$ git commit -m "Add Goodbye method"
```

سيكون الخرج كما يلي:

```
[main 3235010] Add Goodbye method
 1 file changed, 4 insertions(+)
```

سنحتاج إلى دفع التغييرات بعد تنفيذها إلى مستودع جيت هب الخاص بنا، وتكون عادةً هذه الخطوة مختلفة قليلًا عند العمل على مشروع برمجي أكبر أو عند العمل مع مطورين آخرين في مشروع. عند إجراء تطوير على ميزة جديدة، يُنشئ المطور فرع جيت جديد لإجراء تلك التغييرات وتحضيرها إلى أن تصبح هذه الميزة مستقرة وجاهزة للإصدار، وبعدها سيأتي مطور آخر ويُراجع هذه التغييرات ضمن ذلك الفرع للتأكد أو لاكتشاف المشكلات التي ربما غفل عنها المطور الأول. يُدمج الفرع الخاص بتلك الميزة في الفرع الافتراضي، مثل `master` أو `main` بمجرد الانتهاء من المراجعة، وتُجمع هذه التغييرات في الفرع الافتراضي حتى يحين وقت نشر إصدار جديد.

هنا لا تمر الوحدة بهذه العملية (إضافة ميزة جديدة في فرع آخر)، لذا سيكون دفع التغييرات التي أجريناها على المستودع، هو فقط التغييرات التي أجريناها:

```
$ git push
```

سيكون الخرج على النحو التالي:

```
numerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 369 bytes | 369.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:your_github_username/pubmodule.git
  931071d..3235010 main -> main
```

يوضح الخرج أن الشيفرة الجديدة جاهزة للاستخدام في الفر ع الافتراضي من قبل المستخدمين.

كل ما فعلناه حتى الآن يُطابق ما فعلناه عند نشر الوحدة في البداية. الآن، تأتي نقطة مهمة في عملية إطلاق الإصدارات، وهي اختيار رقم الإصدار.

إذا نظرنا إلى التغييرات التي أجريناها على الوحدة، فإن التغيّر الوحيد على واجهة برمجة التطبيقات العامة (أو أي تغيير) هو إضافة التابع `Goodbye` إلى الوحدة. بما أنه يمكن للمستخدم التحديث من الإصدار السابق الذي كان يحتوي فقط على الدالة `Hello`، دون إجراء تغييرات من جانبهم، سيكون هذا التغيير متوافقًا مع الإصدارات السابقة. قد يعني ذلك (ضمن مفهوم الإصدار الدلالي) التغيير المتوافق مع الإصدارات السابقة لواجهة برمجة التطبيقات العامة زيادةً في رقم الإصدار الثانوي. هذا الإصدار هو الإصدار الأول من الوحدة التي نشرناها، لذلك ليس هناك إصدار سابق لنزيد عليه. عمومًا، إذا كان الإصدار الحالي هو `0.0.0` أي "لا يوجد إصدار"، ستقودنا زيادة الإصدار الثانوي إلى الإصدار `0.1.0`، وهو الإصدار التالي من الوحدة التي أنشأناها.

الآن، بعد أن أصبح لدينا رقم إصدار نُعطيه للوحدة، أصبح بإمكاننا استخدامه مع وسوم جيت لنشر إصدار جديد. عندما يستخدم المطورون جيت لتتبع شيفرة المصدر الخاصة بهم (حتى في لغات أخرى غير جو)، فإن العرف الشائع هو استخدام وسوم جيت لتتبع الشيفرة التي أُطلقت لإصدار معين؛ فبهذه الطريقة يمكنهم استخدام الوسم إذا احتاجوا في أي وقت إلى إجراء تغييرات على إصدار قديم. بما أن لغة جو تُنزّل الوحدات من المستودعات المصدرية، بالتالي تستطيع الاستفادة من هذه الخاصية في استخدام وسوم الإصدار نفسها.

لنشر نسخة جديدة من الوحدة التي أنشأناها باستخدام هذه الوسوم، سنضع وسمًا على الشيفرة التي نطلقها باستخدام الأمر `git tag`، وستحتاج أيضًا إلى تقديم وسم الإصدار مثل وسيط لهذا الأمر.

لإنشاء وسم الإصدار، نبدأ بالبادئة `v` ونضيف SemVer بعدها مباشرةً. في حالتنا، سيكون وسم الإصدار النهائي هو `v0.1.0`.

شغل الأمر `git tag` لتمييز الوحدة التي أنشأناها بوسم الإصدار هذا:

```
$ git tag v0.1.0
```

سنحتاج -بعد إضافة وسم الإصدار محليًا- إلى دفع هذا الوسم إلى مستودع جيت هب باستخدام `git push` مع `origin`:

```
$ git push origin v0.1.0
```

بعد نجاح تنفيذ الأمر `git push`، سترى أن الوسم `v0.1.0` قد أُنشئ:

```
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:your_github_username/pubmodule.git
 \* [new tag]     v0.1.0 -> v0.1.0
```

يوضح الخرج أعلاه أن الوسم السابق قد أُضيف، وأن مستودع جيت هب الخاص بنا يحتوي على الوسم الجديد `v0.1.0`، وسيكون متاحًا لمستخدمي الوحدة الوصول إليه.

بعد نشر إصدار جديد من الوحدة واستخدام الأمر `git tag`، لن يكون المستخدم بحاجة إلى تنزيل إصدار جديد بناءً على أحدث إيداع تعمية hash من الفرع الافتراضي، وذلك عندما يرغب بالحصول على أحدث إصدار من الوحدة باستخدام الأمر `go get`.

بعد إطلاق إصدار الوحدة، ستبدأ أداة `go` في استخدام هذه الإصدارات لتحديد أفضل طريقة لتحديث الوحدة، ويتيح لنا ذلك إلى جانب الإصدار الدلالي، إمكانية تكرار الوحدات وتحسينها مع تزويد المستخدمين بتجربة متسقة ومستقرة.

31.9 الخاتمة

أنشأنا خلال هذا الفصل وحدةً برمجيةً عامة باستخدام لغة جو ونشرناها في مستودع جيت هب حتى يتمكن الآخرون من استخدامها، واستخدمنا أيضًا مفهوم الإصدار الدلالي لتحديد رقم إصدار مناسب للوحدة. وسّعنا أيضًا دوال هذه الوحدة، ونشرنا الإصدار الجديد اعتمادًا على مفهوم الإصدار الدلالي مع ضمان عدم تعطّل البرامج التي تعتمد عليها.

هل تطمح لبيع منتجاتك الرقمية عبر الإنترنت؟

استثمر مهاراتك التقنية وأطلق منتجًا رقميًا
يحقق لك دخلًا عبر بيعه على متجر بيكاليكا

أطلق منتجك الآن

32. استخدام وحدة خاصة Private Module

واحدة من ميزات لغة جو هو أنها تتضمن عددًا كبيرًا من الوحدات مفتوحة المصدر، وبالتالي يمكن الوصـول إليها وفحصها واستخدامها والتعلم منها بحريَّة. أحيانًا يكون من الضروري إنشاء وحدة جو خاصة لأسباب مختلفة، مثل الاحتفاظ بمنطق عمل داخلي خاص بالشركة.

سـتتعلم في هـذا الفصـل كيفيـة نشـر وحـدة module خاصـة في لغـة جـو، وكيفيـة إعـداد الاسـتيثاق authentication للوصول إليها، وستتعلم أيضًا كيفية استخدام هذا النوع من الوحدات ضمن مشروع.

32.1 المتطلبات الأساسية

- ستحتاج لامتلاك مساحة عمل خاصـة مُهيئـة وجـاهزة في لغـة جـو، فـإذا لم يكن لـديك واحـدة، فـارجع للتعليمات الـواردة في الفصـل الأول من الكتـاب، وثبت لغـة جـو Go وقم بإعـداد بيئـة تطـوير محليـة بحسب نظام تشغيلك.
- يُفضّل أن تكون على دراية بكيفية إنشاء الوحدات في لغو جو. يمكنك مراجعة فصل اسـتخدام الوحـدات Modules في لغة جو Go.
- لديك معرفة مسبقة بنظام التحكم بالإصدار جيت Git. يمكنك مراجعة مقالة ما هو جيت Git؟
- إنشاء مستودع فار غ باسم `mysecret` للوحدة التي ستنشرها.
- رمز وصول شخصي Personal access token على جيت هاب. ستحتاجه للسماح للغة جو بالوصول إلى مستودعك الخاص.

32.2 توزيع وحدة خاصة

على عكس العديد من لغات البرمجة، تـوزّع لغـة جـو الوحـدات البرمجيـة من المسـتودعات بـدلًا من خـادم الحزمة المركزي. تتمثل إحدى فوائد هذا النهج في أن نشر وحدة خاصة يشبه إلى حـد بعيـد نشـر وحـدة عامـة، إذ تُوزّع الوحدة الخاصة من خلال مستودع شيفرة مصدر خاص، بدلًا من طلب خادم حزمـة خـاص منفصـل تمامًـا. نظرًا لأن معظم خيارات استضافة التعليمات البرمجية المصدر تدعم هذا الأمـر بمـا يكفي، فلا داع لإعـداد خـادم خاص إضافي.

لاستخدام وحدة خاصة، يجب أن يكون لديك حق الوصول إلى وحدة خاصة. سنُنشئ في هـذا القسـم وحـدةً خاصة وننشرها، لتتمكن من استخدامها لاحقًا خلال هذا الفصل من الوصول إلى وحدة خاصة من شيفرة أُخرى.

سنستخدم بدايةً الأمر `git clone` لأخذ نسخة محلية من المستودع الفارغ الذي لا بُد أن نكون قد أنشأناه لأنه جزء من المتطلبات الأساسـية. يجب أن يكـون هـذا المسـتودع خاصًـا، وسـنفترض أن اسـمه `mysecret`. يمكن نسخ هذا المستودع إلى أي مكان نريده على جهاز الحاسوب، لكن يميـل العديـد من المطـورين إلى إنشـاء مجلد خاص يتضمن جميع مشاريعهم. سنستخدم هنا مجلدًا باسم `projects`. ابدأ بإنشاء المجلد وانتقل إليه:

```
$ mkdir projects
$ cd projects
```

من مجلد projects، شغل الأمر `git clone` لنسخ المستودع `mysecret` إلى جهاز الحاسب:

```

$ git clone git@github.com:your_github_username/mysecret.git
```

ستعطي جو تأكيدًا على عملية نسخ الوحدة، وقد تحذّرك بأنـك قـد استنسـخت مسـتودعًا فارغًـا، إذا حصـل ذلك فلا داعٍ للقلق بشأن ذلك الأمر:

```
Cloning into 'mysecret'...
warning: You appear to have cloned an empty repository.
```

انتقـل الآن باسـتخدام الأمـر `cd` إلى المجلـد الجديـد `mysecret` الـذي استنسـخته واسـتخدم الأمـر `go mod init` لإنشاء الوحدة الجديدة وتمرير موقع المستودع اسمًا لها:

```
$ cd mysecret
$ go mod init github.com/your_github_username/mysecret
```

يُعد التأكد من تطابق اسم الوحدة مع موقع المستودع أمرًا مهمًا؛ لأن هذه هي الطريقـة الـتي تعـثر بهـا أداة `go mod init` على مكان تنزيل الوحدة عند استخدامها في مشاريع أخرى.

سنستخدم الآن محرر نصوص مثل نانو nano، لإنشاء وفتح ملف يحمل نفس اسم المستودع mysecret.go.

```
$ nano mysecret.go
```

يمكن تسمية هذا الملف بأي اسم،ولكن يُفضّل استخدام نفس اسم الحزمة لتسهيل معرفة مكان البدء عند العمل مع حزمة غير مألوفة.

الآن سنضيف إلى الملف mysecret.go الدالة `SecretProcess()` التي تقوم بطباعة الرسالة التالية "Running the secret process!":

```
package mysecret
import "fmt"
func SecretProcess() {
  fmt.Println("Running the secret process!")
}
```

يمكنك الآن نشر الوحدة الخاصة ليستخدمها الآخرون. بما أن المستودع الذي أنشأته هو مستودع خاص، بالتالي لا يمكن لشخص غيرك الوصول إليه، ويمنحك ذلك إمكانية التحكم في حق وصول الآخرين إلى وحدتك، وبالتالي يمكنك السماح لأشخاص محددين فقط بالوصول إليها أو عدم السماح لأي شخص بالوصول.

بما أن الوحدات البرمجية الخاصة والعامة في لغة جو هي مستودعات مصدرية، فإن عملية نشر وحدة خاصة تتبع نفس نهج عملية نشر وحدة عامة.

يجب علينا الآن إدراج stage الملفات باستخدام `git add` وإيداعها بالمستودع باستخدام `git commit`، كي ننشر الوحدة:

```
$ git add .
$ git commit -m "Initial private module implementation"
```

سترى تأكيدًا من جيت بأن الإيداع الأولي قد نجح، إضافةً إلى ملخص للملفات المضمنة في الإيداع. سيكون الخرج كما يلي:

```
[main (root-commit) bda059d] Initial private module implementation
 2 files changed, 10 insertions(+)
 create mode 100644 go.mod
 create mode 100644 mysecret.go
```

نستخدم الآن الأمر `git push` لدفع الوحدة إلى مستودع جيت هب:

```
$ git push
```

سيدفع جو تغييراتك ويجعلها متاحة لأي شخص يملك حق الوصول إلى مستودعك الخاص، وسيكون الخرج كما يلي:

```
git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 404 bytes | 404.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:your_github_username/mysecret.git
 \* [new branch]   main -> main
```

يمكنك أيضًا إضافة إصدارات إلى الوحدة الخاصة بنفس الطريقة التي نفعلها مع الوحدات العامة.

أنشأت في هذا القسم وحدةً برمجيةً جديدةً تمتلك دالة تُدعى `SecretProcess` ونشرتها في مستودع `mysecret` الخاص بك على جيت هب، وبما أن هذا المستودع خاص، فهذا يعني أن الوحدة خاصة أيضًا، ومن أجل الوصول إلى هذه الوحدة من شيفرة أخرى أو برنامج آخر في جو، ستحتاج إلى إعداد مُحدد، حتى يعرف مُصرّف اللغة كيفية الوصول إليها.

32.3 ضبط جو لمنح إمكانية الوصول إلى الوحدات البرمجية الخاصة

كما ذكرنا سابقًا، تُوّزع الوحدات البرمجية المكتوبة بلغة جو من مستودعات الشيفرة المصدرية. يُضيف فريق تطوير لغة جو بعض الخدمات المركزية إلى هذه الوحدات للمساعدة في ضمان استمرار وجود هذه الوحدات في حالة حدوث شيء ما للمستودع الأصلي. تتضمن الإعدادات الافتراضية في لغة جو استخدام هذه الخدمات تلقائيًا، ولكن يمكن أن تحدث مشاكل عند محاولة تنزيل وحدة خاصة لأنها تتضمن إمكانية وصول مُقيّدة.

لإخبار مُصرّف اللغة بأن بعض مسارات الاستيراد خاصة ولا يجب محاولة استخدام خدمات جو المركزية معها، يمكنك استخدام متغير البيئة `GOPRIVATE`، وهو يمثّل قائمةً من بادئات مسار الاستيراد مفصولة بفواصل، إذ ستحاول أدوات جو الوصول إليها مباشرةً عند مواجهتها بدلًا من المرور عبر الخدمات المركزية. أحد الأمثلة على ذلك هو الوحدة الخاصة التي أنشأتها للتو.

إذًا، من أجل استخدام الوحدة الخاصة، ينبغي أن نخبر مُصرّف اللغة عن المسار الذي نعدّه خاصًا، وذلك عن طريق ضبط قيمة المتغير `GOPRIVATE`. هناك عدد قليل من الخيارات الممكن إجراؤها عند ضبط قيم متغير

GOPRIVATE، وأحد الخيارات هو ضبطه على github.com، لكن قد لا يفي ذلك بالغرض لأن هذا سيخبر جو بعدم استخدام الخدمات المركزية لأي وحدة مستضافة على github.com، بما في ذلك الوحدات التي لا تمتلكها.

الخيار الثاني هو ضبط GOPRIVATE على مسار المستخدم الخاص بك فقط، مثلًا github.com/your_github_username، فهذا يحل المشكلة السابقة التي تتجلى بعدّ كل الوحدات المستضافة خاصة، ولكن في وقت ما قد يكون لديك وحدات عامة أنشأتها وتريد تنزيلها من نفس الحساب. قد يكون هذا حلًا مناسبًا لحد ما، ولكن كما ذكرنا، قد يمنعك من استخدام وحدات عامة من نفس الحساب.

الخيار الأكثر دقة هو إعداد GOPRIVATE، بحيث يُطابق مسار الوحدة الخاصة بك تمامًا، مثلًا github.com/your_github_username/mysecret، إذ يحل ذلك المشاكل السابقة التي رأيناها في الخيار الأول والثاني، ولكن هنا يجب أن نضيف كل وحدة خاصة إلى هذا المتغير (قد تكون هذه مشكلةً للبعض) كما يلي:

```
GOPRIVATE=github.com/your_github_username/mysecret,github.com/
your_github_username/othersecret
```

يعود اختيار الخيار الأفضل لك، حسبما تراه مناسبًا أكثر.

نظرًا لأن لديك الآن وحدةً خاصة واحدة فقط، سنستخدم اسم المستودع الكامل قيمةً للمتغير. لإجراء ذلك، استخدم الأمر export كما يلي:

```
$ export GOPRIVATE=github.com/your_github_username/mysecret
```

إذا كنت ترغب في التحقق من نجاح عملية ضبط المتغير، فيمكنك استعراض قيمته باستخدام الأمر env مع grep:

```
$ env | grep GOPRIVATE
```

سيكون الخرج كما يلي:

```
GOPRIVATE=github.com/your_github_username/mysecret
```

الآن، أصبح مُصرّف اللغة يعرف أن وحدتك خاصة، لكن هذا لا يزال غير كافٍ لاستخدام الوحدة في شيفرة أو برنامج آخر. جرّب تشغيل هذه الوحدة في وحدة أخرى:

```
$ go get github.com/your_github_username/mysecret
```

وسترى الخطأ التالي:

```
go get: module github.com/your_github_username/mysecret: git ls-remote
-q origin in
/Users/your_github_username/go/pkg/mod/cache/vcs/2f8c...b9ea: exit
status 128:

  fatal: could not read Username for 'https://github.com': terminal
prompts disabled

Confirm the import path was entered correctly.

If this is a private repository, see
https://golang.org/doc/faq#git_https for additional information.
```

تشير رسالة الخطأ هذه إلى أن مُصرّف اللغة حاول تنزيل الوحدة الخاصة بك، لكنه واجه شيئًا لا يمكنه الوصول إليه. بما أنك تستخدم جيت git لتنزيل الوحدة، فغالبًا سيُطلب منك إدخال بيانات الاعتماد credentials الخاصة بك، لكن هنا يتصل مُصرّف اللغة بجيت نيابةً عنك، وبالتالي لا يمكنه مطالبتك بإدخال بيانات الاعتماد، وبالتالي يظهر الخطأ. إذًا، أنت بحاجة إلى توفير طريقة لجيت لاسترداد بيانات الاعتماد الخاصة بك.

32.4 توفير بيانات الاعتماد اللازمة للاتصال بالوحدة الخاصة عند استخدام بروتوكول HTTPS

هناك طريقة واحدة لإخبار جو بكيفية تسجيل الدخول نيابةً عنك، وهي ملف netrc. الموجود في المجلد الرئيسي Home الخاص بالمستخدم؛ إذ يحتوي هذا الملف على أسماء مُضيفات مختلفة Hosts مع بيانات اعتماد تسجيل الدخول لهم، ويستخدم على نطاق واسع في عدة أدوات برمجية بما في ذلك لغة جو.

سيحاول الأمر go get استخدام HTTPS أولًا عندما يحاول تنزيل الوحدة، لكن كما ذكرنا؛ لا يمكنه مطالبتك باسم المستخدم وكلمة المرور. لمنح جيت بيانات الاعتماد الخاصة بك، يجب أن يكون لديك ملف netrc. يتضمن github.com في مجلدك الرئيسي.

لإنشاء ملف netrc. على نظام لينكس Linux أو ماك MacOS أو على نظام ويندوز الفرعي لنظام لينكس Windows Subsystem for Linux -أو اختصارًا WSL، افتح ملف netrc. في المجلد الرئيسي (~/) حتى تتمكن من تحريره:

```
$ nano ~/.netrc
```

أنشئ الآن مدخلًا جديدًا إلى هذا الملف. ينبغي أن تكون قيمة الحقل machine هي اسم المضيف الذي تضبط بيانات الاعتماد له، وهو github.com في هذه الحالة، أما قيمة login فهي اسم المستخدم الخاص بك على جيت هب. أخيرًا، قيمة password يجب أن تكون رمز الوصول الشخصي الخاص بك، والذي من المفترض أن تكون قد أنشأته على جيت هب.

```
machine github.com
login your_github_username
password your_github_access_token
```

يمكنك أيضًا وضع جميع المدخلات ضمن سطر واحد في الملف:

```
machine github.com login your_github_username password
your_github_access_token
```

> إذا كنت تستخدم Bitbucket لاستضافة شيفرة المصدر الخاصة بك، فقد تحتاج أيضًا إلى إضافة مُدخل ثاني من أجل api.bitbucket.org إضافةً إلى bitbucket.org. كانت Bitbucket توفّر سابقًا استضافةً لأنواع متعددة من أنظمة التحكم في الإصدار، لذلك سيستخدم مُصرّف جو واجهة برمجة التطبيقات للتحقق من نوع المستودع قبل محاولة تنزيله.

هذا الأمر كان في الماضي ولم يعد الأمر كذلك، إلا أن واجهة برمجة التطبيقات التي تُمكّنك من فحص نوع المستودع ما زالت موجودة. بكافة الأحوال، إذا واجهت هذه المشكلة، فقد تبدو رسالة الخطأ على النحو التالي:

```
go get bitbucket.org/your_github_username/mysecret: reading
https://api.bitbucket.org/2.0/repositories/your_bitbucket_username/
protocol?fields=scm: 403 Forbidden

  server response: Access denied. You must have write or admin access.
```

إذا رأيت رسالة الخطأ "403 Forbidden" أثناء محاولة تنزيل وحدة خاصة، تحقق من اسم مستخدم جو الذي تحاول الاتصال به؛ فمن الممكن أن تشير إلى اسم مستخدم مختلف مثل `api.bitbucket.org`، والذي ينبغي إضافته إلى الملف netrc..

بذلك تكون قد أعددت بيئتك لاستخدام مصادقة HTTPS لتنزيل الوحدة الخاصة بك. ذكرنا سابقًا أن الأمر `go get` يستخدم افتراضيًا HTTPS عندما يحاول تنزيل الوحدة، لكن من الممكن أيضًا جعله يستخدم بروتوكول النقل الآمن Secure Shell -أو اختصارًا SSH- بدلًا من ذلك. يمكن أن يكون استخدام SSH بدلًا من HTTPS مفيدًا حتى تتمكن من استخدام نفس مفتاح SSH الذي استخدمته لدفع الوحدة الخاصة بك، كما يسمح لك باستخدام مفاتيح النشر deploy keys عند إعداد بيئة CI/CD إذا كنت تفضل عدم إنشاء رمز وصول شخصي.

32.5 توفير بيانات الاعتماد اللازمة للاتصال بالوحدة الخاصة عند استخدام بروتوكول SSH

يوفر جيت خيار إعداد يسمى `insteadOf` لاستخدام مفتاح SSH الخاص بك بمثابة طريقة للمصادقة مع الوحدة الخاصة بك بدلًا من HTTPS. بمعنى آخر يتيح لك هذا الخيار أن تقول "بدلًا من" استخدام

`https://github.com/` مثـــل عنـــوان URL لإرســـال طلبـــات إلى جيت، فـــأنت تُريـــد استخدام `ssh://git@github.com/`.

يتواجد هذا الإعداد في أنظمة لينكس وماك ونظام ويندوز الفرعي WSL في ملف gitconfig.. قد تكون على دراية بهذا الملف فعلًا، إذ يتضمن أيضًا إعدادات عنوان البريد الإلكتروني والاسم الخاصين بك. افتح هذا الملف `~/.gitconfig` الآن من مجلدك الرئيسي `Home` باستخدام محرر النصوص الذي تفضله وليكن نانو:

```
$ nano ~/.gitconfig
```

عدّل هذا الملف ليحتوي على قسم `url` من أجل `ssh://git@github.com/` كما يلي:

```
[user]
  email = your_github_username@example.com
  name = Sammy the Shark
[url "ssh://git@github.com/"]
  insteadOf = https://github.com/
```

ترتيب قسم `user` بالنسبة للقسم `url` غير مهم، ولا داع للقلق إن لم يكن هناك أي شيء آخر غير قسم `url` الذي أضفته منذ قليل. أيضًا تريتب حقلي `email` و `name` داخل قسم `user` غير مهم.

سيخبر القسم الجديد الذي أضفته جيت أن أي عنوان URL تكون بادئته `https://github.com/` يجب أن تُستبدل بادئته إلى `ssh://git@github.com/` (الاختلاف بالبادئة). وهذا سيؤثر طبعًا على أوامر `go get`، لأن لغة جو تستخدم HTTPS افتراضيًا. لو أخذنا مسار الوحدة الخاصة بك مثالًا، فإن جو سوف يُحوّل مسار الاستيراد `github.com/your_github_username/mysecret` إلى عنوان URL يكون كما يلي: `https://github.com/your_github_username/mysecret`. عندما يصادف جو عنوان URL هذا، سيرى عنوان URL يطابق البادئة `https://github.com/` المشار إليها عن طريق `insteadOf` وسوف يعيد عنوان URL إلى `ssh://git@github.com/your_github_username/mysecret`

يمكن أيضًا استخدام هذا النمط لنطاقات أخرى وليس فقط لجيت هب، طالما أن `ssh://git@` يعمل مع ذلك المضيف أيضًا.

ضبطنا خلال هذا القسم جيت لاستخدام SSH، وذلك من أجل تنزيل وحدات جو، وذلك من خلال تحديث ملف gitconfig.، إذ أضفنا القسم `url`. الآن بعد أن أكملنا الإعدادات اللازمة للمصادقة، أصبح بالإمكان الوصول إلى الوحدة واستخدامها في برامج جو.

32.6 استخدام وحدة خاصة

تعلمت خلال الأقسام السابقة كيفية إعداد جو ليتمكن من الوصول إلى الوحدة الخاصة بك عبر HTTPS أو SSH أو ربما كليهما. إذًا، أصبح بإمكانك الآن استخدامها مثل أي وحدة عامة أخرى. ستنشئ الآن وحدةً جديدةً تستخدم وحدتك الخاصة.

أنشئ الآن مجلدًا باسم `myproject` باستخدام الأمر `mkdir` وضعه في مجلد المشاريع `projects`:

```
$ mkdir myprojec
```

انتقل الآن إلى المجلد الذي أنشأته باستخدام الأمر `cd` وهيّئ الوحدة الجديدة باستخدام الأمر `go mod init` اعتمادًا على عنوان URL للمستودع الذي ستضع فيه محتويات المشروع، مثل `github.com/your_github_username/myproject`. إذا كنت لا تخطط لدفع المشروع إلى مستودع آخر، فيمكن أن يكون اسم الوحدة `myproject` أو أي اسم آخر، ولكن من الممارسات الجيدة استخدام عناوين URL الكاملة، لأن معظم الوحدات التي تجري مشاركتها ستحتاج إليها.

```
$ cd myproject
$ go mod init github.com/your_github_username/myproject
```

سيكون الخرج على النحو التالي:

```
go: creating new go.mod: module
github.com/your_github_username/myproject
```

أنشئ الآن ملف main.go لتضع فيه أول شيفرة باستخدام محرر النصوص الذي تفضله وليكن نانو:

```
$ nano main.go
```

ضع الشيفرة التالية داخل هذا الملف، والتي تتضمن الدالة `main()` التي سنستدعي الوحدة بداخلها:

```
package main
import "fmt"
func main() {
  fmt.Println("My new project!")
}
```

شغل ملف main.go باستخدام الأمر `go run` لرؤية المشروع النهائي الذي يستخدم الوحدة الخاصة بك:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
My new project!
```

أضف الآن وحدتك الخاصة مثل اعتمادية لمشروعك الجديد باستخدام الأمر `go get` تمامًا كما تفعل مع الوحدة العامة:

```
$ go get github.com/your_github_username/mysecret
```

تنزّل عندها أداة `go` شيفرة الوحدة الخاصة وتضيفها مثل اعتمادية باستخدام سلسلة نصية للإصدار تطابق إيداع التعميلة hash الأخير ووقت ذلك الإيداع:

```
go: downloading github.com/your_github_username/mysecret v0.0.0-
20210920195630-bda059d63fa2
go get: added github.com/your_github_username/mysecret v0.0.0-
20210920195630-bda059d63fa2
```

افتح ملف main.go مرةً أخرى وحدّثه لإضافة استدعاء لدالة الوحدة الخاصة `SecretProcess` ضمن دالة `main()` الرئيسية. ستحتاج أيضًا إلى تحديث عبارة `import` لإضافة وحدتك الخاصة `github.com/your_github_username/mysecret`:

```
package main
import (
  "fmt"
  "github.com/your_github_username/mysecret"
)
func main() {
  fmt.Println("My new project!")
  mysecret.SecretProcess()
}
```

شغل ملف main.go باستخدام الأمر `go run` لرؤية المشروع النهائي الذي يستخدم الوحدة الخاصة بك:

```
$ go run main.go
```

ستلاحظ أن الخرج يتضمن الجملة `My new project!` من الشيفرة الأصلية، كما يتضمن الجملة `Running the secret process!` من الوحدة `mysecret` التي استوردناها.

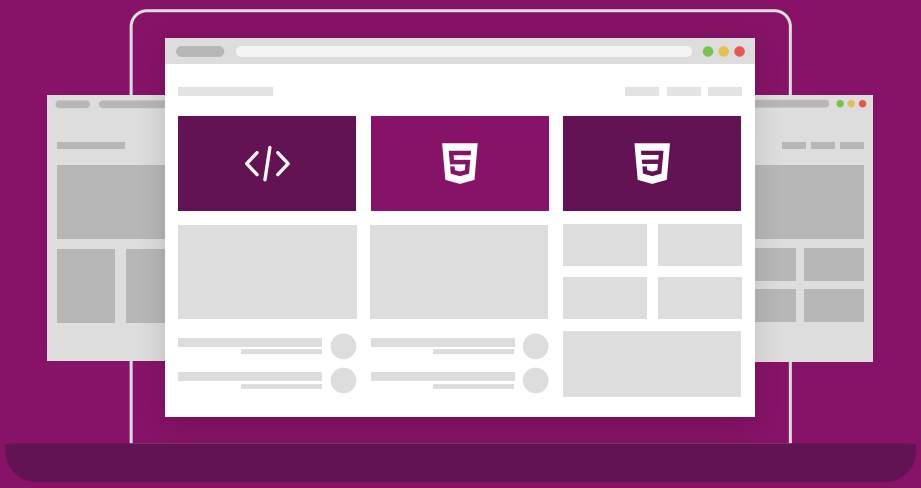
```
My new project!
Running the secret process!
```

استخدمنا في هذا القسم الأمر `go init` لإنشاء وحدة جديدة للوصول إلى الوحدة الخاصة التي نشرناها سابقًا. بعد إنشاء الوحدة، ستتمكن من استخدام الأمر `go get` لتنزيل الوحدة الخاصة بك؛ تمامًا كما لو كنت تتعامل مع وحدة عامة. استخدمنا أيضًا الأمر `go run` لتصريف وتشغيل البرنامج الذي يستخدم الوحدة الخاصة.

32.7 الخاتمة

نشرنا خلال هذا الفصل وحدة خاصة باستخدام لغة جو، وتعلمنا كيفية إعداد متطلبات المصادقة باستخدام بروتوكول HTTPS ومفتاح SSH من أجل الوصول إلى الوحدة الخاصة من شيفرة أخرى. استخدمنا أيضًا الوحدة الخاصة ضمن مشروع.

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

33. تنفيذ عدة دوال من خلال ميزة التسايُر Concurrency

واحدة من أهم الميزات التي تدعمها لغة جو هي القدرة على إجراء أكثر من عملية في وقت واحد، وهذا ما يُسمى **التسايُر Concurrency**، وقد أصبحت فكرة تشغيل التعليمات البرمجية بطريقة متسايرة جزءًا مهمًا في تطبيقات الحاسب نظرًا لما تتيحه من استثمار أكبر للموارد الحاسوبية المتاحة والسرعة في إنهاء تنفيذ البرامج، فبدلًا من تنفيذ كتلة واحدة من التعليمات البرمجية في وقت واحد، يمكن تنفيذ عدة كتل من التعليمات البرمجية.

مفهوم التساير هو فكرة يمكن دعم البرامج بها من خلال تصميم البرنامج بطريقة تسمح بتنفيذ أجزاء منه باستقلالية عن الأجزاء الأخرى.

تمتلك لغة جو ميزتين، هما: خيوط معالجة جو Goroutines والقنوات Channels تُسهّلان إجراء عمليات التساير؛ إذ تُسهّل الأولى عملية إعداد الشيفرة المتسايرة للبرنامج، وتجعل الثانية عملية التواصل بين أجزاء البرنامج المتساير آمنة.

سنتعرّف في هذا الفصل على كل من خيوط معالجة جو والقنوات، إذ سننشئ برنامجًا يستخدم خيوط معالجة جو لتشغيل عدة دوال في وقت واحد، ثم سنضيف قنوات لتحقيق اتصال آمن بين أجزاء البرنامج المتساير. أخيرًا، سنضيف عدة خيوط معالجة (كل منها يُسمّى عامل Worker، إشارةً إلى أدائه مهمة ما)، لمحاكاة حالات أكثر تعقيدًا.

33.1 الفرق بين التزامن Synchronous وعدم التزامن Asynchronous والتساير Concurrency والتوازي Parallelism

التزامن و**عدم التزامن** هما نموذجان مختلفان للبرمجة، يشيران إلى أنماط البرمجة. تُكتب التعليمات البرمجية في النموذج الأول مثل خطوات، إذ تُنفّذ التعليمات البرمجية من الأعلى إلى الأسفل، خطوةً بخطوة، وتنتقل إلى الخطوة الثانية فقط عندما تنتهي من الخطوة الأولى، ويمكن هنا التنبؤ بالخرج:

```
func step1() { print("1") }
func step2() { print("2") }
func main() {
  step1()
  step2()
}
// result -> 12
```

تُكتب التعليمات البرمجية في النموذج الثاني مثل مهمات، ويجري تنفيذها بعد ذلك على التساير. **التنفيذ المتساير** يعني أنه من المحتمل أن تُنفّذ جميع المهام في نفس الوقت، وهنا لا يمكن التنبؤ بالخرج:

```
func task1() { print("1") }
func task2() { print("2") }
func main() {
  task1()
  task2()
}
// result -> 12 or 21
```

في نموذج البرمجة غير المتزامن، يكون التعامل مع المهام على أنها خطوة واحدة تُشغّل مهام متعددة، ولا يؤخذ بالحسبان كيفية ترتيب هذه المهام. يمكن تشغيلها في وقت واحد أو في بعض الحالات، ستكون هناك بعض المهام التي تُنفّذ أولًا ثم تتوقف مؤقتًا وتأتي مهام أخرى لتحل محلها بالتناوب وهكذا. يُطلق على هذا السلوك اسم **متساير**.

لفهم الأمر أكثر تخيل أنّه طُلب منك تناول كعكة ضخمة وغناء أغنية، وستفوز إذا كنت أسرع من يغني الأغنية وينهي الكعكة. القاعدة هي أن تغني وتأكل في نفس الوقت، ويمكنك أن تأكل كامل الكعكة ثم تغني كامل الأغنية، أو أن تأكل نصف كعكة ثم تغني نصف أغنية ثم تفعل ذلك مرةً أخرى، إلخ. ينطبق الأمر نفسه على علوم الحاسب؛ فهناك مهمتان تُنفذان بصورةٍ متسايرة، ولكن تُنفذان على وحدة معالجة مركزية أحادية النواة،

لذلك ستقرر وحدة المعالجة المركزية تشغيل مهمة أولًا ثم المهمة الأخرى، أو تشغيل نصف مهمة ونصف مهمة أخرى، إلخ. يجعلنا هذا التقسيم نشعر بأن جميع المهام تُنفّذ بنفس الوقت.

حسنًا تبقى لنا معرفة **التنفيذ المتوازي**. بالعودة إلى مثال الكعكة والأغنية تخيل أن يُسمح لك بالاستعانة بصديق، وبالتالي يمكن أن يغني بينما أنت تأكل. هذا يقابل أن يكون لدينا وحدة معالجة مركزية بنواتين، إذ يمكن تنفيذ المهمة -التي يمكن تقسيمها لمهمتين فرعيتين- على نواتين مختلفتين، وهذا ما يسمى **بالتوازي**، وهو نوع معين من التساير، إذ تُنفّذ المهام فعلًا في وقت واحد. لا يمكن تحقيق التوازي إلا في بيئات متعددة النواة.

33.2 المتطلبات

- ستحتاج لإصدار مُثبّت من جو 1.13 أو أعلى، ويمكنك الاستعانة بالتعليمات الواردة في الفصل الأول من الكتاب، لثبيت لغة جو Go وإعداد بيئة تطوير محلية بحسب نظام تشغيلك.
- على درايةٍ بكيفية عمل واستخدام الدوال في لغة جو. يمكنك الاطلاع على فصل كيفية تعريف واستدعاء الدوال في لغة جو Go.

33.3 تشغيل عدة دوال بذات الوقت باستخدام خيوط المعالجة Goroutines

تُصمّم المعالجات الحديثة أو وحدة المعالجة المركزية CPU لأجهزة الحواسيب بحيث يمكنها تنفيذ أكبر عدد من المجاري التدفقية Streams من الشيفرة (أي كتل من التعليمات البرمجية) في نفس الوقت. لتحقيق هذه الإمكانية تتضمن المعالجات نواة أو عدة أنوية (يمكنك التفكير بكل نواة على أنها معالج أصغر) كل منها قادر على تنفيذ جزء من تعليمات البرنامج في نفس الوقت. بالتالي، يمكننا أن نستنتج أنّه يمكن الحصول على أداء أسرع بازدياد عدد الأنوية التي تُشغّل البرنامج. لكن مهلًا، لا يكفي وجود عدة أنوية لتحقيق الأمر؛ إذ يجب أن يدعم البرنامج إمكانية التنفيذ على نوى متعددة وإلا لن يُنفّذ البرنامج إلا على نواة واحدة في وقت واحد ولن نستفيد من خاصية النوى المتعددة والتنفيذ المتساير.

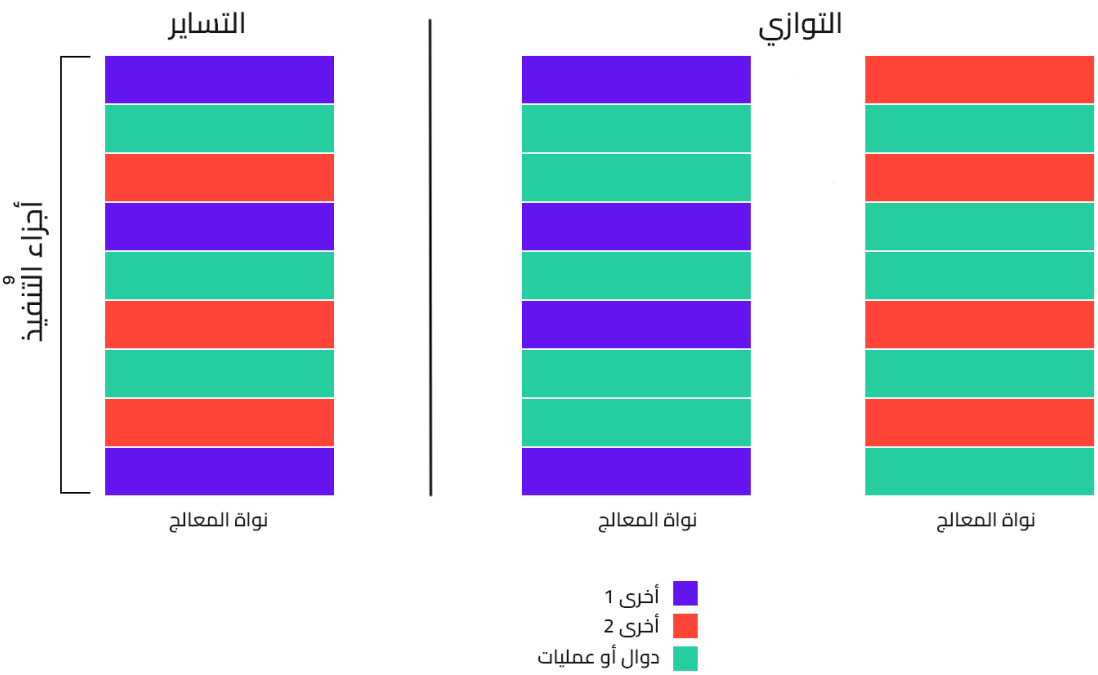
للاستفادة من وجود النوى المتعددة، يقع على عاتق المبرمج تصميم شيفرة البرنامج، بحيث يمكن تقسيم هذه الشيفرة إلى كتل تُنفّذ باستقلالية عن بعضها. ليس تقسيم شيفرة البرنامج إلى عدة أجزاء أو كتل برمجية أمرًا سهلًا، فهو يمثّل تحدٍ للمبرمج. لحسن الحظ أن لغة جو تجعل الأمر أسهل من خلال الأداة `goroutine` أو "تنظيم جو".

تنظيم جو هو نوع خاص من الدوال يمكنه العمل في نفس الوقت الذي تعمل به خيوط المعالجة الأخرى. نقول عن برنامج أنّه مُصمّم ليُنفّذ على التساير عندما يتضمن كتل من التعليمات البرمجية التي يمكن تنفيذها باستقلالية في وقت واحد. بالحالة العادية عندما تُستدعى دالة ما، ينتظر البرنامج انتهاء تنفيذ كامل الدالة قبل أن يكمل تنفيذ الشيفرة. يُعرف هذا النوع من التنفيذ باسم التشغيل في "المقدمة Foreground" لأنه يمنع

البرنامج من تنفيذ أي شيء آخر قبل أن ينتهي. من خلال "تنظيم جو"، ستُستدعى الدالة وتنفّذ في "الخلفية Background" بينما يُكمل البرنامج تنفيذ باقي الشيفرة. نقول عن شيفرةً -أو جزء من شيفرة- أنها تُنفّذ في الخلفية عندما لا يمنع تنفيذها باقي أجزاء الشيفرة من التنفيذ، أي عندما لا تكون باقي أجزاء الشيفرة مُضطرة لانتظارها حتى تنتهي.

تأتي قوة خيوط معالجة جو -وهي عبارة عن خيط معالجة lightweight thread يديره مشغل جو الآني Go runtime)- من فكرة أن كل تنظيم يمكنه أن يشغل نواة معالج واحدة في نفس الوقت. إذا كان لديك معالج بأربع نوى ويتضمن برنامجك 4 خيوط معالجة، فإن كل تنظيم يمكنه أن يُنفّذ على نواة في نفس الوقت. عندما تُنفّذ عدة أجزاء من الشيفرة البرمجيّة في نفس الوقت على نوى مختلفة (كما في الحالة السابقة) -نقول عن عملية التنفيذ أنها تفرعيّة (أو متوازية). يمكنك الاطلاع على مقال تنفيذ المهام بالتوازي في dot NET على أكاديمية حسوب لمزيدٍ من المعلومات حول تنفيذ المهام على التوازي.

لتوضيح الفرق بين التساير concurrency والتوازي parallelism، ألقِ نظرةً على المخطط التالي. عندما يُنفّذ المعالج دالةً ما، فهو عادةً لا يُنفّذها من البداية للنهاية دفعةً واحدة، إذ يعمل نظام التشغيل أحيانًا على التبديل بين الدوال أو خيوط المعالجة أو البرامج الأخرى التي تُنفّذ على نواة المعالج عندما تنتظر الدالة حدوث شيءٍ ما (مثلًا قد يتطلب تنفيذ الدالة استقبال مُدخلات من المستخدم أو قراءة ملف). يعرض لنا المخطط كيف يمكن للبرنامج المصمّم للعمل على التساير التنفيّذ على نواة واحدة أو عدة نوى، كما يوضح أيضًا أنه من الملائم أكثر تنفيّذ خيوط معالجة جو على التوازي من التشغيل على نواة واحدة.



يوّضح المخطط على اليسار والمُسمّى "تساير Concurrency" كيف يمكن تنفيذ برنامج متساير التصميم على نواة معالج وحيدة من خلال تشغيل أجزاء من goroutine1 ثم دالة أو تنظيم أو برنامج ثم goroutine2 ثم goroutine1 مرةً أخرى وهكذا. هنا يشعر المستخدم بأن البرنامج ينفِّذ كل الدوال أو خيوط المعالجة في نفس الوقت، على الرغم من أنها تُنفّذ على التسلسل.

يوضّح العمود الأيمن من المخطط والمسمّى "توازي Parallelism" كيف أن نفس البرنامج يمكن أن يُنفّذ على التوازي على معالج يملك نواتين، إذ يُظهر المخطط أن النواة الأولى تنفِّذ goroutine1 وهناك دوال أو خيوط معالجة أو برامج أخرى تُنفّذ معها أيضًا، نفس الأمر بالنسبة للنواة الثانية التي تنفِّذ goroutine1. نلاحظ أحيانًا أن goroutine1 و goroutine2 تُنفّذان في نفس الوقت لكن على نوى مختلفة.

يظهر هذا المخطط أيضًا سمات أخرى من سمات جو القوية، وهي قابلية التوسع Scalability، إذ يكون البرنامج **قابلًا للتوسع** عندما يمكن تشغيله على أي شيء بدءًا من جهاز حاسوب صغير به عدد قليل من الأنوية وحتى خادم كبير به عشرات الأنوية والاستفادة من هذه الموارد الإضافية، أي كلما أعطيته موارد أكبر، يمكنه الاستفادة منها. يوضّح المخطط أنه من خلال استخدام خيوط معالجة جو، يمكن لبرنامج متساير أن يُنفّذ على نواة وحيدة، لكن مع زيادة عدد الأنوية سيكون البرنامج قابلًا للتنفيذ المتوازي، وبالتالي يمكن تنفيّذ أكثر من تنظيم في نفس الوقت وتسريع الأداء.

للبدء بإنشاء برنامج متساير، علينا أولًا إنشاء مجلد multifunc في المكان الذي تختاره. قد يكون لديك مجلد مشاريع خاص بك، لكن هنا سنعتمد مجلدًا اسمه projects. يمكنك إنشاء المجلد إما من خلال بيئة تطوير متكاملة IDE أو سطر الأوامر. إذا كنت تستخدم سطر الأوامر، فابدأ بإنشاء مجلد projects وانتقل إليه:

```
$ mkdir projects
$ cd projects
```

من المجلد projects استخدم الأمر mkdir لإنشاء مجلد المشروع multifunc وانتقل إليه:

```
$ mkdir multifunc
$ cd multifunc
```

افتح الآن ملف main.go باستخدام محرر نانو nano أو أي محرر آخر تريده:

```
$ nano main.go
```

ضع بداخله الشيفرة التالية:

```
package main
import (
  "fmt"
```

```
)
func generateNumbers(total int) {
  for idx := 1; idx <= total; idx++ {
        fmt.Printf("Generating number %d\n", idx)
  }
}
func printNumbers() {
  for idx := 1; idx <= 3; idx++ {
        fmt.Printf("Printing number %d\n", idx)
  }
}
func main() {
  printNumbers()
  generateNumbers(3)
}
```

يعرّف هذا البرنامج الأولي دالـتين `generateNumbers` و `printNumbers`، إضـافةً إلى الدالـة الرئيسـية `main()` التي تُستدعى هذه الدوال ضمنها. تأخذ الدالة الأولى معامًلا يُدعى `total` يُمثّل عدد الأعداد المطلوب توليدها، وفي حالتنا مررنا القيمة 3 لهذه الدالة وبالتالي سـنرى الأعـداد من 1 إلى 3 على شاشـة الخـرج. لا تأخـذ الدالة الثانية أيّة معاملات، فهي تطبع دومًا الأرقام من 0 إلى 3.

بعد حفظ ملف main.go، يمكننا تشغيله باستخدام الأمر التالي:

```
$ go run main.go
```

سيكون الخرج كما يلي:

```
Printing number 1
Printing number 2
Printing number 3
Generating number 1
Generating number 2
Generating number 3
```

نلاحظ أن الدالة `printNumbers()` نُفّـذت أولًا، ثم تلتهـا الدالـة `generateNumbers()`، وهـذا منطقي لأنها استُدعيت أولًا. تخيل الآن أن كل من هاتين الدالتين تحتاج 3 ثوان حتى تُنفّذ. بالتـالي عنـد تنفيـذ البرنـامج السابق بطريقة متزامنة synchronously (خطوةً خطوة)، سيتطلب الأمر 6 ثـوانٍ للتنفيـذ (3 ثـوان لكـل دالـة). الآن، لو تأملنا قليلًا سنجد أن هاتين الـدالتين مسـتقلتان عن بعضـهما بعضًـا، أي لا تعتمـد أحـدهما على نتيجـة

تنفيذ الأخرى، وبالتالي يمكننا الاستفادة من هذا الأمر والحصول على أداء أسرع للبرنامج من خلال تنفيذ الدوال بطريقة متسايرة باستخدام خيوط معالجة جو. نظريًّا: سنتمكن من تنفيذ البرنامج في نصف المدة (3 ثوان)، وذلك لأن كل من الدالتين تحتاج 3 ثوان لتُنفّذ، وبما أنهما سيُنفذان في نفس الوقت، بالتالي يُفترض أن ينتهي تنفيذ البرنامج خلال 3 ثوان.

يختلف حقيقةً الجانب النظري هنا عن التجربة، فليس ضروريًا أن يكتمل التنفيذ خلال 3 ثوان، لأن هناك العديد من العوامل الأخرى الخارجية، مثل البرامج الأخرى التي تؤثر على زمن التنفيذ.

إن تنفيذ دالة على التساير من خلال تنظيم جو مُشابه لتنفيّذ دالة بطريقة متزامنة. لتنفيذ دالة من خلال تنظيم جو (أي بطريقة متسايرة) يجب وضع الكلمة المفتاحية `go` قبل استدعاء الدالة.

لجعل البرنامج يُنفّذ جميع خيوط المعالجة على التساير، يتعين علينا إضافة تعديل بسيط للبرنامج، بحيث نجعله ينتظر انتهاء تنفيذ كل خيوط معالجة جو. هذا ضروري لأنه إذا انتهى تنفيذ دالة `main` ولم تنتظر انتهاء خيوط معالجة جو، فقد لا يكتمل تنفيذ خيوط معالجة جو (تحدث مقاطعة لتنفيذها إن لم تكن قد انتهت).

لتحقيق عملية الانتظار هذه نستخدم `WaitGroup` من حزمة `sync` التابعة لجو، والتي تتضمّن الأدوات الأولية للتزامن synchronization primitives، مثل `WaitGroup` المُصممة لتحقيق التزامن بين عدة أجزاء من البرنامج. ستكون مهمة التزامن في مثالنا هي تعقب اكتمال تنفيذ الدوال السابقة وانتظارها حتى تنتهي لكي يُسمح بإنهاء البرنامج.

تعمل الأداة الأولية `WaitGroup` عن طريق حساب عدد الأشياء التي تحتاج إلى انتظارها باستخدام دوال `Add` و `Done` و `Wait`. تزيد الدالة `Add` العداد من خلال الرقم المُمرر إلى الدالة، بينما تنقص الدالة `Done` العداد بمقدار واحد. تنتظر الدالة `Wait` حتى تصبح قيمة العداد 0، وهذا يعني أن الدالة `Done` استُدعيت بما يكفي من المرات لتعويض استدعاءات `Add`. بمجرد وصول العداد إلى الصفر، ستعود دالة الانتظار وسيستمر البرنامج في العمل.

لنعدّل ملف main.go لتنفيذ الدوال من خلال خيوط معالجة جو باستخدام الكلمة المفتاحية `go`، ولنضِف `sync.WaitGroup` إلى البرنامج:

```
package main
import (
  "fmt"
  "sync"
)
func generateNumbers(total int, wg *sync.WaitGroup) {
  defer wg.Done()
  for idx := 1; idx <= total; idx++ {
```

```
            fmt.Printf("Generating number %d\n", idx)
    }
}
func printNumbers(wg *sync.WaitGroup) {
    defer wg.Done()
    for idx := 1; idx <= 3; idx++ {
            fmt.Printf("Printing number %d\n", idx)
    }
}
func main() {
    var wg sync.WaitGroup
    wg.Add(2)
    go printNumbers(&wg)
    go generateNumbers(3, &wg)
    fmt.Println("Waiting for goroutines to finish...")
    wg.Wait()
    fmt.Println("Done!")
}
```

سنحتاج بعد التصريح عن `WaitGroup` إلى معرفة عدد الأشياء التي يجب انتظارها. سيدرّك `wg` عند وضع التعليمة `wg.Add(2)` داخل الدالة `main` قبل بدء تنفيذ خيوط معالجة جو، أن عليه انتظار استدعائين `Done` حتى يُنهي عملية الانتظار. إذا لم نفعل ذلك قبل بدء تنفيذ خيوط معالجة جو، فمن المحتمل أن تحدث حالات تعطّل في البرنامج أو قد تحدث حالة انهيار Panic في الشيفرة، لأن `wg` لا يعرف أنه يجب أن ينتظر أية استدعاءات `Done`.

ستستخدم كل دالة `defer` بعد ذلك، لاستدعاء `Done` بهدف تخفيض العداد بواحد بعد انتهاء تنفيذ الدالة. تُحدَّث الدالة `main` أيضًا لتضمين استدعاء `Wait` من `WaitGroup`، لذا ستنتظر الدالة `main` حتى تُستدعى الدالة `Done` مرتين قبل إنهاء البرنامج.

بعد حفظ ملف main.go، يمكننا تشغيله باستخدام الأمر التالي:

```
$ go run main.go
```

ويكون الخرج على النحو التالي:

```
Printing number 1
Waiting for goroutines to finish...
Generating number 1
```

```
Generating number 2
Generating number 3
Printing number 2
Printing number 3
Done!
```

قد يختلف الخرج عما تراه أعلاه (تذكر أن التنفيذ المتساير لا يمكن التنبؤ بسلوكه)، وقد يختلف في كل مرة تُنفّذ فيها الشيفرة. سيعتمد الخرج الناتج عن تنفيذ الدالتين السابقتين على مقدار الوقت الذي يمنحه نظام التشغيل ومُصرّف لغة جو لكل دالة وهذا أمر يصعب معرفته؛ فمثلًا قد تُمنح كل دالة وقتًا كافيًا لتُنفّذ كل تعليماتها دفعةً واحدة قبل أن يقاطع نظام التشغيل تنفيذها، وبالتالي سيكون الخرج كما لو أن التنفيذ كان تسلسيًا، وفي بعض الأحيان لن تحصل الدالة على ما يكفي من الوقت دفعةً واحدة، أي تطبع سطر ثم يتوقف تنفيذها ثم تطبع الدالة الثانية سطر ثم يتوقف تنفيذها ثم نعود للدالة الأولى فتطبع باقي الأسطر وهكذا، وسنرى عندها خرجًا مشابهًا للخرج أعلاه.

على سبيل التجربة، يمكننا حذف تعليمة استدعاء `wg.Wait()` في الدالة الرئيسية `main` ثم تنفيذ الشيفرة عدة مرات باستخدام `go run`. اعتمادًا على جهاز الحاسب المُستخدم، قد ترى بعض النتائج من دالتي `generateNumbers` و `printNumbers`، ولكن من المحتمل أيضًا ألا ترى أي خرج منهما إطلاقًا، وذلك لأنه عند حذفنا لاستدعاء الدالة `Wait`، لن ينتظر البرنامج اكتمال تنفيذ الدالتين حالما ينتهي من تنفيذ باقي تعليماته؛ أي بدلًا من استخدام مبدأ "ضعهم في الخلفية وأكمل عملك وانتظرهم"، سيعتمد مبدأ "ضعهم في الخلفية وأكمل عملك وانسَ أمرهم"، وبما أن الدالة الرئيسية تنتهي بعد وقت قصير من دالة `Wait`، فهناك فرصة جيدة لأن يصل برنامجك إلى نهاية الدالة `main` ويخرج قبل انتهاء تنفيذ خيوط معالجة جو. عند حصول ذلك قد ترى بعضًا من الأرقام تُطبع من خلال الدالتين، لكن غالبًا لن ترى الخرج المتوقع كاملًا.

أنشأنا في هذا القسم برنامجًا يستخدم الكلمة المفتاحية `go` لتنفيذ دالتين على التساير وفقًا لمبدأ خيوط معالجة جو وطباعة سلسلة من الأرقام. استخدمنا أيضًا `sync.WaitGroup` لجعل البرنامج ينتظر هذه خيوط المعالجة حتى تنتهي قبل الخروج من البرنامج.

ربما نلاحظ أن الدالتين `generateNumbers` و `printNumbers` لا تعيدان أية قيم، إذ يتعذر على خيوط معالجة جو إعادة قيم مثل الدوال العادية، على الرغم من أنه بمقدورنا استخدام الكلمة المفتاحية `go` مع الدوال التي تُعيد قيم، ولكن سيتجاهل المُصرّف هذه القيم ولن تتمكن من الوصول إليها. هذا يطرح تساؤلًا؛ ماذا نفعل إذا كنا بحاجة إلى تلك القيم المُعادة (مثلًا نريد نقل بيانات من تنظيم إلى تنظيم آخر)؟ يكمن الحل باستخدام قنوات جو التي أشرنا لها في بداية الفصل، والتي تسمح لخيوط معالجة جو بالتواصل بطريقة آمنة.

33.4 التواصل بين خيوط معالجة جو بأمان من خلال القنوات

أحد أصعب أجزاء البرمجة المتسايرة هو الاتصال بأمان بين أجـزاء البرنـامج المختلفـة الـتي تعمـل في وقت واحد، فإذا لم تكن حذرًا قد تتعرض لمشاكل من نوع خاص أثناء التنفيذ. على سبيل المثال، يمكن أن يحـدث مـا يسمى سباق البيانات Data race عندما يجري تنفيذ جزأين من البرنامج على التساير، ويحاول أحدهما تحـديث متغير بينما يحاول الجزء الآخر قراءته في نفس الـوقت. عنـدما يحـدث هـذا، يمكن أن تحـدث القـراءة أو الكتابـة بترتيب غير صحيح، مما يؤدي إلى استخدام أحد أجزاء البرنامج أو كليهما لقيمة خاطئة. مثلًا، كـان من المفـترض الكتابة ثم القراءة، لكن حدث العكس، وبالتالي نكون قد قرأنا قيمة خاطئة. يأتي اسم "سباق البيانـات" من فكـرة أن عـاملين Worker يتسـابقان للوصـول إلى نفس المتغـير أو المـورد. على الـرغم من أنـه لا يـزال من الممكن مواجهة مشكلات التساير مثل سباق البيانات في لغة جو، إلا أن تصميم اللغة يُسهّل تجنبها.

إضافةً إلى خيوط معالجة جو، تُعد قنوات جو ميزةً أخرى تهدف إلى جعل التساير سهلًا وأكثر أمنًا للاستخدام. يمكن التفكير بالقناة على أنها أنبوب pipe بين تنظيمين أو أكثر، ويمكن للبيانات العبور خلالها. يضع أحـد خيـوط معالجة البيانات في أحد طرفي الأنبـوب ليتلقـاه تنظيم آخـر في الطـرف الآخـر، وتجـري معالجـة الجـزء الصـعب المتمثل في التأكد من انتقال البيانات من واحد إلى آخر بأمان نيابةً عنّا.

إنشاء قناة في جو مُشابه لإنشاء شريحة slice، باستخدام الدالة المُضمّنة `make()`. يكون التصريح من خلال استخدام الكلمة المفتاحية `chan` متبوعةً بنوع البيانات التي تريد إرسالها عـبر القنـاة؛ فمثلًا لإنشـاء قنـاة تُرسـل قيمًا من الأعداد الصحيحة، يجب أن تستخدم النوع `chan int`؛ وإذا أردت قنـاة لإرسـال قيم بـايت `[]byte`، سنكتب `chan []byte`:

```
bytesChan := make(chan []byte)
```

يمكنك -بمجرد إنشاء القناة- إرسال أو استقبال البيانات عبر القنـاة باسـتخدام العامـل `<-`، إذ يحـدد موضـع العامل `<-` بالنسبة إلى متغير القناة ما إذا كنت تقرأ من القناة أو تكتب إليها؛ فللكتابة إلى قناة نبدأ بمتغير القناة متبوعًا بالعامل `<-`، ثم القيمة التي نريد كتابتها إلى القناة:

```
intChan := make(chan int)
intChan <- 10
```

لقراءة قيمة من قناة: نبدأ بالمتغير الذي نريد وضع القيمة فيه، ثم نضع إما `=` أو `:=` لإسناد قيمة إلى المتغير متبوعًا بالعامل `<-`، ثم القناة التي تريد القراءة منها:

```
intChan := make(chan int)
intVar := <- intChan
```

للحفاظ على هاتين العمليتين في وضع سليم، تذكر أن السهم <- يشير دائمًا إلى اليسار (عكس ->)، ويشير السهم إلى حيث تتجه القيمة. في حالة الكتابة إلى قناة، يكون السهم منطلقًا من القيمة إلى القناة. أما عند القراءة من قناة، فيكون السهم من القناة إلى المتغير. على غرار الشريحة: يمكن قراءة القناة باستخدام الكلمة المفتاحية range في حلقة for. عند قراءة قناة باستخدام range، ستُقرأ القيمة التالية من القناة في كل تكرار للحلقة وتُوضع في متغير الحلقة، وستستمر القراءة من القناة حتى إغلاق القناة، أو الخروج من الحلقة بطريقة ما، مثل استخدام تعليمة break:

```
intChan := make(chan int)
for num := range intChan {
  // Use the value of num received from the channel
  if num < 1 {
        break
  }
}
```

قد نرغب أحيانًا في السماح لدالة فقط بالقراءة أو الكتابة من القناة وليس كلاهما. لأجل ذلك يمكننا أن نضيف العامل <- إلى تصريح القناة chan. يمكننا استخدام المعامل <- بطريقة مشابهة لعملية القراءة والكتابة من قناة للسماح فقط بالقراءة أو الكتابة أو القراءة والكتابة معًا. على سبيل المثال، لتعريف قناة للقراءة فقط لقيم int، سيكون التصريح <-chan int:

```
func readChannel(ch <-chan int) {
  // ch is read-only
}
```

لجعل القناة للكتابة فقط، سيكون التصريح chan<- int:

```
func writeChannel(ch chan<- int) {
  // ch is write-only
}
```

لاحظ أن السهم يُشير إلى خارج القناة للقراءة ويشير إلى القناة من أجل الكتابة. إذا لم يكن للتصريح سهم، كما في حالة chan int، فهذا يعني أنه يمكن استخدام القناة للقراءة والكتابة.

أخيرًا، عند انتهاء الحاجة من استخدام القناة، يمكن إغلاقها باستخدام الدالة close()، وتُعد هذه الخطوة ضرورية، فقد يؤدي إنشاء القنوات وتركها دون استخدام عدة مرات في أحد البرامج ما يُعرف باسم تسرب الذاكرة Memory leak، إذ يحدث تسرب للذاكرة عندما يُنشئ برنامج ما شيئًا ما يستخدم ذاكرة الحاسب، لكنه لا يحرّر تلك الذاكرة بعد الانتهاء من استخدامها، وهذا من شأنه أن يُبطئ تنفيذ البرنامج والحاسب عمومًا؛ فعند إنشاء

قناة باستخدام make()، يخصص نظام التشغيل جزءًا من ذاكرة الحاسب للقناة، وتحرّر هذه الذاكرة عند استدعاء الدالة close() ليُتاح استخدامها من قبل شيء آخر.

لنحدّث الآن ملف main.go لاستخدام قناة int chan للتواصل بين خيوط معالجة جـو. ستنشـئ الدالـة generateNumbers أرقامًا وتكتبها على القناة بينمـا سـتقرأ الدالـة printNumbers هـذه الأرقـام من القنـاة وتطبعها على الشاشة. سننشئ في الدالة main قناةً جديدة لتمريرها مثل معامل لكل دالـة من الـدوال الأخـرى، ثم تسـتخدم close() على القنـاة لإغلاقهـا لعـدم الحاجـة لاسـتخدامها بعـد ذلـك. يجب ألا تكـون دالـة generateNumbers تنظيم جو بعد ذلك، لأنه بمجرد الانتهاء من تنفيذ هذه الدالة، سينتهي البرنامج من إنشاء جميـع الأرقـام الـتي يحتاجهـا. تُسـتدعى دالـة close() بهـذه الطريقـة على القنـاة فقـط قبـل انتهـاء تشـغيل كلتا الدالتين.

```
package main
import (
  "fmt"
  "sync"
)
func generateNumbers(total int, ch chan<- int, wg *sync.WaitGroup) {
  defer wg.Done()
  for idx := 1; idx <= total; idx++ {
        fmt.Printf("sending %d to channel\n", idx)
        ch <- idx
  }
}
func printNumbers(ch <-chan int, wg *sync.WaitGroup) {
  defer wg.Done()
  for num := range ch {
        fmt.Printf("read %d from channel\n", num)
  }
}
func main() {
  var wg sync.WaitGroup
  numberChan := make(chan int)
  wg.Add(2)
  go printNumbers(numberChan, &wg)
  generateNumbers(3, numberChan, &wg)
  close(numberChan)
```

```
    fmt.Println("Waiting for goroutines to finish...")
    wg.Wait()
    fmt.Println("Done!")
}
```

تستخدم أنواع chan في معاملات الدالتين generateNumbers و printNumbers أنواع القراءة فقط والكتابة فقط، إذ تحتاج الدالة generateNumbers إلى القدرة على الكتابة في القناة وبالتالي نجعل chan للكتابة فقط من خلال جعل السهم <- يشير إلى القناة، بينما printNumbers تحتاج للقراءة فقط من خلال جعل السهم <- يشير إلى خارج القناة.

على الرغم من أنه كان بإمكاننا جعل الدوال تستطيع القراءة والكتابة وليس فقط واحدًا منهما من خلال استخدام chan int، إلا أنه من الأفضل تقييدهم وفقًا لما تحتاجه كل دالة، وذلك لتجنب التسبب بطريق الخطأ في توقف برنامجك عن العمل والوقوع في حالة الجمود أو التوقف التام deadlock. تحدث حالة الجمود عندما ينتظر جزء A من البرنامج جزءًا آخر B لفعل شيء ما، لكن الجزء B هذا ينتظر أيضًا الجزء A لفعل شيء ما. هنا أصبح كل منهما ينتظر الآخر وبالتالي فإن كلاهما سيبقى منتظرًا ولن ينتهي من التنفيذ، ولن يكتمل تنفيذ البرنامج أيضًا.

قد يحدث الجمود بسبب الطريقة التي تعمل بها اتصالات القنوات في لغة جو، فعندما يكتب جزء من البرنامج إلى قناة، فإنه سينتظر حتى يقرأ جزءًا آخر من البرنامج من تلك القناة قبل المتابعة، وبالمثل، إذا كان البرنامج يقرأ من قناة، فإنه سينتظر حتى يكتب جزءًا آخر من البرنامج على تلك القناة قبل أن يستمر. عندما يكتب جزء A على القناة، سينتظر الجزء B أن يقرأ ما كتبه على القناة قبل أن يستمر في عمله. بطريقة مشابهة إذا كان الجزء A يقرأ من قناة، سينتظر حتى يكتب الجزء B قبل أن يستمر في عمله. يُقال أن جزءًا من البرنامج "محظور Blocking" عندما ينتظر حدوث شيء آخر، وذلك لأنه ممنوع من المتابعة حتى يحدث ذلك الشيء. تُحظر القنوات عند الكتابة إليها أو القراءة منها، لذلك إذا كانت لدينا دالة نتوقع منها أن تكتب إلى القناة ولكنها عن طريق الخطأ تقرأ من القناة، فقد يدخل البرنامج في حالة الجمود لأن القناة لن يُكتب فيها إطلاقًا. لضمان عدم حدوث ذلك نستخدم أسلوب القراءة فقط <-chan int أو الكتابة فقط chan<- int بدلًا من أسلوب القراءة والكتابة معًا chan int.

أحد الجوانب المهمة الأخرى للشيفرة المحدّثة هو استخدام close() لإغلاق القناة بمجرد الانتهاء من الكتابة عليها عن طريق generateNumbers، إذ يؤدي استدعاء الدالة close() في البرنامج السابق إلى إنهاء حلقة for ... range في دالة printNumbers. نظرًا لأن استخدام range للقراءة من القناة يستمر حتى تُغلق القناة التي يُقرأ منها، بالتالي إذا لم تُستدعى close على numberChan فلن تنتهي printNumbers أبدًا، وفي هذه الحالة لن يُستدعى التابع Done الخاص بـ WaitGroup إطلاقًا من خلال defer عند الخروج من

printNumbers، وإذا لم يُستدعى، لن ينتهي تنفيذ البرنامج، لأن التابع Wait الخاص بـ WaitGroup في الدالة main لن يستمر. هذا مثال آخر على حالة الجمود، لأن الدالة main تنتظر شيئًا لن يحدث أبدًا.

نفّذ الآن ملف main.go باستخدام الأمر go run:

```
$ go run main.go
```

قد يختلف الخرج قليلًا عما هو معروض أدناه، ولكن يجب أن يكون متشابهًا:

```
sending 1 to channel
sending 2 to channel
read 1 from channel
read 2 from channel
sending 3 to channel
Waiting for functions to finish...
read 3 from channel
Done!
```

يُظهر خرج البرنامج السابق أن الدالة generateNumbers تولّد الأرقام من واحد إلى ثلاثة أثناء كتابتها على القناة المشتركة مع printNumbers. حالما تستقبل printNumbers الرقم تطبعه على شاشة الخرج، وبعد أن تولّد generateNumbers الأرقام الثلاثة كلها، سيكون قد انتهى تنفيذها وستخرج، سامحةً بذلك للدالة main بإغلاق القناة والانتظار ريثما تنتهيprintNumbers. بمجرد أن تنتهي printNumbers من طباعة الأرقام، تستدعي Done الخاصة بـ WaitGroup وينتهي تنفيذ البرنامج.

على غرار نتائج الخرج التي رأيناها سابقًا، سيعتمد الخرج الذي تراه على عوامل خارجية مختلفة، مثل عندما يختار نظام التشغيل أو مُصرّف لغة جو تشغيل تنظيم أو عامل معين قبل الآخر أو يبدّل بينهما، ولكن يجب أن يكون الخرج متشابهًا عمومًا.

تتمثل فائدة تصميم البرامج باستخدام خيوط معالجة جو والقنوات في أنه بمجرد تصميم البرامج بطريقة تقبل التقسيم، يمكنك توسيع نطاقه ليشمل المزيد من خيوط معالجة جو. بما أن generateNumbers يكتب فقط على القناة، فلا يهم عدد الأشياء الأخرى التي تقرأ من تلك القناة، إذ سيرسل فقط أرقامًا إلى أي شيء يقرأ القناة. يمكنك الاستفادة من ذلك عن طريق تشغيل أكثر من دالة printNumbers مثل تنظيم جو، بحيث يقرأ كل منها من نفس القناة ويتعامل مع البيانات في نفس الوقت.

الآن، بعد أن استخدم البرنامج قنوات للتواصل، نفتح ملف main.go مرةً أخرى لنُحدّث البرنامج بطريقة تمكننا من استخدام عدة دوال printNumbers على أنها خيوط معالجة جو. سنحتاج إلى تعديل استدعاء wg.Add، بحيث نضيف واحدًا لكل تنظيم نبدأه، ولا داعٍ لإضافة واحد إلىWaitGroup من أجل استدعاء

generateNumbers بعد الآن، لأن البرنامج لن يستمر دون إنهاء تنفيذ كامل الدالة، على عكس ما كان يحدث عندما كنا ننفذه مثل تنظيم.

للتأكد من أن هذه الطريقة لا تقلل من عدد WaitGroup عند انتهائها، يجب علينا إزالة سطر defer wg.Done() من الدالة. تسهّل إضافة رقم التنظيم إلى printNumbers رؤية كيفية قراءة القناة بواسطة كل منهم. تُعد زيادة كمية الأرقام التي تُنشأ فكرةً جيدة أيضًا بحيث يسهل تتبعها:

```
func generateNumbers(total int, ch chan<- int, wg *sync.WaitGroup) {
  for idx := 1; idx <= total; idx++ {
        fmt.Printf("sending %d to channel\n", idx)
        ch <- idx
  }
}
func printNumbers(idx int, ch <-chan int, wg *sync.WaitGroup) {
  defer wg.Done()
  for num := range ch {
        fmt.Printf("%d: read %d from channel\n", idx, num)
  }
}
func main() {
  var wg sync.WaitGroup
  numberChan := make(chan int)
  for idx := 1; idx <= 3; idx++ {
        wg.Add(1)
        go printNumbers(idx, numberChan, &wg)
  }
  generateNumbers(5, numberChan, &wg)
  close(numberChan)
  fmt.Println("Waiting for goroutines to finish...")
  wg.Wait()
  fmt.Println("Done!")
}
```

بعد تحديث ملف main.go، يمكنك تشغيل البرنامج مرةً أخرى باستخدام go run. ينبغي أن يبدأ البرنامج بإنشاء ثلاثة خيوط معالجة للدالة printNumbers قبل المتابعة وتوليد الأرقام، كما ينبغي أن يُنشئ البرنامج أيضًا خمسة أرقام بدلًا من ثلاثة لتسهيل رؤية الأرقام موزعة بين كل من خيوط المعالجة الثلاثة للدالة printNumbers:

```
$ go run main.go
```

قد يبدو الخرج مشابهًا لهذا (قد يختلف الخرج قليلًا):

```
sending 1 to channel
sending 2 to channel
sending 3 to channel
3: read 2 from channel
1: read 1 from channel
sending 4 to channel
sending 5 to channel
3: read 4 from channel
1: read 5 from channel
Waiting for goroutines to finish...
2: read 3 from channel
Done!
```

بالنظر إلى الخرج في هذه المرة، هناك احتمال كبير ليختلف الخرج عن الناتج الذي تـراه أعلاه، لأنـه هنـاك 3 خيوط معالجة من الدالة `printNumbers` تُنفّذ، وأيٌّ منها قد يقرأ أحد الأرقام المولّدة، وبالتالي هناك احتمـالات خرج عديدة. عندما يتلقى أحد خيوط معالجة الدالة `printNumbers` رقمًا، فإنه يقضي وقتًـا قصـيرًا في طباعـة هذا الرقم على الشاشة، وفي نفس الوقت يكون هناك تنظيم آخـر يقـرأ الـرقم التـالي من القنـاة ويفعـل الشـيء نفسه. عندما ينتهي تنظيم من قراءة الـرقم الـذي اسـتقبله وطباعتـه على الشاشـة، سـيذهب للقنـاة مـرةً أخـرى ويحاول قراءة رقم آخر وطباعته، وإذا لم يجد رقمًا جديدًا، فسيبدأ الحظر حتى يمكن قراءة الرقم التـالي. بمجـرد أن تنتهي الدالـة `generateNumbers` من التنفيـذ وتسـتدعى الدالـة `close()` على القنـاة، سـتغلِق كـل خيـوط معالجة الدالة `printNumbers` حلقاتها وتخرج، وعندما تخرج جميع خيـوط المعالجـة الثلاثـة وتسـتدعي `Done` من `WaitGroup`، يصل عدّاد `WaitGroup` إلى الصفر وينتهي البرنامج. يمكنك أيضًا تجربة زيادة أو إنقاص عدد خيوط المعالجة أو الأرقام التي تُنشأ لمعرفة كيف يؤثر ذلك على الخرج.

عند استخدام خيوط معالجة جو، تجنب أن تُكثر منها؛ فمن الناحيـة النظريـة يمكن أن يحتـوي البرنـامج على مئات أو حتى الآلاف من خيوط المعالجة، وهذا مـا قـد يكـون لـه تـأثير عكسـي على الأداء، فقـد يُبطئ برنامجـك والحاسب والوقوع في حالة مجاعة الموارد Resource Starvation. في كل مـرة تُنفّـذ فيهـا لغـة جـو تنظيمًـا، يتطلب ذلك وقتًا إضافيًا لبدء التنفيذ من جديد، إضـافةً إلى الـوقت اللازم لتشـغيل الشـيفرة في الدالـة التاليـة، وبالتالي من الممكن أن يستغرق الحاسب وقتًا أطول في عمليـة التبـديل بين خيـوط المعالجـة مقارنـةً تشـغيل التنظيم نفسه، وهذا ما نسميه مجاعة الموارد، لأن البرنامج وخيوط المعالجة لا تأخـذ المـوارد الكافيـة للتنفيـذ أو ربما لا تحصل على أية موارد، وفي هذه الحالة يكون من الأفضل تخفيض عدد خيـوط المعالجـة (أجـزاء الشـيفرة

التي تعمل بالتساير) التي ينفذها البرنامج، لتخفيض عبء الوقت الإضافي المُستغرق في التبديل بينها، ومنح المزيد من الوقت لتشغيل البرنامج نفسه. يُفضّل غالبًا أن يكون عدد خيوط المعالجة مساوٍ لعدد النوى الموجودة في المعالج أو ضعفها.

يتيح استخدام مزيج من خيوط معالجة جو والقنوات إمكانية إنشاء برامج قوية جدًا وقابلة للتوسع من أجل أجهزة حواسيب أكبر وأكبر. رأينا في هذا القسم أنه يمكن استخدام القنوات للتواصل بين عدد قليل من خيوط معالجة جو أو حتى آلاف دون الحاجة للكثير من التغييرات. إذا أخذنا هذا في الحسبان عند كتابة البرامج، سنتمكن من الاستفادة من التساير المتاح في لغة جو لتزويد المستخدمين بتجربة شاملة أفضل.

33.5 الخاتمة

أنشأنا في هذا الفصل برنامجًا يطبع أرقامًا على الشاشة باستخدام الكلمة المفتاحية `go` وخيوط معالجة جو التي تتيح لنا التنفيذ المتساير. بمجرد تشغيل البرنامج أنشأنا قنواتًا جديدةً تمرِّر قيمًا صحيحة `int` عبرها باستخدام `make(chan int)`، ثم استخدمنا القناة من خلال إرسال أرقام من تنظيم جو إلى تنظيم جو آخر عبرها، ليطبعها الأخير على الشاشة بدوره. أخيرًا وسّعنا البرنامج من خلال إنشاء عدة خيوط معالجة تؤدي نفس المهمة (تستقبل أرقام من القناة وتطبعها)، وكان مثالًا على كيفية استخدام القنوات وخيوط المعالجة لتسريع البرامج على أجهزة الحواسيب متعددة النوى.

خمسات

لبيع وشراء الخدمات المصغرة

أكبر سوق عربي لبيع وشراء الخدمات المصغرة
اعرض خدماتك أو احصل على ما تريد بأسعار تبدأ من 5$ فقط

تصفح الخدمات

34. إرفاق معلومات إضافية عن الأخطاء

عندما تفشل دالة في لغة جو، فإنها تُعيد قيمةً باستخدام الواجهة `error` للسماح للمُستدعي بمعالجة الخطأ. في كثير من الأحيان يستخدم المطورون الدالة `fmt.Errorf` من الحزمة `fmt` لإعادة هذه القيم. قبل الإصدار 1.13 من لغة جو، كان الجانب السلبي لاستخدام هذه الدالة هو أننا سنفقد المعلومات المتعلقة بالأخطاء.

ولحل هذه المشكلة استخدم المطورون حزمًا توفر أساليب لتغليف wrap هذه الأخطاء داخل أخطاء أخرى، أو إنشاء أخطاء مخصصة من خلال تنفيذ التابع `Error() string` على أحد أنواع خطأ `struct` الخاصة بهم. أحيانًا، يكون إنشاء هذه الأنواع من `struct` مملًا إذا كان لديك عدد من الأخطاء التي لا تحتاج إلى المعالجة الصريحة من قبل من المُستدعي.

جاء إصدار جو 1.13 مع ميزات لتسهيل التعامل مع هذه الحالات، وتتمثل إحدى الميزات في القدرة على تغليف الأخطاء باستخدام دالة `fmt.Errorf` بقيمة خطأ `error` يمكن فكها لاحقًا للوصول إلى الأخطاء المغلفة. بالتالي، أنهى تضمين دالة تغليف للأخطاء داخل مكتبة جو القياسية الحاجة إلى استخدام طرق ومكتبات خارجية. إضافةً إلى ذلك، تجعل الدالتان `errors.Is` و `errors.As` من السهل تحديد ما إذا كان خطأ محدد مُغلّف في أي مكان داخل خطأ ما، ويمنحنا الوصول إلى هذا الخطأ مباشرةً دون الحاجة إلى فك جميع الأخطاء يدويًا.

سننشئ في هذا الفصل برنامجًا يستخدم هذه الدوال من أجل إرفاق معلومات إضافية مع الأخطاء التي تُعاد من الدوال، ثم سننشئ بنية `struct` للأخطاء المخصصة والتي تدعم دوال التغليف wrapping ودوال فك التغليف unwrapping.

34.1 المتطلبات

- لتتابع هذا الفصل، تحتاج لامتلاك مساحة عمل خاصة مُهيئة وجاهزة في لغة جو، فإذا لم يكن لديك واحدة، فارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبت لغة جو Go وجهز بيئة تطوير محلية بحسب نظام تشغيلك.
- يُفضّل أن تكون على دراية بكيفية إنشاء الوحدات في لغو جو. يمكنك مراجعة فصل استخدام الوحدات Modules في لغة جو Go.
- (اختياري) قراءة فصل معالجة الأخطاء في لغة جو Go قد يكون مفيدًا للحصول على شرح أكثر تعمقًا لمعالجة الأخطاء ومعرفة طريقة معالجة حالات الانهيار في لغة جو Go. فنحن عمومًا نغطي بعض الموضوعات منها في هذا الفصل، لكن بشرح عالي المستوى.

34.2 إعادة ومعالجة الأخطاء في لغة جو

تُعد معالجة الأخطاء من الممارسات الجيدة التي تحدث أثناء تنفيذ البرامج حتى لا يراها المستخدمون أبدًا، لكن لا بُد من التعرّف عليها أولًا لمعالجتها. يمكننا في لغة جو معالجة الأخطاء في البرامج من خلال إعادة معلومات متعلقة بهذه الأخطاء من الدوال التي حدث فيها الخطأ باستخدام واجهة `interface` من نوع خاص هو `error`، إذ يسمح استخدام هكذا واجهة لأي نوع بيانات في لغة جو أن يُعاد مثل قيمة من نوع `error` طالما أن ذلك النوع لديه تابع `Error() string` معرّف. توفر مكتبة جو القياسية دوالًا، مثل `fmt.Errorf` للتعامل مع هذا الأمر وإعادة خطأ `error`.

سننشئ في هذا الفصل برنامجًا مع دالة تستخدم `fmt.Errorf` لإعادة خطأ `error`، وسنضيف أيضًا معالج خطأ للتحقق من الأخطاء التي يمكن أن ترجعها الدالة. يمكنك العودة للفصل معالجة الأخطاء في لغة جو.

لدى معظم المطورين مجلد يضعون داخله مشاريعهم، وسنستخدم في هذا الفصل مجلدًا باسم `projects`. لننشئ هذا المجلد وننتقل إليه:

```
$ mkdir projects
$ cd projects
```

سننشئ داخل مجلد `projects` مجلدًا باسم `errtutorial` لوضع البرنامج داخله:

```
$ mkdir errtutorial
```

سننتقل إليه:

```
$ cd errtutorial
```

سنستخدم الآن الأمر `go mod init` لإنشاء وحدة `errtutorial`:

```
$ go mod init errtutorial
```

نفتح الآن ملف main.go من مجلد `errtutorial` باستخدام محرر نانو nano أو أي محرر آخر تريده:

```
$ nano main.go
```

الآن، سنكتب البرنامج، الذي سيمر ضمن حلقة على الأرقام من 1 إلى 3 ويحاول أن يحدد ما إذا كان أحد هذه الأرقام صالحًا أم لا باستخدام دالة تُدعى `validateValue`؛ فإذا لم يكن الرقم صالحًا، سيستخدم البرنامج الدالة `fmt.Errorf` لتوليد قيمة من النوع `error` تُعاد منها، إذ تسمح هذه الدالة بإنشاء قيمة `error`، بحيث تكون رسالة الخطأ هي الرسالة التي تقدمها للدالة، وهي تعمل بطريقة مشابهة للدالة `fmt.Printf`، لكن تُعاد مثل خطأ بدلًا من طباعة الرسالة على الشاشة.

الآن، ستكون هناك عملية تحقق من قيمة الخطأ في الدالة الرئيسية `main`، لمعرفة ما إذا كانت القيمة هي `nil` أو لا؛ فإذا كانت `nil` ستكون الدالة نجحت في التنفيذ دون أخطاء وسنرى الرسالة `valid!`، وإلا سيُطبع الخطأ الحاصل.

لنضع الآن هذه الشيفرة داخل ملف main.go:

```
package main
import (
  "fmt"
)
func validateValue(number int) error {
  if number == 1 {
        return fmt.Errorf("that's odd")
  } else if number == 2 {
        return fmt.Errorf("uh oh")
  }
  return nil
}
func main() {
  for num := 1; num <= 3; num++ {
        fmt.Printf("validating %d... ", num)
        err := validateValue(num)
        if err != nil {
              fmt.Println("there was an error:", err)
```

```
        } else {
            fmt.Println("valid!")
        }
    }
}
```

تأخذ الدالة `validateValue` في البرنامج السابق رقمًا وتعيد خطأً اعتمادًا على ما إذا كانت القيمة المُمررة صالحة أم لا. في مثالنا العدد 1 غير صالح، وبالتالي تُعيد الدالة خطأ مع رسالة "that's odd". الرقم 2 غير صالح ويؤدي إلى إرجاع خطأ مع رسالة "uh oh". تستخدم الدالة `validateValue` الدالة `fmt.Errorf` لتوليد قيمة الخطأ المُعادة، إذ تُعد الدالة `fmt.Errorf` ملائمة لإعادة الأخطاء، لأنها تسمح بتنسيق رسالة خطأ باستخدام تنسيق مشابه للدالة `fmt.Printf` أو `fmt.Sprintf` دون الحاجة إلى تمرير هذه السلسلة `string` إلى `errors.New`.

ستبدأ حلقة `for` داخل الدالة `main()` بالمرور على الأرقام من 1 إلى 3 وتخزّن القيمة ضمن المتغير `num`. سيؤدي استدعاء `fmt.Printf` داخل متن الحلقة إلى طباعة الرقم الذي يتحقق منه البرنامج حاليًا. بعد ذلك، سيجري استدعاء الدالة `validateValue` مع تمرير المتغير `num` (الذي نريد التحقق من صلاحيته)، وتخزين نتيجة الاستدعاء هذا في المتغير `err`. أخيرًا، إذا كانت قيمة `err` ليست `nil`، فهذا يعني أن خطًأ ما قد حدث وستُطبع رسالة الخطأ باستخدام `fmt.Println`، أما إذا كانت قيمته `nil`، فهذا يعني أن الرقم صالح وسنرى على الخرج "valid!".

بعد حفظ التغييرات الأخيرة يمكن تشغيل main.go باستخدام الأمر `go run` من المجلد `errtutorial`:

```
$ go run main.go
```

سيُظهر الخرج أن البرنامج تحقق من صحة كل رقم، وأن الرقم 1 و 2 أديا إلى إظهار الأخطاء المناسبة لهما:

```
validating 1... there was an error: that's odd
validating 2... there was an error: uh oh
validating 3... valid!
```

نلاحظ أن البرنامج يحاول التحقق من الأرقام الثلاثة من خلال الدالة `validateValue`؛ ففي المرة الأولى قد حصل على قيمة غير صالحة هي 1 فطبع رسالة الخطأ "that's odd"؛ وحصل في المرة الثانية على قيمة غير صالحة أيضًا هي 2 فطبع رسالة خطأ مختلفة هي "uh oh"؛ وحصل في المرة الثالثة على الرقم 3 وهي قيمة صالحة فأعاد قيمة "valid!" وفي هذه الحالة تكون قيمة الخطأ المُعادة هي `nil` إشارةً إلى عدم حدوث أي مشكلات وأن الرقم صالح. وفقًا للطريقة التي كُتبت فيها الدالة `validateValue`، يمكننا أن نقول أنها ستُعيد `nil` من أجل أي قيمة باستثناء الرقمين 1 و 2.

استخدمنا في هذا القسم الدالة `fmt.Errorf` لإنشاء قيم خطأ مُعادة من دالة، وأضفنا مُعالج خطأ يطبع رسالة الخطأ عند حصوله من الدالة. قد يكون من المفيد أحيانًا معرفة معنى الخطأ، وليس مجرد حدوث خطأ. سنتعلم في القسم التالي كيفية تخصيص معالجة الأخطاء لحالات محددة.

34.3 معالجة أخطاء محددة باستخدام أخطاء الحارس Sentinel Errors

عندما نستقبل قيمة خطأ من دالة، فإن أبسط طريقة لمعالجة هذا الخطأ هي التحقق من قيمته إذا كانت `nil` أو لا، إذ يخبرنا هذا ما إذا كانت الدالة تتضمن خطأً، ولكن قد نرغب أحيانًا في تخصيص معالجة الأخطاء لحالة خطأ محددة. لنتخيل أن لدينا شيفرة مرتبطة بخادم بعيد، وأن معلومات الخطأ الوحيدة التي نحصل عليها هي "لديك خطأ". قد نرغب في معرفة ما إذا كان الخطأ بسبب عدم توفر الخادم، أو إذا كانت بيانات اعتماديات الاتصال connection credentials الخاصة بنا غير صالحة. إذا كنا نعلم أن الخطأ يعني أن بيانات اعتماد المستخدم خاطئة، فقد نرغب في إعلام المستخدم فورًا بذلك، ولكن إذا كان الخطأ يعني أن الخادم غير متاح، فقد نرغب في محاولة إعادة الاتصال عدة مرات قبل إعلام المستخدم. سنتمكن من خلال تحديد الاختلاف بين هذه الأخطاء من كتابة برامج أكثر قوة وسهولة في الاستخدام.

تتمثل إحدى الطرق التي يمكن من خلالها التحقق من نوع محدد من الخطأ في استخدام التابع `Error` على النوع `error` للحصول على الرسالة من الخطأ ومقارنة هذه القيمة بنوع الخطأ الذي نبحث عنه. لنتخيل أننا نريد إظهار رسالة غير الرسالة "there was an error: uh oh" التي رأيناها سابقًا عند الحصول على الخطأ "uh oh"، وإحدى الطرق لمعالجة هذه الحالة هي التحقق من القيمة المعادة من التابع `Error` كما يلي:

```
if err.Error() == "uh oh" {
  // Handle 'uh oh' error.
  fmt.Println("oh no!")
}
```

تتحقق الشيفرة أعلاه من قيمة السلسلة المُعادة من `err.Error()` لمعرفة ما إذا كانت القيمة هي "uh oh"وستجري الأمور على نحو سليم، لكن لن تعمل الشيفرة السابقة إذا كانت السلسلة النصية التي تُعبّر عن الخطأ "uh oh" مختلفة قليلًا في مكان آخر في البرنامج.

يمكن أن يؤدي التحقق من الأخطاء بهذه الطريقة أيضًا إلى تحديثات مهمة على الشيفرة إذا كانت رسالة الخطأ نفسها بحاجة إلى التحديث، إذ يجب تحديث كل مكان يجري فيه التحقق من الخطأ.

خذ على سبيل المثال الشيفرة التالية:

```
func giveMeError() error {
  return fmt.Errorf("uh h")
}
```

```
err := giveMeError()
if err.Error() == "uh h" {
  // "uh h" error code
}
```

تتضمن رسالة الخطأ هنا خطأً إملائيًا، إذ يغيب الحرف o عن السلسلة "uh oh". إذا حدث ولاحظنا هذا الخطأ وأردنا إصلاحه، يتعين علينا إصلاحه في جميع أجزاء الشيفرة الأخرى التي تتضمن جملة التحقق err.Error() == "uh oh"، وإذا نسينا أحدهم وهذا محتمل لأنه تغيير في حرف واحد فقط، فلن يعمل معالج الخطأ المخصص لأنه يتوقع "uh h" وليس "uh oh". قد نرغب في مثل هذه الحالات بمعالجة خطأ محدد بطريقة مختلفة، إذ من الشائع إنشاء متغير يكون الغرض منه الاحتفاظ بقيمة خطأ. يمكن للشيفرة بهذه الطريقة أن تتحقق من حصول هذا المتغير بدلًا من السلسلة. تبدأ أسماء هذه المتغيرات عادةً بالعبارة Err أو err للإشارة إلى أنها أخطاء. استخدم err، إذا كان الهدف استخدام الخطأ فقط داخل الحزمة المعرّف فيها، أما إذا أردت استخدامه خارج الحزمة، استخدم البادئة Err ليصبح قيمة مُصدّرة exported value على غرار ما نفعله مع الدوال أو البنى struct.

لنفترض الآن أنك كنت تستخدم إحدى قيم الخطأ هذه في المثال السابق الذي تضمن أخطاء إملائية:

```
var errUhOh = fmt.Errorf("uh h")
func giveMeError() error {
  return errUhOh
}
err := giveMeError()
if err == errUhOh {
  // "uh oh" error code
}
```

جرى هنا تعريف المتغير errUhOh على أنه قيمة الخطأ للخطأ uh oh (على الرغم من أنه يحتوي على أخطاء إملائية). تُعيد الدالة giveMeError قيمة errUhOh لأنها تريد إعلام المُستدعي بحدوث خطأ uh oh. تقارن شيفرة معالجة الخطأ بعد ذلك قيمة err التي أُعيدت من giveMeError مع errUhOh لمعرفة ما إذا كان الخطأ "uh oh" هو الخطأ الذي حدث.

حتى إذا جرى العثور على الخطأ الإملائي وجرى إصلاحه، فستظل جميع التعليمات البرمجية تعمل، لأن التحقق من الخطأ يجري من خلال المقارنة مع القيمة errUhOh، وقيمة errUhOh هي قيمة ثابتة تُعاد من الدالة giveMeError.

تُعرّف قيمة الخطأ المُراد فحصها ومقارنتها بهذه الطريقة بخطأ الحارس sentinel error، وهو خطأ مُصمّم ليكون قيمة فريدة يمكن مقارنتها دائمًا بمعنًى معين. ستحمل قيمة errUhOh السابقة دائمًا نفس المعنى

(حدوث خطأ "uh oh")، لذلك يمكن للبرنامج الاعتماد على مقارنة خطأ مع errUhOh لتحديد ما إذا كان هذا الخطأ قد حدث أم لا.

تتضمن مكتبة جو القياسية عددًا من أخطاء الحارس لتطوير برامج جو. أحد الأمثلة على ذلك هو خطأ sql.ErrNoRows، الذي يُعاد عندما لا يُعيد استعلام قاعدة البيانات أية نتائج، لذلك يمكن معالجة هذا الخطأ بطريقة مختلفة عن خطأ الاتصال. بما أنه خطأ حارس، بالتالي يمكن استخدامه في شيفرة التحقق من الأخطاء لمعرفة متى لا يُعيد الاستعلام أية صفوف rows، ويمكن للبرنامج التعامل مع ذلك بطريقة مختلفة عن لأخطاء الأخرى.

عند إنشاء قيمة خطأ حارس، تُستخدم الدالة errors.New من حزمة errors بدلًا من دالة fmt.Errorf التي كنا نستخدمها. لا يؤدي استخدام errors.New بدلًا من fmt.Errorf إلى إجراء أي تغييرات أساسية في كيفية عمل الخطأ، ويمكن استخدام كلتا الدالتين بالتبادل في معظم الأوقات. أكبر فرق بين الاثنين هو أن errors.New تنشئ خطأ مع رسالة ثابتة، بينما تسمح دالة fmt.Errorf بتنسيق الرسالة مع القيم بطريقة مشابهة لآلية تنسيق السلاسل في fmt.Printf أو fmt.Sprintf.

نظرًا لأن أخطاء الحارس هي أخطاء أساسية بقيم لا تتغير، يُعد استخدام errors.New لإنشائها شائعًا. لنحدّث الآن البرنامج السابق من أجل استخدام خطأ الحارس مع الخطأ "oh uh" بدلًا من fmt.Errorf. نفتح ملف main.go لإضافة خطأ الحارس errUhOh الجديد وتحديث البرنامج لاستخدامه.

تُحدَّث دالة validateValue بحيث تعيد خطأ الحارس بدلًا من استخدام fmt.Errorf. وتُحدّث دالة main() بحيث تتحقق من وجود خطأ حارس errUhOh وطباعة"oh no" عندما يواجهها خطأ بدلًا من رسالة "there was an error" التي تظهر لأخطاء أخرى.

```
package main
import (
  "errors"
  "fmt"
)
var (
  errUhOh = errors.New("uh oh")
)
func validateValue(number int) error {
  if number == 1 {
        return fmt.Errorf("that's odd")
  } else if number == 2 {
        return errUhOh
```

```
    }
    return nil
}
func main() {
    for num := 1; num <= 3; num++ {
        fmt.Printf("validating %d... ", num)
        err := validateValue(num)
        if err == errUhOh {
            fmt.Println("oh no!")
        } else if err != nil {
            fmt.Println("there was an error:", err)
        } else {
            fmt.Println("valid!")
        }
    }
}
```

نحفظ الشيفرة ونشغّل البرنامج كالعادة باستخدام الأمر `go run`:

```
$ go run main.go
```

سيُظهر الخرج هذه المرة ناتج الخطأ العام للقيمة 1، لكنه يستخدم الرسالة المخصصة "uh oh!" عندما تصادف الخطأ `errUhOh` الناتج من تمرير 2 إلى `validateValue`:

```
validating 1... there was an error: that's odd
validating 2... oh no!
validating 3... valid!
```

يؤدي استخدام أخطاء الحارس داخل شيفرة فحص الأخطاء إلى تسهيل التعامل مع حالات الخطأ الخاصة، إذ يمكن لأخطاء الحارس مثلًا المساعدة في تحديد ما إذا كان الملف الذي نقرأه قد فشل لأننا وصلنا إلى نهاية الملف، والذي يُشار إليه بخطأ الحارس io.EOF، أو إذا فشل لسبب آخر.

أنشأنا في هذا القسم برنامج جو يستخدم خطأ حارس باستخدام `errors.New` للإشارة إلى حدوث نوع معين من الأخطاء. بمرور الوقت ومع نمو البرنامج، قد نصل إلى النقطة التي نريد فيها تضمين مزيدٍ من المعلومات في الخطأ الخاص بنا بدلًا من مجرد قيمة الخطأ "uh oh". لا تقدم قيمة الخطأ هذه أي سياق حول مكان حدوث الخطأ أو سبب حدوثه، وقد يكون من الصعب تعقب تفاصيل الخطأ في البرامج الأكبر حجمًا.

للمساعدة في استكشاف الأخطاء وإصلاحها ولتقليل وقت تصحيح الأخطاء، يمكننا الاستفادة من تغليف الأخطاء لتضمين التفاصيل التي نحتاجها.

34.4 تغليف وفك تغليف الأخطاء

يُقصد بتغليف الأخطاء أخذ قيمة خطأ معينة ووضع قيمة خطأ أخرى داخلها، وكأنها هدية مغلفة، وبما أنها مغلفة، فهذا يعني أنك بحاجة لفك غلافها لمعرفة ما تحتويه. يمكننا من خلال تغليف الخطأ تضمين معلومات إضافية حول مصدر الخطأ أو كيفية حدوثه دون فقدان قيمة الخطأ الأصلية، نظرًا لوجودها داخل الغلاف.

قبل الإصدار 1.13 كان من الممكن تغليف الأخطاء، إذ كان بالإمكان إنشاء قيم خطأ مخصصة تتضمن الخطأ الأصلي، ولكن سيتعين علينا إما إنشاء أغلفة خاصة، أو استخدام مكتبة تؤدي الغرض نيابةً عنا. بدءًا من الإصدار 1.13 أضافت لغة جو دعمًا لعملية تغليف الأخطاء وإلغاء التغليف بمثابة جزء من المكتبة القياسية عن طريق إضافة الدالة errors.Unwrap والعنصر النائب `%w` لدالة `fmt.Errorf`.

سنحدّث خلال هذا القسم برنامجنا لاستخدام العنصر النائب `%w` لتغليف الأخطاء بمزيد من المعلومات، وبعد ذلك ستستخدم الدالة `errors.Unwrap` لاسترداد المعلومات المغلفة.

34.4.1 تغليف الأخطاء مع الدالة fmt.Errorf

كانت الدالة `fmt.Errorf` تُستخدم سابقًا لإنشاء رسائل خطأ منسقة بمعلومات إضافية باستخدام عناصر نائبة مثل `%v` للقيم المعمّمة و `%s` للسلاسل النصية، أما حديثًا (بدءًا من الإصدار 1.13) أُضيف عنصر نائب جديد هو `%w`. عندما يجري تضمين هذا العنصر ضمن تنسيق السلسلة وتتوفر قيمة للخطأ، ستتضمّن قيمة الخطأ المُعادة من `fmt.Errorf` قيمة الخطأ `error` المُغلّف في الخطأ المُنشأ.

افتح ملف main.go وحدّثه ليشمل دالةً جديدةً تسمى `runValidation`. ستأخذ هذه الدالة الرقم الذي يجري التحقق منه حاليًا وستُشغّل أي عملية تحقق مطلوبة على هذا الرقم، وفي حالتنا ستحتاج إلى تنفيذ الدالة `runValidation` فقط. إذا واجه البرنامج خطأً في التحقق من القيمة، سيغلِّف الخطأ باستخدام `fmt.Errorf` والعنصر النائب `%w` لإظهار حدوث خطأ في التشغيل، ثم يعيد هذا الخطأ الجديد. ينبغي أيضًا تحديث الدالة `main`، فبدلًا من استدعاء `validateValue` مباشرةً، نستدعي `runValidation`:

```
...
var (
  errUhOh = errors.New("uh oh")
)
func runValidation(number int) error {
  err := validateValue(number)
  if err != nil {
```

```
        return fmt.Errorf("run error: %w", err)
    }
    return nil
}
...
func main() {
    for num := 1; num <= 3; num++ {
        fmt.Printf("validating %d... ", num)
        err := runValidation(num)
        if err == errUhOh {
            fmt.Println("oh no!")
        } else if err != nil {
            fmt.Println("there was an error:", err)
        } else {
            fmt.Println("valid!")
        }
    }
}
```

يمكننا الآن -بعد حفظ التحديثات- تشغيل البرنامج:

```
$ go run main.go
```

سيظهر خرج مشابه لما يلي:

```
validating 1... there was an error: run error: that's odd
validating 2... there was an error: run error: uh oh
validating 3... valid!
```

هناك عدة أشياء يمكن ملاحظتها من هذا الخرج. إذ نرى أولًا رسالة الخطأ للقيمة 1 مطبوعـةً الآن ومتضـمنة في رسـالة الخطـأ. يوضـح هـذا أن الخطـأ جـرى تغليفـه بواسـطة الدالـة `run error: that's odd` الخاصة بالدالة `fmt.Errorf` وأن قيمة الخطأ التي جرى تغليفهـا `runValidation` مُضـمّنة `that's odd` في رسالة الخطأ.

هناك مشكلة، إذ أن معالج الخطأ الخاص الـذي أضـفناه إلى خطـأ `errUhOh` لم يُنفّـذ، وسـنرى في السـطر الثاني من الخرج والذي يتحقق من صلاحية الرقم 2 -رسالة الخطأ الافتراضية للقيمة 2 وهي:

```
there was an error: run error: uh oh
```

بدلًا من الرسالة المتوقعة "uh no!" نحن نعلم أن الدالة `ValidateValue` لا تزال تُعيد الخطأ "uh oh"، إذ يمكننا رؤية ذلك في نهاية الخطأ المُغلف، لكن معالج الخطأ في `errUhOh` لم يعد يعمل. يحدث هذا لأن الخطأ الذي أُعيد من الدالة `runValidation` لم يعد الخطأ `errUhOh`، وإنما الخطأ المغلف الذي أُنشئ بواسطة الدالة `fmt.Errorf`. عندما تحاول الجملة الشرطية `if` مقارنة متغير `err` مع `errUhOh`، فإنها تُعيد خطأ لأن `errUhOh` لم يعد مساويًا للخطأ الذي يُغلّف `errUhOh`، ولحل هذه المشكلة يجب الحصول الخطأ من داخل الغلاف عن طريق فك التغليف باستخدام دالة `errors.Unwrap`.

34.4.2 فك تغليف الأخطاء باستخدام errors.Unwrap

إضافةً إلى العنصر النائب `%w` في الإصدار 1.13، أُضيفت بعض الدوال الجديدة إلى حزمة الأخطاء `errors`. واحدة من هذه الدوال هي الدالة `errors.Unwrap`، التي تأخذ خطأ `error` مثل معامل، وإذا كان الخطأ المُمرّر مُغلّف خطأ، ستعيد الخطأ المُغلّف، وإذا لم يكن الخطأ المُمرّر غلافًا تُعيد `nil`.

نفتح الآن ملف main.go، وباستخدام الدالة `errors.Unwrap` سنحدّث آلية التحقق من خطأ `errUhOh` لمعالجة الحالة التي يجري فيها تغليف `errUhOh` داخل مغلّف خطأ:

```
func main() {
  for num := 1; num <= 3; num++ {
        fmt.Printf("validating %d... ", num)
        err := runValidation(num)
        if err == errUhOh || errors.Unwrap(err) == errUhOh {
              fmt.Println("oh no!")
        } else if err != nil {
              fmt.Println("there was an error:", err)
        } else {
              fmt.Println("valid!")
        }
  }
}
```

شغّل البرنامج:

```
$ go run main.go
```

لتكون النتيجة على النحو التالي:

```
validating 1... there was an error: run error: that's odd
validating 2... oh no!
```

```
validating 3... valid!
```

سنرى الآن في السطر الثاني من الخرج أن خطأ "uh oh!" للقيمة 2 قد ظهر. يسمح الاستدعاء الإضافي للدالة `errors.Unwrap` الذي أضفناه إلى تعليمة `if` باكتشاف `errUhOh` عندما تكون `err` هي قيمة خطأ `errUhOh` وكذلك إذا كان `err` هو خطأ يُغلّف خطأ `errUhOh` مباشرةً.

استخدمنا في هذا القسم العنصر `%w` المضاف إلى `fmt.Errorf` لتغليف الخطأ `errUhOh` داخل خطأ آخر وإعطائه معلومات إضافية. استخدمنا بعد ذلك `errors.Unwrap` للوصول إلى الخطأ `errorUhOh` المُغلّف داخل خطأ آخر. يُعد تضمين أخطاء داخل أخطاء أخرى مثل قيم ضمن سلسلة `string` أمرًا مقبولًا بالنسبة للأشخاص الذين يقرؤون رسائل الخطأ، ولكن قد ترغب أحيانًا في تضمين معلومات إضافية مع غلاف الأخطاء لمساعدة البرنامج في معالجة الخطأ، مثل رمز الحالة status code في خطأ طلب HTTP، وفي هكذا حالة يمكنك إنشاء خطأ مخصص جديد لإعادته. يمكنك الاطلاع على مقال كيفية استكشاف وإصلاح رموز أخطاء HTTP الشائعة على أكاديمية حسوب لمزيدٍ من المعلومات حول رموز حالة أخطاء HTTP الشائعة.

34.5 أخطاء مغلفة مخصصة

بما أن القاعدة الوحيدة للواجهة `error` في لغة جو هي أن تتضمن تابع `Error`، فمن الممكن تحويل العديد من أنواع لغة جو إلى خطأ مخصص، وتتمثل إحدى الطرق في تعريف نوع بنية `struct` مع معلومات إضافية حول الخطأ وتضمين تابع `Error`.

بالنسبة لخطأ التحقق Validation error، سيكون من المفيد معرفة القيمة التي تسببت بالخطأ. لننشئ الآن بنيةً جديدة اسمها `ValueError` تحتوي على حقل من أجل القيمة `Value` التي تسبب الخطأ وحقل `Err` يحتوي على الخطأ الفعلي. تستخدم عادةً أنواع الأخطاء المخصصة اللاحقة `Error` في نهاية اسم النوع، للإشارة إلى أنه نوع يتوافق مع الواجهة `error`. افتح الآن ملف main.go وأضِف البنية الجديدة `ValueError`، إضافةً إلى دالة `newValueError` لتُنشئ نسخًا من هذه البنية.

نحتاج أيضًا إلى إنشاء تابع يُسمى `Error` من أجل البنية `ValueError` لكي تُعد من النوع `error`. يجب أن يُعيد التابع `Error` القيمة التي تريد عرضها عندما يجري تحويل الخطأ إلى سلسلة نصية. نستخدم في حالتنا الدالة `fmt.Sprintf` لإعادة سلسلة نصية تعرض `error: value` ثم الخطأ المُغلّف. حدِّث الدالة `ValidateValue`، فبدلًا من إعادة الخطأ الأساسي فقط، ستستخدم الدالة `newValueError` لإعادة خطأ مخصص:

```
...
var (
  errUhOh = fmt.Errorf("uh oh")
)
```

```
type ValueError struct {
  Value int
  Err  error
}
func newValueError(value int, err error) *ValueError {
  return &ValueError{
        Value: value,
        Err:  err,
  }
}
func (ve *ValueError) Error() string {
  return fmt.Sprintf("value error: %s", ve.Err)
}
...
func validateValue(number int) error {
  if number == 1 {
        return newValueError(number, fmt.Errorf("that's odd"))
  } else if number == 2 {
        return newValueError(number, errUhOh)
  }
  return nil
}
...
```

لنُشغّل البرنامج الآن:

```
$ go run main.go
```

ستكون النتيجة على النحو التالي:

```
validating 1... there was an error: run error: value error: that's odd
validating 2... there was an error: run error: value error: uh oh
validating 3... valid!
```

يظهر الناتج الآن أن الأخطاء مُغلّفة داخل `ValueError` من خلال عرض `value error:` قبلها، لكن نلاحظ أن خطأ "uh oh" لم يُكتشف مرةً أخرى لأن `errUhOh` داخل طبقتين من الأغلفة الآن، هما: `ValueError` وغلاف `fmt.Errorf` من `runValidation`. تُستخدم الدالة `errors.Unwrap` مرةً واحدةً فقط على الخطأ، لذلك ينتج عن استخدام `errors.Unwrap(err)` القيمة `*ValueError` وليس

errUhOh. تتمثل إحدى الحلول في تعديل عملية التحقق من errUhOh بإضافة استدعاء errors.Unwrap() مرتين لفك كلتا الطبقتين.

نفتح الآن ملف main.go ونُعدّل الدالة main() لإضافة التعديل:

```
...
func main() {
  for num := 1; num <= 3; num++ {
        fmt.Printf("validating %d... ", num)
        err := runValidation(num)
        if err == errUhOh ||
              errors.Unwrap(err) == errUhOh ||
              errors.Unwrap(errors.Unwrap(err)) == errUhOh {
              fmt.Println("oh no!")
        } else if err != nil {
              fmt.Println("there was an error:", err)
        } else {
              fmt.Println("valid!")
        }
  }
}
```

لنُشغّل البرنامج الآن:

```
$ go run main.go
```

ستكون النتيجة على النحو التالي:

```
validating 1... there was an error: run error: value error: that's odd
validating 2... there was an error: run error: value error: uh oh
validating 3... valid!
```

معالجة الأخطاء الخاصة errUhOh لا تعمل، وكنا نتوقع ظهور خرج معالج الخطأ الخاص "oh no!" من أجل سطر التحقق الثاني لكن ما زالت الرسالة الافتراضية "there was an error: run error:" تظهر بدلًا منها. هذا يحدث لأن errors.Unwrap لا تعرف كيفية فك تغليف الخطأ المخصص ValueError. يجب أن يكون لدى الخطأ المخصص تابع فك تغليف Unwrap خاص، يعيد الخطأ الداخلي مثل قيمة error. فعندما كنا نُنشئ أخطاءً باستخدام fmt.Errorf مع العنصر النائب %w، كان مُصرّف جو يُنشئ خطأ مع تابع تغليف Unwrap()

تلقائيًا، لذلك لم نكن بحاجة إلى إضافة التابع يدويًا، لكننا هنا نستخدم دالةً خاصة، وبالتالي يجب أن نُضيف هـذا التابع يدويًا.

إذًا، لإصلاح مشكلة `errUhOh` نفتح ملف main.go ونضيف التـابع `Unwrap()` إلى `ValueError` الـتي تُعيد الحقل `Err` الذي يحتوي على الخطأ الداخلي المغلف:

```
...
func (ve *ValueError) Error() string {
  return fmt.Sprintf("value error: %s", ve.Err)
}
func (ve *ValueError) Unwrap() error {
  return ve.Err
}
...
```

لنُشغّل البرنامج الآن:

```
$ go run main.go
```

ستكون النتيجة كما يلي:

```
validating 1... there was an error: run error: value error: that's odd
validating 2... oh no!
validating 3... valid!
```

نلاحظ أن المشكلة قد حُلت وظهرت رسالة الخطأ "uh no!" التي تُشير إلى الخطأ `errUhOh` هذه المرة، لأن `errors.Unwrap` أصبح قادرًا على فك تغليف `ValueError`.

أنشأنا في هذا القسم خطأ جديد مخصص `ValueError`، لتزويدنا والمستخدمين بمعلومـات إضـافية عن عملية التحقق في جزء من رسالة الخطأ. أضفنا أيضًا دعمًا للدالـة `Unwrap()` لعمليـة فـك تغليـف الأخطـاء إلى `ValueError`، بحيث يمكن استخدام `errors.Unwrap` للوصول إلى الخطأ المغلف.

يمكننا أن نلاحظ عمومًا أن معالجة الأخطاء أصبحت صعبة نوعًا ما ويصـعب الحفـاظ عليهـا، إذ يجب علينـا إضافة دالة فك تغليف `errors.Unwrap` من أجل كل طبقة تغليف جديدة. لحسـن حظنـا هنـاك دوال جـاهزة `errors.Is` و `errors.As` تجعل العمل مع الأخطاء المغلفة أسهل.

34.6 التعامل مع الأخطاء المغلفة Wrapped Errors

عند الحاجة إلى إضافة استدعاء جديد للدالة errors.Unwrap من أجل كل طبقة تغليف في البرنامج، سيستغرق الأمر وقتًا طويلًا ويصعب الحفاظ عليه. لهذا الأمر أُضيفت الدالتان errors.Is و errors.As إلى حزمة الأخطاء errors بدءًا من الإصدار 1.13 من لغة جو، إذ تعمل هاتان الدالتان على تسهيل التعامل مع الأخطاء من خلال السماح لك بالتفاعل مع الأخطاء بغض النظر عن مدى عمق تغليفها داخل الأخطاء الأخرى. تسمح الدالة errors.Is بالتحقق ما إذا كانت قيمة خطأ حارس معين موجودةً في أي مكان داخل خطأ مغلف؛ بينما تتيح الدالة errors.As الحصول على مرجع لنوع معين من الأخطاء من أي مكان داخل خطأ مغلّف.

34.6.1 فحص قيمة خطأ باستخدام الدالة errors.Is

يؤدي استخدام الدالة errors.Is للتحقق من وجود خطأ معين إلى جعل معالجة الخطأ الخاص errUhOh أقصر بكثير، لأنه يعالج جميع الأخطاء المتداخلة تلقائيًّا بدل إجرائها يدويًا من قبلنا. تأخذ الدالة معاملين كل منهما خطأ error، المعامل الأول هو الخطأ الذي تلقيناه والثاني هو الخطأ الذي نريد التحقق منه. لإزالة التعقيدات من عملية معالجة الخطأ errUhOh، سنفتح ملف main.go، ونحدّث عملية التحقق من errUhOh في دالة main لنستخدم الدالة errors.Is:

```
...
func main() {
  for num := 1; num <= 3; num++ {
        fmt.Printf("validating %d... ", num)
        err := runValidation(num)
        if errors.Is(err, errUhOh) {
              fmt.Println("oh no!")
        } else if err != nil {
              fmt.Println("there was an error:", err)
        } else {
              fmt.Println("valid!")
        }
  }
}
```

لنُشغّل البرنامج الآن:

```
$ go run main.go
```

سنحصل على النتيجة التالية:

```
validating 1... there was an error: run error: value error: that's odd
validating 2... oh no!
validating 3... valid!
```

يظهر الخرج رسالة الخطأ "uh no!"، وهذا يعني أنه على الرغم من وجود عملية فحص خطأ واحدة من أجل errUhOh، سيظل بالإمكان فك سلسلة الأخطاء والوصول إلى الخطأ المُغلّف. تستفيد الدالة errors.Is من التابع Unwrap لمواصلة البحث العميق في سلسلة من الأخطاء، حتى تعثر على قيمة الخطأ المطلوبة أو خطأ حارس أو تُصادف تابع Unwrap يعيد قيمة nil. بعد إضافة دالة errors.Is في إصدار 1.13، أصبح استخدامها موصى به للتحقق من وجود أخطاء. تجدر الإشارة إلى أنه يمكن استخدام هذه الدالة مع قيم الخطأ الأخرى، مثل خطأ sql.ErrNoRows سالف الذكر.

34.6.2 استرداد نوع الخطأ باستخدام errors.As

الدالة الثانية التي سنتحدث عنها هي errors.As، والتي تُستخدم عندما نريد الحصول على مرجع لنوع معين من الأخطاء للتفاعل معه بتفصيل أكبر. مثلًا يتيح الخطأ المخصص ValueError الذي أضفناه سابقًا الوصول إلى القيمة الفعلية التي يجري التحقق من صحتها في حقل Value الخاص بالخطأ، ولكن لا يمكن الوصول إليه إلا إذا كان لدينا مرجع لهذا الخطأ أولًا. هنا يأتي دور errors.As التي تأخذ معاملين، الأول خطأ والثانية متغير لنوع الخطأ، إذ يجري المرور على سلسلة الأخطاء لمعرفة ما إذا كان أي من الأخطاء المغلفة يتطابق مع النوع المُقدم، فإذا حصل تطابق مع أحدها فسيُضبط المتغير المُمرر لنوع الخطأ بالخطأ الذي عثر عليه errors.As، وستعيد الدالة true وإلا ستعيد false.

إذًا، أصبح بإمكاننا باستخدام هذه الدلة الاستفادة من نوع ValueError لإظهار معلومات إضافية عن الخطأ في معالج الأخطاء. لنفتح ملف main.go للمرة الأخيرة ونحدّث الدالة main() لإضافة حالة جديدة لمعالجة الأخطاء من نوع ValueError، بحيث تطبع قيمة الخطأ والرقم غير الصالح وخطأ التحقق:

```
...
func main() {
  for num := 1; num <= 3; num++ {
        fmt.Printf("validating %d... ", num)
        err := runValidation(num)
        var valueErr *ValueError
        if errors.Is(err, errUhOh) {
              fmt.Println("oh no!")
        } else if errors.As(err, &valueErr) {
```

```
                fmt.Printf("value error (%d): %v\n", valueErr.Value,
valueErr.Err)
        } else if err != nil {
                fmt.Println("there was an error:", err)
        } else {
                fmt.Println("valid!")
        }
    }
}
```

صرّحنا في الشيفرة السابقة عن متغير جديد اسمه `valueErr` واستخدمنا `errors.As`، للحصول على مرجع إلى `ValueError` في حال كان مغلفًا داخل قيمة `err`. بمجرد الوصول إلى الخطأ الخاص `ValueError`، سنتمكن من الوصول إلى أية حقول إضافية يوفرها هذا النوع، مثل القيمة الفعلية التي فشل التحقق منها. قد يكون هذا مفيدًا إذا كانت عملية التحقق عميقة في البرنامج ولم يكن لدينا إمكانية الوصول إلى القيم لمنح المستخدمين تلميحات حول مكان حدوث خطأ ما. مثال آخر على المكان الذي يمكن أن يكون مفيدًا فيه هو في حال كنا في صدد برمجة شبكة وواجهنا خطأ net.DNSError. يمكنك -من خلال الحصول على مرجع للخطأ- معرفة ما إذا كان الخطأ ناتجًا عن عدم القدرة على الاتصال، أو ما إذا كان الخطأ ناتجًا عن القدرة على الاتصال، ولكن لم يُعثر على المورد المطلوب، وهذا يمكّننا من التعامل مع الخطأ بطرق مختلفة.

لرؤية كيف تجري الأمور مع الدالة `errors.As` دعونا نُشغّل البرنامج:

```
$ go run main.go
```

ستكون النتيجة على النحو التالي:

```
validating 1... value error (1): that's odd
validating 2... oh no!
validating 3... valid!
```

لن ترى رسالة الخطأ الافتراضي "there was an error:" هذه المرة في الخرج، لأن جميع الأخطاء تُعالج بواسطة معالجات الأخطاء الأخرى. يظهر السطر الأول من الخرج الخاص بالتحقق من صحة القيمة 1 أن الدالة `errors.As` تُعيد `true` لأن رسالة الخطأ "value error ..." عُرضت. بما أن الدالة `errors.As` تُعيد `true`، يُضبط المتغير `valueErr` ليكون خطأ `ValueError` ويمكن استخدامه لطباعة القيمة التي فشلت في التحقق من الصحة من خلال الوصول إلى `valueErr.Value` (لاحظ كيف طُبعت القيمة 1 والتي فشلت في اختبار التحقق من الصحة).

يظهر أيضًا السطر الثاني من الخرج، والذي يشير إلى اختبار التحقق من صحة الرقم 2، أنه على الرغم من تغليف `errUhOh` داخل غلاف `ValueError`، ما زالت رسالة الخطأ "oh no!" تظهر (بالرغم من وجود طبقتي تغليف)، وهذا لأن معالج الأخطاء الخاص الذي يستخدم الدالة `errors.Is` مع `errUhOh` تأتي أولًا في مجموعة تعليمات اختبار `if` لمعالجة الأخطاء. بما أن هذا المعالج يُعيد `true` قبل تنفيذ `errors.As`، يُنفّذ معالج "oh no!". لو كانت الدالة `errors.As` تظهر قبل الدالة `errors.Is` في البرنامج لرأينا خرجًا مشابهًا لحالة القيمة 1، أي أن رسالة "oh no!" ستكون"value error (2): uh oh".

حدّثنا خلال هذا القسم البرنامج لنتمكن من استخدام `errors.Is` لإزالة الاستدعاءات الكثيرة والإضافية للدالة `errors.Unwrap` وجعل شيفرة معالجة الأخطاء أكثر متانة وسهولة للتعديل. استخدمنا أيضًا الدالة `errors.As` للتحقق ما إذا كان أي من الأخطاء المغلّفة هو `ValueError`، ثم استخدمنا حقولها في حال توافرها.

34.7 الخاتمة

تعرّفنا خلال هذا الفصل على كيفية تغليف خطأ باستخدام العنصر النائب `%w` وكيفية فكه باستخدام الدالة `errors.Unwrap`. أنشأنا أيضًا نوع خطأ مخصص يدعم الدالة `errors.Unwrap`، واستخدمناه لاستكشاف دوال مُساعدة جديدة `errors.Is` و `errors.As` تُسهّل علينا عملية إرفاق معلومات أعمق وأكثر تفصيلًا عن الأخطاء التي نُنشئها، أو نتعامل معها وتضمن استمرارية عملية فحص الأخطاء حتى في حالة الأخطاء العميقة.

دورة إدارة تطوير المنتجات

تعلم تحويل أفكارك لمنتجات ومشاريع حقيقية بدءًا من دراسة السوق
وتحليل المنافسين وحتى إطلاق منتج مميز وناجح

التحق بالدورة الآن

35. التعامل مع التاريخ والوقت

صُممت البرمجيات لتسهيل إنجاز الأعمال، وفي بعض الأحيان يتطلب الأمر التعامل مع التاريخ والوقت، إذ يمكن رؤية قيم التاريخ والوقت في معظم البرامج الحديثة، فمثلًا لو أردنا تطوير تطبيق يعطينا تنبيهات عن أوقات الصلاة، سيتوجب على البرنامج تشغيل تنبيه عند وقت وتاريخ محدد. مثال آخر هو تتبع الوقت في السيارات الحديثة، إذ نحتاج إلى إرسال تنبيهات إلى مالك السيارة لإبلاغه عن وجود مشكلة أو حاجة معينة للسيارة، أو تتبع التغييرات في قاعدة بيانات لإنشاء سجل تدقيق، أو الوقت الذي يتطلبه إنجاز عملية ما مثل عبور شارع معين للوصول إلى الوجهة ...إلخ. يشير هذا إلى ضرورة التعامل مع التاريخ والوقت في البرامج والتفاعل معها وعرضها على المستخدمين بتنسيق واضح وسهل الفهم، فهذه خاصية أساسية لهكذا تطبيقات.

سنُنشئ خلال هذا الفصل برنامج جو يمكنه معرفة التوقيت المحلي الحالي من خلال جهاز الحاسب الذي يعمل عليه، ثم عرضه على الشاشة بتنسيق سهل القراءة. بعد ذلك، سنفسّر سلسلة نصية لاستخراج معلومات التاريخ والوقت، كما سنغيّر أيضًا قيم التاريخ والوقت من منطقة زمنية إلى أخرى، ونضيف أو نطرح قيم الوقت لتحديد الفاصل الزمني بين وقتين.

35.1 المتطلبات

لتتابع هذا الفصل، ستحتاج إلى إصدار مُثبّت من جو 1.16 أو أعلى، ويمكنك الاستعانة بالتعليمات الواردة في الفصل الأول من الكتاب، لثبيت لغة جو Go وإعداد بيئة تطوير محلية بحسب نظام تشغيلك.

35.2 الحصول على التاريخ والوقت الحالي

سنستخدم خلال هذا القسم حزمة لغة جو `time` للوصول إلى قيم التاريخ والوقت الحالي. تُقدّم لنا هذه الحزمة القياسية من مكتبة لغة جو القياسية -العديد من الدوال المتعلقة بالتاريخ والوقت، والتي يمكن من خلالها تمثيل نقاط معينة من الوقت باستخدام النوع `time.Time`. يمكننا أيضًا من خلال هذه الدوال التقاط بعض المعلومات عن المنطقة الزمنية التي تمثِّل التاريخ والوقت المقابل.

كما هي العادة، سنحتاج لبدء إنشاء برامجنا إلى إنشاء مجلد للعمل ووضع الملفات فيه. يمكن أن نضع المجلد في أي مكان على الحاسب، إذ يكون للعديد من المبرمجين عادةً مجلدٌ يضعون داخله كافة مشاريعهم. سنستخدم في هذا الفصل مجلدًا باسم `projects`، لذا فلننشئ هذا المجلد وننتقل إليه:

```
$ mkdir projects
$ cd projects
```

من داخل هذا المجلد، سنشغّل الأمر `mkdir` لإنشاء مجلد `datetime` ثم سنستخدم `cd` للانتقال إليه:

```
$ mkdir datetime
$ cd datetime
```

يمكننا الآن فتح ملف `main.go` باستخدام محرر نانو nano أو أي محرر آخر تريده:

```
$ nano main.go
```

نضيف الدالة `main` التي سنكتب فيها تعليمات الحصول على التاريخ والوقت وعرضهما:

```
package main
import (
  "fmt"
  "time"
)
func main() {
  currentTime := time.Now()
  fmt.Println("The time is", currentTime)
}
```

استخدمنا الدالة `time.Now` من الحزمة `time` للحصول على الوقت الحالي مثل قيمة من النوع `time.Time` وتخزينها في المتغير `currentTime`، ثم طبعنا قيمة هذا المتغير باستخدام الدالة `fmt.Println`، إذ سيُطبع وفقًا لتنسيق سلسلة نصية افتراضي خاص بالنوع `time.Time`.

شغّل الآن ملف البرنامج main.go باستخدام الأمر `go run`:

```
$ go run main.go
```

سيبدو الخرج الذي يعرض التاريخ والوقت الحاليين مشابهًا لما يلي:

```
The time is 2021-08-15 14:30:45.0000001 -0500 CDT m=+0.000066626
```

طبعًا في كل مرة نُشغّل فيها هذا البرنامج سيكون التاريخ والوقت مختلفين، كما أن المنطقة الزمنية `-0500 CDT` مُتغيرة تبعًا للمنطقة الزمنية التي ضُبط عليها الحاسب كما سبق وأشرنا. نلاحظ وجود القيمة `m=`، التي تُشير إلى ساعة رتيبة monotonic clock، وتُستخدم ضمنيًّا في جو عند قياس الاختلافات في الوقت، وقد صُممت لتعويض التغييرات المحتملة في تاريخ ووقت ساعة نظام الحاسب أثناء تشغيل البرنامج. من خلال هذه الساعة، ستبقى القيمة المُعادة من الدالة `time.Now` صحيحة حتى لو جرى تغيير ساعة نظام الحاسب لاحقًا. مثلًا لو استدعينا الدالة `time.Now` الآن وكان الوقت 10:50، ثم بعد دقيقتين جرى تأخير لساعة الحاسب بمقدار 60 دقيقة، ثم بعد 5 دقائق (من الاستدعاء الأول للدالة `time.Now`) استدعينا الدالة `time.Now` مرةً أخرى (في نفس البرنامج)، سيكون الخرج 10:55 وليس 9:55. لست بحاجة إلى فهم أكثر من ذلك حول آلية عمل هذه الساعة خلال الفصل، لكن إذا كنت ترغب في معرفة المزيد حول الساعات الرتيبة وكيفية استخدامها، لكن يمكنك الذهاب إلى التوثيق الرسمي ورؤية المزيد من التفاصيل لو أحببت.

قد يكون التنسيق الذي يظهر به التاريخ والوقت على شاشة الخرج غير مناسب وقد ترغب بتغييره أو أنه يتضمن أجزاء من التاريخ أو الوقت أكثر مما تريد عرضه. لحسن الحظ، يوفر النوع `time.Time` العديد من الدوال لتنسيق عرض التاريخ والوقت وعرض أجزاء محددة منهما؛ فمثلًا لو أردنا معرفة السنة فقط من المتغير `currentTime` يمكننا استخدام التابع `Year` أو يمكننا عرض الساعة فقط من خلال التابع `Hour`.

لنفتح ملف main.go مرةً أخرى ونضيف التعديلات التالية لرؤية ذلك:

```
...
func main() {
  currentTime := time.Now()
  fmt.Println("The time is", currentTime)
  fmt.Println("The year is", currentTime.Year())
  fmt.Println("The month is", currentTime.Month())
  fmt.Println("The day is", currentTime.Day())
  fmt.Println("The hour is", currentTime.Hour())
  fmt.Println("The minute is", currentTime.Hour())
  fmt.Println("The second is", currentTime.Second())
}
```

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج:

```
The time is 2021-08-15 14:30:45.0000001 -0500 CDT m=+0.000066626
The year is 2021
The month is August
The day is 15
The hour is 14
The minute is 14
The second is 45
```

كما ذكرنا منذ قليل: في كل مرة نُشغّل فيها هذا البرنامج سيكون التاريخ أو الوقت مختلفين، لكن التنسيق يجب أن يكون متشابهًا. طبعنا في هذا المثال التاريخ كاملًا (السطر الأول)، ثم استخدمنا توابع النوع `time.Time` لعرض تفاصيل محددة من التاريخ كلٌ منها بسطر منفرد؛ بحيث عرضنا السنة ثم الشهر ثم اليوم ثم الساعة وأخيرًا الثواني. ربما نلاحظ أن الشهر طُبع اسمه `August` وليس رقمه كما في التاريخ الكامل، وذلك لأن التابع `Month` يعيد الشهر على أنه قيمة من النوع `time.Month` بدلًا من مجرد رقم، ويكون التنسيق عند طباعته سلسلة نصية `string`.

لنحدّث ملف main.go مرةً أخرى ونضع استدعاءات التوابع السابقة كلها ضمن دالة `fmt.Printf` لنتمكن من طباعة التاريخ والوقت الحاليين بتنسيق أقرب إلى ما قد نرغب في عرضه على المستخدم:

```
...
func main() {
  currentTime := time.Now()
  fmt.Println("The time is", currentTime)
  fmt.Printf("%d-%d-%d %d:%d:%d\n",
        currentTime.Year(),
        currentTime.Month(),
        currentTime.Day(),
        currentTime.Hour(),
        currentTime.Hour(),
        currentTime.Second())
}
```

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
The time is 2021-08-15 14:30:45.0000001 -0500 CDT m=+0.000066626
2021-8-15 14:14:45
```

الخرج الآن أقرب بكثير إلى ما نريد، ولكن لا تزال هناك بعض الأشياء الـتي يمكن تعـديلها في الخـرج، فمثلًا عُرض الشهر رقمًا هذه المرة، لأننا استخدمنا العنصر النائب `%d` الذي سيُجبر النوع `time.Month` على اسـتخدام رقم وليس سلسلة نصية. هنا قد نفضل عرض رقمين 08 بدلًا من رقم واحـد 8، وبالتـالي يجب أن نغيّـر تنسـيق `fmt.Printf` وفقًا لذلك، ولكن ماذا لو أردنا أيضًا إظهار الوقت بتنسيق 12 ساعة بـدلًا من 24 سـاعة كمـا هـو موضح في الخرج أعلاه؟ سيتطلب هذا بعض العمليات الحسابية الخاصة ضمن الدالة `fmt.Printf`.

بالرغم من إمكانية طباعة التواريخ والأوقات باستخدام `fmt.Printf`، ولكن يمكن أن يُصبح الأمر مرهقًا إذا أردنا إجراء تنسيقات أعقد، فقد ينتهي بنا الأمر بكتابة عدد كبير من الأسطر البرمجية لكل جزء نريد عرضـه، أو قـد نحتاج إلى إجراء عدد من العمليات الحسابية الخاصة لتحديد ما نريد عرضه.

أنشأنا في هذا القسم برنامجًا يستخدم الدالـة `time.Now` للحصـول على الـوقت الحـالي واسـتخدمنا دوال مثل `Year` و `Hour` على النوع `time.Time` لعرض معلومـات عن التـاريخ والـوقت الحـاليين، كمـا تعرّفنـا على كيفية إجراء تنسيقات بسيطة على آلية عرضهما، مثل عرض أجـزاء محـددة من التـاريخ والـوقت. رأينـا أيضًـا أن عملية الحصول على تنسيقات أعقد تصبح أصعب باستخدام الدالة `fmt.Printf`. لتسهيل الأمر توفر لغـة جـو تـابع خـاص لتنسـيق التـواريخ والأوقـات ويعمـل بطريقـة مشـابهة لعمـل الدالـة `fmt.Printf` الـتي استخدمناها في البداية.

35.3 طباعة وتنسيق تواريخ محددة

إضافة إلى التوابع التي رأيناها في القسم السابق، يوفر النوع `time.Time` تابعًا يُدعى `Format`، يسمح لك بتقـديم نسـق layout على هيئـة سلسـلة نصـية `string` (بطريقـة مشـابهة لتنسـيق السلاسـل في دالـتي `fmt.Printf` و `fmt.Sprintf`). يُخـبر هـذا النسـق التـابع `Format` بالشـكل الـذي نريـده لطباعـة التاريخ والوقت.

نسـتخدم خلال هـذا القسـم نفس الشـيفرة السـابقة، ولكن بطريقـة أكـثر إيجـازًا وسـهولةً من خلال التـابع `Format`. بدايةً ربما يكون من الأفضل معرفة كيف يـؤثر تـابع `Format` على خـرج التـاريخ والـوقت في حـال لم يتغير في كل مرة نُشغّل البرنامج، أي عندما نُثبّت التاريخ والوقت. نحصل حاليًـا على الـوقت الحـالي باسـتخدام `time.Now`، لذلك في كل مرة نُشغّل البرنامج سـيظهر رقم مختلـف. هنـاك دالـةٌ مفيـدة أخـرى توفرهـا الحزمـة

time، وهي الدالة time.Date التي تسمح بتحديد تاريخ ووقت محددين لتمثيلهم مثل قيم من النوع time.Time.

لنبدأ باستخدام الدالة time.Date بدلًا من time.Now في البرنامج. افتح ملف main.go مرةً أخرى واستبدل time.Now بدالة time.Date:

```
...
func main() {
  theTime := time.Date(2021, 8, 15, 14, 30, 45, 100, time.Local)
  fmt.Println("The time is", theTime)
}
```

تتضمن معاملات الدالة time.Date السنة والشهر واليوم والساعة والدقيقة والثواني من التاريخ والوقت اللذين نريد تمثيلهما للنوع time.Time. يمثل المعامل قبل الأخير النانو ثانية، والمعامل الأخير هو المنطقة الزمنية المطلوب إنشاء الوقت لها (نُغطي موضوع المناطق الزمنية لاحقًا). شغّل البرنامج باستخدام go run:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
The time is 2021-08-15 14:30:45.0000001 -0500 CDT
```

سيكون الخرج الذي الذي سنراه في كل مرة نُشغّل فيها البرنامج هو نفسه باستثناء المنطقة الزمنية لأنها تعتمد على المنطقة الزمنية لجهاز الحاسب الذي تُشغّل عليه، وذلك لأننا نستخدم تاريخ ووقت محددين. طبعًا لا يزال تنسيق الخرج مشابهًا لما رأيناه من قبل، لأن البرنامج لا يزال يستخدم التنسيق الافتراضي للنوع time.Time. بعد أن أصبح لدينا تاريخ ووقت قياسي للعمل به -يمكننا استخدامه لتغيير كيفية تنسيق الوقت عند عرضه باستخدام التابع Format.

35.3.1 تنسيق عرض التاريخ والوقت باستخدام تابع Format

تتضمن العديد من لغات البرمجة طريقة مشابهة لتنسيق التواريخ والأوقات التي تُعرض، ولكن الطريقة التي تُنشئ بها لغة جو تصميمًا لتلك التنسيقات قد تكون مختلفةً قليلًا عن باقي لغات البرمجة. يستخدم تنسيق التاريخ في اللغات الأخرى نمطًا مشابهًا لكيفية عمل Printf في لغة جو، إذ يُستخدم محرف % متبوعًا بحرف يمثل جزءًا من التاريخ أو الوقت المطلوب إدراجه. مثلًا قد تُمثّل السنة المكونة من 4 أرقام بواسطة العنصر النائب %Y موضوعًا ضمن السلسلة؛ أما في لغة جو تُمثّل هذه الأجزاء من التاريخ أو الوقت بمحارف تمثل تاريخًا محددًا. مثلًا، لتمثيل سنة معينة من 4 أرقام نكتب 2006 ضمن السلسلة. تتمثل فائدة هذا النوع من التصميم في أن ما نراه في الشيفرة يمثل ما سنراه في الخرج، وبالتالي عندما نكون قادرين على رؤية شكل

الخرج، فإنه يسهل علينا التحقق من أن التنسيق الذي كتبناه يُطابق ما نبحث عنه، كما أنه يسهل على المطوّرين الآخرين فهم خرج البرنامج دون تشغيل البرنامج أصلًا.

يعتمد جو التصميم الافتراضي التالي، لعرض التاريخ والوقت:

```
01/02 03:04:05PM '06 -0700
```

إذا نظرت إلى كل جزء من التاريخ والوقت في هذا التصميم الافتراضي، فسترى أنها تزيد بمقدار واحد لكل جزء. يأتي الشهر أولًا 01 ثم يليه يوم الشهر 02 ثم الساعة 03 ثم الدقيقة 04 ثم الثواني 05 ثم السنة 06 (أي 2006)، وأخيرًا المنطقة الزمنية 07. يسهل هذا الأمر إنشاء تنسيقات التاريخ والوقت مستقبلًا. يمكن أيضًا العثور على أمثلة للخيارات المتاحة للتنسيق في توثيق لغة جو الخاصة بحزمة `time`.

سنستخدم الآن التابع الجديد `Format` لاستبدال وتنظيف تنسيق التاريخ الذي طبعناه في القسم الأخير. يتطلب الأمر -بدون `Format`- عدة أسطر واستدعاءات لدوال من أجل عرض ما نريده، لكن باستخدام هذه الدالة يُصبح الأمر أسهل وأنظف.

لنفتح ملف main.go ونضيف استدعاء جديد للدالة `fmt.Println` ونمرر لها القيمة المُعادة من استدعاء التابع `Format` على المتغير `theTime`:

```
...
func main() {
  theTime := time.Date(2021, 8, 15, 14, 30, 45, 100, time.Local)
  fmt.Println("The time is", theTime)
  fmt.Println(theTime.Format("2006-1-2 15:4:5"))
}
```

إذا نظرنا إلى التصميم المستخدم للتنسيق، فسنرى أنه يستخدم نفس التصميم الافتراضي في تحديد كيفية تنسيق التاريخ (January 2, 2006). شيء واحد يجب ملاحظته هو أن الساعة تستخدم 15 بدلًا من 03. يوضح هذا أننا نرغب في عرض الساعة بتنسيق 24 ساعة بدلًا من تنسيق 12 ساعة.

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
The time is 2021-08-15 14:30:45.0000001 -0500 CDT
2021-8-15 14:30:45
```

نلاحظ أننا حصلنا على خرج مماثل للقسم الأخير لكن بطريقة أبسط. كل ما نحتاج إليه هو سطر واحد من التعليمات البرمجية وسلسلة تمثّل التصميم، ونترك الباقي للدالة `Format`. اعتمادًا على التاريخ أو الوقت الذي نعرضه، من المحتمل أن يكون استخدام تنسيق متغير الطول مثل التنسيق السابق عند طباعة الأرقام مباشرةً صعب القراءة لنا أو للمستخدمين أو لأي شيفرة أخرى تحاول قراءة القيمة. يؤدي استخدام 1 لتنسيق الشهر إلى عرض شهر آذار (مارس) بالرقم 3، بينما يحتاج شهر تشرين الأول (أكتوبر) محرفين ويظهر بالرقم 10.

افتح الملف main.go وأضِف سطرًا إضافيًا إلى البرنامج لتجربة نسق آخر أكثر تنظيمًا. هذه المرة سنضع بادئة 0 قبل أجزاء التاريخ والوقت الفردية، ونجعل الساعة تستخدم تنسيق 12 ساعة.

```
func main() {
  theTime := time.Date(2021, 8, 15, 14, 30, 45, 100, time.Local)
  fmt.Println("The time is", theTime)
  fmt.Println(theTime.Format("2006-1-2 15:4:5"))
  fmt.Println(theTime.Format("2006-01-02 03:04:05 pm"))
}
```

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج:

```
The time is 2021-08-15 14:30:45.0000001 -0500 CDT
2021-8-15 14:30:45
2021-08-15 02:30:45 pm
```

بإضافة بادئة 0 إلى أجزاء سلسلة النسق، يصبح الرقم 8 للشهر في الخرج الجديد 08، ونفس الأمر بالنسبة للساعة التي أصبحت الآن بتنسيق 12 ساعة. لاحظ أن النسق الذي نكتبه يوافق ما نراه في الخرج، وهذا يجعل الأمور أبسط.

في بعض الحالات يكون هناك برامج أخرى تتفاعل مع البرنامج الذي لدينا، وقد يتضمن ذلك التعامل مع تنسيقات وتصميمات التواريخ في برنامجنا، كما أنه قد يكون من المتعب ومن العبء إعادة إنشاء هذه التنسيقات في كل مرة. في هذه الحالة ربما يكون من الأبسط استخدام تصميم تنسيق مُعرّف مُسبقًا ومعروف.

35.3.2 استخدام تنسيقات معرفة مسبقًا

هناك العديد من التنسيقات جاهزة الاستخدام والشائعة للتاريخ، مثل العلامات الزمنية Timestamps لرسائل التسجيل log messages، وفي حال أردت إنشاء هكذا تنسيقات في كل مرة تحتاجها، سيكون أمرًا مملًا ومتعبًا. تتضمن حزمة `time` تنسيقات جاهزة يمكنك استخدامها لجعل الأمور أسهل واختصار الجهد

المكرر. أحد هذه التنسيقات هو RFC 3339، وهو مستند يُستخدم لتحديد كيفية عمل المعايير على الإنترنت، ويمكن بعد ذلك بناء تنسيقات RFC على بعضها. تحدد بعض تنسيقات RFC مثل RFC 2616 كيفية عمل بروتوكول HTTP، وهناك تنسيقات تُبنى على هذا التنسيق لتوسيع تعريف بروتوكول HTTP. لذا في حالة RFC 3339، يحدد RFC تنسيقًا قياسيًا لاستخدامه في الطوابع الزمنية على الإنترنت. التنسيق معروف ومدعوم جيدًا عبر الإنترنت، لذا فإن فرص رؤيته في مكان آخر عالية.

تُمثّل جميع تنسيقات الوقت المُعرّفة مسبقًا في الحزمة الزمنية `time` بسلسلة ثابتة `const string` تُسمى بالتنسيق الذي تمثله. هناك اثنين من التنسيقات المتاحة للتنسيق RFC 3339، هما: `time.RFC3339` و `time.RFC3339Nano`، والفرق بينهما أن التنسيق الثاني يُمثّل الوقت بالنانو ثانية.

لنفتح الآن ملف main.go ونعدّل البرنامج لاستخدام تنسيق `time.RFC3339Nano` للخرج:

```
...
func main() {
  theTime := time.Date(2021, 8, 15, 14, 30, 45, 100, time.Local)
  fmt.Println("The time is", theTime)
  fmt.Println(theTime.Format(time.RFC3339Nano))
}
```

نظرًا لأن التنسيقات المعرّفة مسبقًا هي قيم من النوع `string` للتنسيق المطلوب، نحتاج فقط إلى استبدال التنسيق الذي كُنا تستخدمه عادةً بهذا التنسيق (أو أي تنسيق جاهز آخر نريده). شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
The time is 2021-08-15 14:30:45.0000001 -0500 CDT
2021-08-15T14:30:45.0000001-05:00
```

يُعد تنسيق `RFC 3339` جيدًا للاستخدام إذا كنا بحاجة إلى حفظ قيمة زمنية مثل سلسلة نصية `string` في مكان ما، إذ يمكن قراءتها بسهولة من قبل العديد من لغات البرمجة والتطبيقات الأخرى، كما أنها مختصرة قدر الإمكان.

عدّلنا خلال هذا القسم البرنامج بحيث استخدمنا التابع `Format` لطباعة التاريخ والوقت، إذ يمنحنا هذا النسق المرن إمكانية الحصول على الخرج الأخير. أخيرًا، تعرّفنا واستخدمنا واحدًا من السلاسل النصية المعرفة مسبقًا لطباعة الوقت والتاريخ اعتمادًا على تنسيقات جاهزة ومدعومة يمكن استخدامها مباشرةً دون تكلّف عناء كتابة تصميمات تنسيق يدويًا.

سنعدّل في القسم التالي برنامجنا لتحويل قيمة السلسلة `string` إلى النوع `time.Time` لنتمكن من معالجتها من خلال تحليلها Parsing. يمكنك الاطلاع على مقال تحليل التاريخ والوقت في dot NET على أكاديمية حسوب لمزيدٍ من المعلومات حول مفهوم التحليل.

35.4 تحويل السلاسل النصية إلى قيم زمنية عبر تحليلها

قد نصادف في العديد من التطبيقات تواريخ ممثلة بسلاسل نصية من النوع `string`، وقد نرغب بإجراء بعض التعديلات أو بعض العمليات عليها؛ فمثلًا قد تحتاج إلى استخراج جزء التاريخ من القيمة، أو جزء الوقت، أو كامل القيمة للتعامل معها ..إلخ، إضافةً إلى إمكانية استخدام التابع `Format` لإنشاء قيم `string` من قيم النوع `time.Time`. تتيح لغة جو أيضًا إمكانية التحويل بالعكس، أي من سلسلة إلى `time.Time` من خلال الدالة `time.Parse`، إذ تعمل هذه الدالة بآلية مشابهة للتابع `Format`، بحيث تأخذ نسق التاريخ والوقت المتوقع إضافةً إلى قيمة السلسلة مثل معاملات.

لنفتح ملف main.go ونحدّثه لاستخدام دالة `time.Parse` لتحويل `timeString` إلى متغير `time.Time`:

```
...
func main() {
  timeString := "2021-08-15 02:30:45"
  theTime, err := time.Parse("2006-01-02 03:04:05", timeString)
  if err != nil {
        fmt.Println("Could not parse time:", err)
  }
  fmt.Println("The time is", theTime)
  fmt.Println(theTime.Format(time.RFC3339Nano))
}
```

على عكس التابع `Format`، تُعيد الدالة `time.Parse` قيمة خطأ محتملة في حالة عدم تطابق قيمة السلسلة المُمرّرة مع التنسيق المُمرّر. إذا نظرنا إلى النسق المستخدم، سنرى أن النسق المُعطى لدالة `time.Parse` يستخدم 1 للشهر 2 لليوم من الشهر ..إلخ، وهذا هو نفس النسق المُستخدم في تابع `Format`.

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج كما يلي:

```
The time is 2021-08-15 02:30:45 +0000 UTC
```

```
2021-08-15T02:30:45Z
```

هناك بعض الأشياء التي يجب ملاحظتها في هذا الخرج، أولها هو أن المنطقة الزمنية الناتجة عن تحليل المتغير `timeString` هي المنطقة الزمنية الافتراضية، وهي إزاحة 0+، والتي تُعرف بالتوقيت العالمي المُنسّق Coordinated Universal Time -أو اختصارًا UTC أو توقيت غرينتش، فنظرًا لعدم احتواء القيمة الزمنية أو التصميم على المنطقة الزمنية، لا تعرف الدالة `time.Parse` المنطقة الزمنية التي تريد ربطها بها، وبالتالي تعدّه غرينتش. إذا كنت بحاجة لتحديد المنطقة الزمنية، يمكنك استخدام الدالة `time.ParseInLocation` لذلك. يمكن ملاحظة استخدام نسق `time.RFC3339Nano`، لكن الخرج لا يتضمن قيم بالنانو ثانية، وسبب ذلك هو أن القيم التي تحللها دالة `time.Parse` ليست نانو ثانية، وبالتالي تُضبط القيمة لتكون 0 افتراضيًا، وعندما تكون النانو ثانية 0، لن يتضمّن استخدام التنسيق `time.RFC3339Nano` قيمًا بالنانو ثانية في الخرج. يمكن للتابع `time.Parse` أيضًا استخدام أيًّا من تنسيقات الوقت المعرفة مسبقًا والمتوفرة في حزمة الوقت `time` عند تحليل قيمة سلسلة.

لنفتح ملف main.go ونعدّل قيمة `timeString`، يحبث نحل مشكلة عدم ظهور قيم النانو ثانية عند استخدام تنسيق `time.RFC3339Nano` ولنحدّث معاملات `time.Parse` بطريقة توافق التعديلات الجديدة:

```
...
func main() {
  timeString := "2021-08-15T14:30:45.0000001-05:00"
  theTime, err := time.Parse(time.RFC3339Nano, timeString)
  if err != nil {
        fmt.Println("Could not parse time:", err)
  }
  fmt.Println("The time is", theTime)
  fmt.Println(theTime.Format(time.RFC3339Nano))
}
```

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
The time is 2021-08-15 14:30:45.0000001 -0500 CDT
2021-08-15T14:30:45.0000001-05:00
```

يُظهر خرج `Format` هذه المرة بأن `time.Parse` كانت قادرة على تحليل كلٍ من المنطقة الزمنية وقيم النانو ثانية من متغير `timeString`.

تعلمنا في هذا القسم كيفية استخدام دالة `time.Parse` لتحليل سلسلة `string` تُمثّل تاريخًا معينًا وفقًا لتصميم معين، وتعلمنا كيفية تحديد المنطقة الزمنية للسلسلة المُحللة. سنتعرّف في القسم التالي على كيفية التعامل مع المناطق الزمنية بطريقة أوسع وكيفية التبديل بين المناطق الزمنية المختلفة من خلال الميزات التي يوفرها النوع `time.Time`.

35.5 التعامل مع المناطق الزمنية

يُشاع في التطبيقات التي يستخدمها مُستخدمين من أنحاء مختلفة من العالم -تخزّين التواريخ والأوقات باستخدام توقيت غرينيتش UTC، ثم التحويل إلى التوقيت المحلي للمستخدم عند الحاجة. يسمح ذلك بتخزين البيانات بتنسيق موحّد ويجعل الحسابات بينها أسهل، نظرًا لأن التحويل مطلوب فقط عند عرض التاريخ والوقت للمستخدم.

أنشأنا خلال الفصل برنامجًا يعمل وفقًا للمنطقة الزمنية المحلية المتواجدين بها. لحفظ قيم البيانات من النوع `time.Time` على أنها قيم UTC، نحتاج أولًا إلى تحويلها إلى قيم UTC باستخدام التابع `UTC` الذي يُعيد القيمة الزمنية الموافقة لنظام UTC من متغير التاريخ الذي يستدعيها.

> يجري التحويل هنا من المنطقة المحلية التي يوجد بها الحاسب الذي نستخدمه إلى UTC، وبالتالي إذا كان حاسبنا موجود ضمن منطقة UTC لن نرى فرقًا.

لنفتح ملف main.go ونُطبق هذا الكلام:

```
...
func main() {
  theTime := time.Date(2021, 8, 15, 14, 30, 45, 100, time.Local)
  fmt.Println("The time is", theTime)
  fmt.Println(theTime.Format(time.RFC3339Nano))
  utcTime := theTime.UTC()
  fmt.Println("The UTC time is", utcTime)
  fmt.Println(utcTime.Format(time.RFC3339Nano))
}
```

هذه المرة يُنشئ البرنامج متغير `theTime` على أنه قيمة من النوع `time.Time` وفقًا لمنطقتنا الزمنية، ثم يطبعه بتنسيقين مختلفين، ثم يستخدم تابع `UTC` للتحويل من المنطقة الزمنية المحلية الحالية إلى المنطقة الزمنية UTC.

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
The time is 2021-08-15 14:30:45.0000001 -0500 CDT
2021-08-15T14:30:45.0000001-05:00
The UTC time is 2021-08-15 19:30:45.0000001 +0000 UTC
2021-08-15T19:30:45.0000001Z
```

سيختلف الخرج اعتمادًا على المنطقة الزمنية المحلية، ولكن سنرى في الخرج السابق أن المنطقة الزمنية في المرة الأولى كانت بتوقيت CDT (التوقيت الصيفي المركزي لأمريكا الشمالية)، وهي "-5" ساعات من UTC. بعد استدعاء تابع `UTC` وطباعة الوقت وفق نظام UTC، سنرى أن الوقت تغير من 14 إلى 19، لأن تحويل الوقت إلى UTC أضاف خمس ساعات. من الممكن أيضًا تحويل UTC إلى التوقيت المحلي باستخدام التابع `Local` على متغير النوع `time.Time` بنفس الأسلوب.

افتح ملف main.go مرةً أخرى، وأضِف استدعاءً للتابع `Local` على المتغير `utcTime` لتحويله مرةً أخرى إلى المنطقة الزمنية المحلية:

```
...
func main() {
  theTime := time.Date(2021, 8, 15, 14, 30, 45, 100, time.Local)
  fmt.Println("The time is", theTime)
  fmt.Println(theTime.Format(time.RFC3339Nano))
  utcTime := theTime.UTC()
  fmt.Println("The UTC time is", utcTime)
  fmt.Println(utcTime.Format(time.RFC3339Nano))
  localTime := utcTime.Local()
  fmt.Println("The Local time is", localTime)
  fmt.Println(localTime.Format(time.RFC3339Nano))
}
```

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

ليكون الخرج:

```
The time is 2021-08-15 14:30:45.0000001 -0500 CDT
2021-08-15T14:30:45.0000001-05:00
The UTC time is 2021-08-15 19:30:45.0000001 +0000 UTC
```

```
2021-08-15T19:30:45.0000001Z
The Local time is 2021-08-15 14:30:45.0000001 -0500 CDT
2021-08-15T14:30:45.0000001-05:00
```

نلاحظ أنه جرى التحويل من UTC إلى المنطقة الزمنية المحلية، وهذا يعني طـرح خمس سـاعات من UTC، وتغيير الساعة من 19 إلى 14.

عدّلنا خلال هذا القسم البرنامج لتحويل البيانات الزمنية من منطقة زمنية محلية إلى توقيت غـرينتش UTC باستخدام التابع `UTC`، كما أجرينا تحويلًا معاكسًا باستخدام التابع `Local`.

تتمثل إحدى الميزات الإضافية التي تقدمها حزمة `time` والـتي يمكن أن تكـون مفيـدة في تطبيقاتـك -في تحديد ما إذا كان وقت معين قبل أو بعد وقت آخر.

35.6 مقارنة الأوقات الزمنية

قد تكون مقارنة تاريخين مع بعضهما صعبة أحيانًـا، بسـبب المتغـيرات الـتي يجب أخـذها بالحسـبان عنـد المقارنة، مثل الحاجة إلى الانتباه إلى المناطق الزمنية أو حقيقة أن الأشهر لها عدد مختلف من الأيام عن بعضها. توفر الحزمة الزمنية `time` تابعين لتسهيل ذلك، هما: `Before` و `After` اللذان يمكن تطبيقهما على متغـيرات النوع `time.Time`. تقبل هذه التوابع قيمةً زمنيةً واحدة وتعيدان إمـا `true` أو `false` اعتمـادًا على مـا إذا كـان الوقت المُمثّل بالمتغير الذي يستدعيهما قبل أو بعد الوقت المقدم.

لنفتح ملف main.go ونرى كيف تجري الأمور:

```
...
func main() {
  firstTime := time.Date(2021, 8, 15, 14, 30, 45, 100, time.UTC)
  fmt.Println("The first time is", firstTime)
  secondTime := time.Date(2021, 12, 25, 16, 40, 55, 200, time.UTC)
  fmt.Println("The second time is", secondTime)
  fmt.Println("First time before second?",
firstTime.Before(secondTime))
  fmt.Println("First time after second?", firstTime.After(secondTime))
  fmt.Println("Second time before first?",
secondTime.Before(firstTime))
  fmt.Println("Second time after first?", secondTime.After(firstTime))
}
```

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
The first time is 2021-08-15 14:30:45.0000001 +0000 UTC
The second time is 2021-12-25 16:40:55.0000002 +0000 UTC
First time before second? true
First time after second? false
Second time before first? false
Second time after first? true
```

بما أننا نستخدم في الشيفرة أعلاه نظام الوقت UTC، فيجب أن يكون الخرج متشابهًا بغض النظر عن المنطقة الزمنية. نلاحظ أنه عند استخدام تابع `Before` مع المتغير `firstTime` وتمرير الوقت `secondTime` الذي نريد المقارنة به، ستكون النتيجة `true` أي أن 15-08-2021 قبل 25-12-2021. عند استخدام `After` مع المتغير `firstTime` وتمرير `secondTime`، تكون النتيجة `false` لأن 15-08-2021 ليس بعد 25-12-2021. يؤدي تغيير ترتيب استدعاء التوابع على `secondTime` إلى إظهار نتائج معاكسة.

هناك طريقة أخرى لمقارنة القيم الزمنية في حزمة الوقت وهي تابع `Sub`، الذي يطرح تاريخًا من تاريخ آخر ويُعيد قيمةً من نوع جديد هو `time.Duration`. على عكس قيم النوع `time.Time` التي تمثل نقاط زمنية مطلقة، تمثل قيمة `time.Duration` فرقًا في الوقت. مثلًا، قد تعني عبارة "في ساعة واحدة" مدة duration لأنها تعني شيئًا مختلفًا بناءً على الوقت الحالي من اليوم، لكنها تمثّل "عند الظهر" وقتًا محددًا ومطلقًا. تستخدم لغة جو النوع `time.Duration` في بعض الحالات، مثل الوقت الذي نريد فيه تحديد المدة التي يجب أن تنتظرها الدالة قبل إعادة خطأ أو كما هو الحال هنا، إذ نحتاج إلى معرفة مقدار الزمن بين وقت وآخر.

لنفتح الآن ملف main.go ونستخدم التابع `Sub` على المتغير `firstTime` و `secondTime` ونطبع النتائج:

```
...
func main() {
  firstTime := time.Date(2021, 8, 15, 14, 30, 45, 100, time.UTC)
  fmt.Println("The first time is", firstTime)
  secondTime := time.Date(2021, 12, 25, 16, 40, 55, 200, time.UTC)
  fmt.Println("The second time is", secondTime)
  fmt.Println("Duration between first and second time is",
firstTime.Sub(secondTime))
  fmt.Println("Duration between second and first time is",
secondTime.Sub(firstTime))
```

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
The first time is 2021-08-15 14:30:45.0000001 +0000 UTC
The second time is 2021-12-25 16:40:55.0000002 +0000 UTC
Duration between first and second time is -3170h10m10.0000001s
Duration between second and first time is 3170h10m10.0000001s
```

يوضح الناتج أعلاه أن هناك 3170 ساعة و 10 دقائق و 10 ثوان و 100 نانو ثانية بين التاريخين، وهناك بعض الأشياء التي يجب ملاحظتها في الخرج، أولها أن المدة بين المرة الأولى `first time` والثانية `second time` قيمة سالبة، وهذا يشير إلى أن الوقت الثاني بعد الأول، وسيكون الأمر مماثلًا إذا طرحنا 5 من 0 وحصلنا على 5-. نلاحظ أيضًا أن أكبر وحدة قياس للمدة هي ساعة، لذلك فهي لا تُقسم إلى أيام أو شهور. بما أن عدد الأيام في الشهر غير متسق وقد يكون لكلمة "اليوم" معنى مختلف عند التبديل للتوقيت الصيفي، فإن قياس الساعة هو أدق مقياس، إذ أنه لا يتقلب.

عدّلنا البرنامج خلال هذا القسم مرتين للمقارنة بين الأوقات الزمنية باستخدام ثلاث توابع مختلفة؛ إذ استخدمنا أولًا التابعين `Before` و `After` لتحديد ما إذا كان الوقت قبل أو بعد وقت آخر؛ ثم استخدمنا `Sub` لمعرفة المدة بين وقتين.

ليس الحصول على المدة بين وقتين الطريقة الوحيدة التي تستخدم بها الحزمة الزمنية الدالة `time.Duration`، إذ يمكننا أيضًا استخدامها لإضافة وقت أو إزالته من قيمة من النوع `time.Time`.

35.7 إضافة وطرح الأوقات الزمنية

إحدى العمليات الشائعة عند كتابة التطبيقات التي تستخدم التواريخ والأوقات هي تحديد وقت الماضي أو المستقبل بناءً على وقت آخر مرجعي، فمثلًا يمكن استخدام هذا الوقت المرجعي لتحديد موعد تجديد الاشتراك بخدمة، أو ما إذا كان قد مر قدرٌ معينٌ من الوقت منذ آخر مرة جرى فيها التحقق من أمر معين. توفر حزمة الوقت `time` طريقةً لمعالجة الأمر، وذلك من خلال تحديد المدد الزمنية الخاصة بنا باستخدام متغيرات من النوع `time.Duration`.

إنشاء قيمة من النوع `time.Duration` بسيط جدًا والعمليات الرياضية عليه متاحة كما لو أنه متغير عددي عادي؛ فمثلًا لإنشاء مدة زمنية `time.Duration` تُمثّل ساعة أو ساعتين أو ثلاثة ..إلخ، يمكننا استخدام `time.Hour` مضروبةً بعدد الساعات التي نريدها:

```
oneHour := 1 * time.Hour
twoHours := 2 * time.Hour
```

```
tenHours := 10 * time.Hour
```

نستخدم `time.Minute` و `time.Second` في حال الدقائق والثواني:

```
tenMinutes := 10 * time.Minute
fiveSeconds := 5 * time.Second
```

يمكن أيضًا إضافة مدة زمنية إلى مدة أخرى للحصول على مجموعهما. لنفتح ملف `main.go` ونطبق ذلك:

```
...
func main() {
  toAdd := 1 * time.Hour
  fmt.Println("1:", toAdd)
  toAdd += 1 * time.Minute
  fmt.Println("2:", toAdd)
  toAdd += 1 * time.Second
  fmt.Println("3:", toAdd)
}
```

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
1: 1h0m0s
2: 1h1m0s
3: 1h1m1s
```

نلاحظ من الخرج أن المدة الأولى المطبوعة هي ساعة واحدة، وهذا يتوافق مع `1 * time.Hour` في الشيفرة. أضفنا بعد ذلك `1 * time.Minute` إلى القيمة `toAdd` أي ساعة ودقيقة واحدة. أخيرًا أضفنا `1 * time.Second` إلى قيمة `toAdd`، لينتج لدينا ساعة واحدة ودقيقة واحدة وثانية في المدة الزمنية الممثلة بالنوع `time.Duration`.

يمكن أيضًا دمج عمليات إضافة المُدد معًا في عبارة واحدة، أو طرح مدة من أخرى:

```
oneHourOneMinute := 1 * time.Hour + 1 * time.Minute
tenMinutes := 1 * time.Hour - 50 * time.Minute
```

لنفتح ملف main.go ونعدله بحيث نستخدم توليفة من هذه العمليات لطرح دقيقـة واحـدة وثانيـة واحـدة من toAdd:

```
...
func main() {
  ...
  toAdd += 1 * time.Second
  fmt.Println("3:", toAdd)
  toAdd -= 1*time.Minute + 1*time.Second
  fmt.Println("4:", toAdd)
}
```

شغّل البرنامج باستخدام go run:

```
$ go run main.go
```

سيعطي الخرج التالي:

```
1: 1h0m0s
2: 1h1m0s
3: 1h1m1s
4: 1h0m0s
```

يظهر السطر الرابع من الخرج، والذي يُمثّل ناتج طرح المجموع 1*time.Minute + 1*time.Second من toAdd أن العملية ناجحة، إذ حصلنا على قيمة ساعة واحدة (طُرحت الثانية والدقيقة).

يتيح لنا استخدام هذه المُدد المقترنة بالتابع Add للنـوع time.Time حسـاب المـدد الزمنيـة بين نقطـتين زمنيتين إحداهما نقطة مرجعية، مثل حساب المدة منذ أول يوم اشتراك في خدمة معينة حتى اللحظة.

لرؤية مثال آخر نفتح ملف main.go ونجعل قيمة toAdd تساوي 24 سـاعة أي 24 * time.Hour، ثم نستخدم التابع Add على قيمة متغير من النوع time.Time لمعرفة الوقت الذي سيكون بعد 24 ساعة من تلك النقطة كما يوضح الكود التالي:

```
...
func main() {
  theTime := time.Date(2021, 8, 15, 14, 30, 45, 100, time.UTC)
  fmt.Println("The time is", theTime)
  toAdd := 24 * time.Hour
  fmt.Println("Adding", toAdd)
```

```
  newTime := theTime.Add(toAdd)
  fmt.Println("The new time is", newTime)
}
```

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
The time is 2021-08-15 14:30:45.0000001 +0000 UTC
Adding 24h0m0s
The new time is 2021-08-16 14:30:45.0000001 +0000 UTC
```

نلاحظ أن إضافة 24 ساعة إلى التاريخ 2021-08-15 سينتج عنه التاريخ الجديد 2021-08-16. يمكننا أيضًا استخدام التابع `Add` لطرح الوقت، إذ سنستخدم قيمة سالبة ببساطة، فيصبح الأمر كما لو أننا نستخدم التابع `Sub`. لنفتح ملف main.go ونطبق هذا الكلام، fحيث سنطرح هذه المرة 24 ساعة، أي يجب أن نستخدم القيمة 24-.

```
...
func main() {
  theTime := time.Date(2021, 8, 15, 14, 30, 45, 100, time.UTC)
  fmt.Println("The time is", theTime)
  toAdd := -24 * time.Hour
  fmt.Println("Adding", toAdd)
  newTime := theTime.Add(toAdd)
  fmt.Println("The new time is", newTime)
}
```

شغّل البرنامج باستخدام `go run`:

```
$ go run main.go
```

سيكون الخرج كما يلي:

```
The time is 2021-08-15 14:30:45.0000001 +0000 UTC
Adding -24h0m0s
The new time is 2021-08-14 14:30:45.0000001 +0000 UTC
```

نلاحظ من الخرج أنه قد طُرح 24 ساعة من الوقت الأصلي.

استخدمنا خلال هذا القسم التوابع time.Hour و time.Minute و time.Second لإنشاء قيم من النوع time.Duration.كما استخدمنا أيضًا قيم النوع time.Duration مع التابع Add للحصول على قيمة جديدة لمتغير من النوع time.Time. وسيكون لدينا القدرة الكافية على التعامل مع التاريخ والوقت في التطبيقات من خلال التوابع time.Now و Add و Before و After.

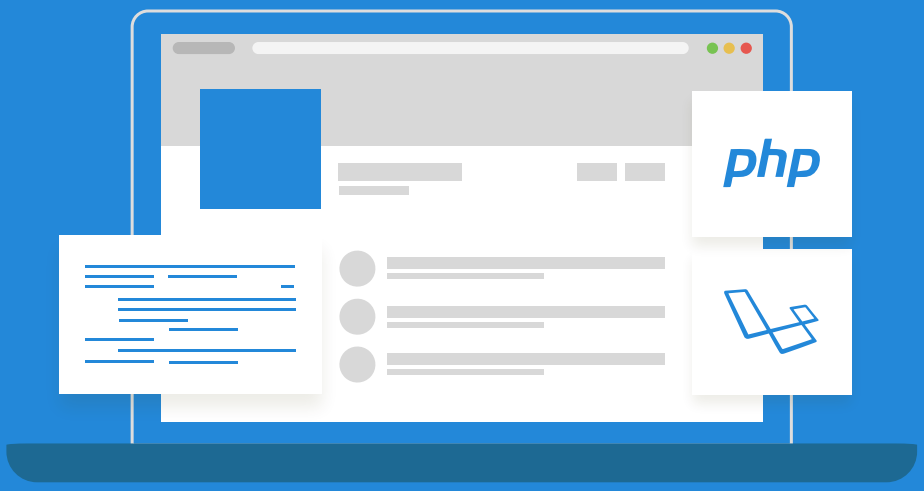
35.8 الخاتمة

استخدمنا خلال هذا الفصل الدالة time.Now للحصول على التاريخ والوقت المحلي الحالي على شكل قيم من النوع time.Time، ثم استخدمنا التوابع Year و Month و Hour ...إلخ. للحصول على معلومات محددة من التاريخ والوقت.

استخدمنا بعد ذلك التابع Format لطباعة الوقت بالطريقة التي نريدها وفقًا لتنسيق مخصص نقدمه أو وفقًا لتنسيق جاهز مُعرّف مسبقًا. استخدمنا الدالة time.Parse لتحويل قيمة سلسلة نصية string تمثّل بيانات زمنية إلى قيمة من النوع time.Time لنتمكن من التعامل معها.

تعلمنا أيضًا كيفية التبديل من المنطقة الزمنية المحلية إلى منطقة غرينتش UTC باستخدام التابع Local والعكس. تعلمنا استخدام توابع Sub و Add لإجراء عمليات جمع وطرح على البيانات الزمنية، لإيجاد الفرق بين مدتين زمنيتين أو لإضافة مدة زمنية إلى بيانات زمنية محددة.

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

36. استخدام السياقات Contexts

عند تطوير التطبيقات الكبيرة، وخصوصًا برمجيات الخادم - يكون من المفيد أحيانًا لدالةٍ ما معرفة بعض المعلومات عن البيئة التي تُنفّذ بها إلى جانب المعلومات اللازمة لعمل الدالة نفسها. لنأخذ مثلًا دالة خادم ويب تتعامل مع طلب HTTP لعميل معين، هنا قد تحتاج الدالة إلى معرفة عنوان URL الذي يطلبه العميل فقط لتحقيق الاستجابة، وفي هذه الحالة ربما تحتاج فقط إلى تمرير العنوان مثل معامل إلى الدالة. المشكلة أن هناك بعض الأشياء المفاجئة التي يمكن أن تحدث مثل انقطاع الاتصال مع العميل قبل تحقيق الاستجابة وتلقيه الرد. بالتالي، إذا كانت الدالة التي تؤدي الاستجابة لا تعرف أن العميل غير متصل، لن يصل الرد والعمليات التي يجريها الخادم ستكون مجرد هدر للموارد الحاسوبية على استجابة لن تُستخدم. لتفادي هكذا حالات يجب أن يكون بمقدور الخادم معرفة سياق الطلب (مثل حالة اتصال العميل)، وبالتالي إمكانية إيقاف معالجة الطلب بمجرد انقطاع الاتصال مع العميل. هذا من شأنه الحفاظ على الموارد الحاسوبية ويحد من الهدر ويتيح للخادم التحرر أكثر من الضغط وتقديم أداء أفضل. تظهر فائدة هذا النوع من المعلومات أكثر في الحالات التي تتطلّب فيها الدوال وقتًا طويلًا نسبيًا في التنفيذ، مثل إجراء استدعاءات قاعدة البيانات. لمعالجة هذه القضايا ومنح إمكانية الوصول الكامل لمثل هذه المعلومات تُقدم لغة جو حزمة السياق `context` في المكتبة القياسية.

سنُنشئ خلال هذا الفصل برنامجًا يستخدم سياقًا داخل دالة. سنعدّل بعدها البرنامج لتخزين بيانات إضافية في السياق واستردادها من دالة أخرى. بعد ذلك سنستفيد من فكرة السياق لإرسال إشارة للدالة التي تُجري عملية المعالجة، لتوقف تنفيذ أي عمليات معالجة مُتبقية.

36.1 المتطلبات

- إصدار مثبت من لغة جو، ارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبت لغة جو Go وجهز بيئة تطوير محلية بحسب نظام تشغيلك.

- يُفضّل أن تكون على دراية بكيفية إنشاء الوحدات في لغو جو. يمكنك مراجعة فصل استخدام الوحدات Modules في لغة جو Go.
- (اختياري) قراءة فصل معالجة الأخطاء في لغة جو Go قد يكون مفيدًا للحصول على شرح أكثر تعمقًا لمعالجة الأخطاء ومعرفة طريقة معالجة حالات الانهيار في لغة جو Go. فنحن عمومًا نغطي بعض الموضوعات منها في هذا الفصل، لكن بشرح عالي المستوى.
- فهم لتنظيمات جو goroutines والقنوات channels. يمكنك الاطلاع على فصل تشغيل عدة دوال على التساير في لغة جو Go.
- معرفة بكيفية التعامل مع التاريخ والوقت في لغة جو. يمكنك الاطلاع على فصل التعامل مع التاريخ والوقت في لغة جو Go.
- معرفة كيفية التعامل مع تعليمة `switch` في لغة جو. يمكنك الاطلاع على فصل التعامل مع التعليمة Switch في لغة جو Go.

36.2 إنشاء سياق context

تستخدم العديد من الدوال في لغة جو حزمة `context` لجمع معلومات إضافية حول البيئة التي تُنفّذ فيها، وعادةً ما يُقدّم هذا السياق للدوال التي تستدعيها أيضًا. ستتمكن البرامج من خلال واجهة `context.Context` التي توفرها حزمة السياق، وتمريرها من من دالة إلى أخرى، من نقل معلومات السياق من أعلى نقطة تنفيذ في البرنامج (دالة `main`) إلى أعمق نقطة تنفيذ في البرنامج (دالة ضمن دالة أخرى أو ربما أعمق). مثلًا، تُقدّم الدالة Context من النوع `http.Request` سياقًا `context.Context` يتضمن معلومات عن العميل الذي أرسل الطلب، ويُحذف أو ينتهي هذا السياق في حالة قُطع الاتصال مع العميل، حتى لو لم يكن الطلب قد انتهت معالجته. بالتالي، إذا كانت هناك دالة ما تستدعي الدالة QueryContext التابعة إلى `sql.DB`، وكان قد مُرر لهذه الدالة قيمة `context.Context`، وقُطع الاتصال مع العميل، سيتوقف تنفيذ الاستعلام مباشرةً في حالة لم يكن قد انتهى من التنفيذ.

سننشئ في هذا القسم برنامجًا يتضمن دالة تتلقى سياقًا مثل معامل، ونستدعي هذه الدالة باستخدام سياق فارغ نُنشئه باستخدام الدالتين `context.TODO` و `Context.Background`. كما هو معتاد، سنحتاج لبدء إنشاء برامجنا إلى إنشاء مجلد للعمل ووضع الملفات فيه، ويمكن وضع المجلد في أي مكان على الحاسب، إذ يكون للعديد من المبرمجين عادةً مجلدٌ يضعون داخله كافة مشاريعهم. سنستخدم في هذا الفصل مجلدًا باسم `projects`، لذا فلننشئ هذا المجلد وننتقل إليه:

```
$ mkdir projects
$ cd projects
```

الآن من داخل هذا المجلد، سنشغّل الأمر `mkdir` لإنشاء مجلد `contexts` ثم سنستخدم `cd` للانتقال إليه:

```
$ mkdir contexts
$ cd contexts
```

يمكننا الآن فتح ملف main.go باستخدام محرر نانو nano أو أي محرر آخر تريده:

```
$ nano main.go
```

سنُنشئ الآن دالة `doSomething` داخل ملف main.go. تقبل هذه الدالة مثل `context.Context` معامل، ثم نضيف دالة `main` التي تُنشئ سياقًا وتستدعي `doSomething` باستخدام ذلك السياق.

نضيف ما يلي داخل ملف `main.go`:

```
package main
import (
  "context"
  "fmt"
)
func doSomething(ctx context.Context) {
  fmt.Println("Doing something!")
}
func main() {
  ctx := context.TODO()
  doSomething(ctx)
}
```

استخدمنا الدالة `context.TODO` داخل الدالة `main`، لإنشاء سياق فارغ ابتدائي. يمكننا استخدام السياق الفارغ مثل موضع مؤقت placeholder عندما لا نكون متأكدين من السياق الذي يجب استخدامه. لدينا أيضًا الدالة `doSomething` التي تقبل معاملًا وحيدًا هو `context.Context` -له الاسم `ctx`- وهو الاسم الشائع له، ويُفضل أن يكون أول معامل في الدالة في حال كان هناك معاملات أخرى، لكن الدالة لا تستخدمه الآن فعليًّا.

لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
Doing something!
```

نلاحظ أن الخرج الذي أظهرته الدالة `fmt.Println` هـو `Doing something!` نتيجـةً لاسـتدعاء الدالـة `doSomething`. لنعدّل البرنامج الآن ونستخدم الدالة `context.Background` التي تُنشئ سياقًا فارغًا:

```
...
func main() {
  ctx := context.Background()
  doSomething(ctx)
}
```

تنشئ الدالة `context.Background` سياقًا فارغًا مثل `context.TODO`، ومن حيث المبـدأ تـؤدي كلتـا الدالتين السابقتين نفس الغرض. الفرق الوحيد هو أن `context.TODO` تُستخدم عندما تُريد أن تُخبر المطورين الآخرين أن هذا السياق هو مجرد سياق مبدأي وغالبًا يجب تعديلـه، أمـا `context.Background` فتُسـتخدم عندما لا نحتاج إلى هكذا إشارة إلى المطورين الآخرين، أي نحتاج ببساطة إلى سـياق فـارغ لا أكـثر ولا أقـل، وفي حال لم تكن متأكدًا أيهما تستخدم، ستكون الدالة `context.Background` خيارًا افتراضيًا.

لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج:

```
Doing something!
```

سيكون الخرج طبعًا نفسـه كمـا في المـرة السـابقة، وبـذلك نكـون قـد تعلمنـا كيفيـة إنشـاء سـياق فـارغ بطريقتين مختلفتين.

لا يفيدنا السياق الفارغ تمامًا إذا بقي على هذا النحو، فعلى الرغم من قدرتنا على تمريره بين الـدوال، إلا أنـه لا يوفر أية معلومات. لجعل السياق مفيدًا نُضيف له بيانات يمكن للدوال الأخرى استردادها والاطلاع عليها.

36.3 إضافة معلومات إلى السياق

إحدى فوائد استخدام `context.Context` في برنامج مـا هي القـدرة على الوصـول إلى البيانـات المخزنـة داخل سياق ما، إذ يمكن لكل طبقـة من البرنـامج إضـافة معلومـات إضـافية حـول مـا يحـدث من خلال إضـافة البيانات إلى سياق وتمرير السياق من دالة إلى أخرى. مثلًا، قد تضيف الدالة الأولى اسم مسـتخدم إلى السـياق، والدالة التالية مسار الملف إلى المحتوى الذي يحاول المستخدم الوصـول إليـه، والدالـة الثالثـة تقـرأ الملـف من قرص النظام وتسجل ما إذا كان قد نجح تحميله أم لا، إضافةً إلى المستخدم الذي حاول تحميله.

يمكن استخدم الدالة `Context.WithValue` من حزمة السياق لإضافة قيمة جديدة إلى السياق. تقبل الدالة ثلاث معاملات: السياق الأب (الأصلي) `context.Context` والمفتاح والقيمة. السياق الأب هو السياق الذي يجب إضافة القيمة إليه مع الاحتفاظ بجميع المعلومات الأخرى المتعلقة بالسياق الأصلي. يُستخدم المفتاح لاسترداد القيمة من السياق. يمكن أن يكون المفتاح والقيمة من أي نوع بيانات، وفي هذا الفصل سيكونان من نوع سلسلة نصية `string`.

تعيد الدالة `Context.WithValue` قيمةً من النوع `context.Context` تتضمن السياق الأب مع المعلومات المُضافة. يمكن الحصول على القيمة التي يُخزنها السياق `context.Context` من خلال استخدام التابع `Value` مع المفتاح.

لنفتح الآن ملف main.go ولنضِف قيمة إلى السياق باستخدام الدالة السابقة، ثم نحدِّث دالة `doSomething`، بحيث تطبع تلك القيمة باستخدام دالة `fmt.Printf`:

```
...
func doSomething(ctx context.Context) {
  fmt.Printf("doSomething: myKey's value is %s\n", ctx.Value("myKey"))
}
func main() {
  ctx := context.Background()
  ctx = context.WithValue(ctx, "myKey", "myValue")
  doSomething(ctx)
}
```

أسندنا في الشيفرة السابقة سياقًا جديدًا إلى المتغير `ctx`، الذي يُستخدم للاحتفاظ بالسياق الأب، ويُعد هذا نمط شائع الاستخدام في حال لم يكن هناك سبب للإشارة إلى سياق أب محدد. إذا كنا بحاجة إلى الوصول إلى السياق الأب في وقت ما، يمكننا إسناد القيمة إلى متغير جديد، كما سنرى قريبًا.

لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
doSomething: myKey's value is myValue
```

يمكننا رؤية القيمة `myValue` التي خزّناها في السياق ضمن الخرج السابق، وذلك نتيجةً لاستدعاء الدالة `doSomething` ضمن الدالة `main`. طبعًا القيمة `myValue` المُستخدمة هنا هي فقط من أجل التوضيح، فمثلًا لو كنا نعمل على خادم كان من الممكن أن تكون هذه القيمة شيئًا آخر مثل وقت بدء تشغيل البرنامج.

عند استخدام السـياقات من المهم معرفـة أن القيم المخزنـة في سـياقٍ `context.Context` مـا تكـون ثابتـة immutable. بالتالي عندما اسـتدعينا الدالـة `context.WithValue` ومررنـا لهـا السـياق الأب، حصـلنا على سياق جديد وليس السياق الأب أو نسخة منه، وإنما حصلنا على سياق جديد يضـم المعلومـات الجديـدة إضـافةً إلى سياق الأب مُغلّفًا wrapped ضمنه.

لنفتح الآن ملف main.go لإضافة دالـة جديـدة `doAnother` تقبـل سـياقًا `context.Context` وتطبـع قيمة السياق من خلال المفتاح، ونعدّل أيضًا الدالة `doSomething`، بحيث نُنشئ داخلهـا سـياقًا جديـدًا يُغلّـف السياق الأب ويضيف معلومات جديدة ولتكن `anotherValue`، ثم نستدعي الدالة `doAnother` على السياق `anotherCtx` الناتج، ونطبع في السطر الأخير من الدالة قيمة السياق الأب.

```
...
func doSomething(ctx context.Context) {
  fmt.Printf("doSomething: myKey's value is %s\n", ctx.Value("myKey"))
  anotherCtx := context.WithValue(ctx, "myKey", "anotherValue")
  doAnother(anotherCtx)
  fmt.Printf("doSomething: myKey's value is %s\n", ctx.Value("myKey"))
}
func doAnother(ctx context.Context) {
  fmt.Printf("doAnother: myKey's value is %s\n", ctx.Value("myKey"))
}
...
```

لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج:

```
doSomething: myKey's value is myValue
doAnother: myKey's value is anotherValue
doSomething: myKey's value is myValue
```

نلاحظ في الخرج سطرين من الدالة `doSomething` وسطر من الدالـة `doAnother`. لم نغـير شـيئًا داخـل الدالة `main`؛ إذ أنشأنا سياقًا فارغًا وغلّفناه مـع القيمـة `myValue` والمفتـاح `myKey` ومرّرنـا السـياق النـاتج إلى `doSomething`، ويمكننا أن نلاحظ أن هذه القيمة مطبوعة على أول سطر من الخرج.

يظهر السطر الثاني من الخرج أنه عند استخدام `context.WithValue` داخل `doSomething` لتغليف السياق الأب `ctx` وتوليد السياق `anotherCtx` بالقيمة `anotherValue` والمفتاح `myKey` (المفتاح نفسه للأب لم يتغير) وتمرير هذا السياق الناتج إلى `doAnother`، فإن القيمة الجديدة تتخطى القيمة الابتدائية.

بالنسبة للسطر الأخير، نلاحظ أنه يطبع القيمة المرتبطة بالمفتاح `myKey` والمرتبطة بالسياق الأب وهي `myValue`. نظرًا لأن الدالة `context.WithValue` تغلّف السياق الأب فقط، سيبقى السياق الأب محتفظًا بنفس القيم الأصلية نفسها. عندما نستدعي التابع `Value` على سياق ما، فإنه يُعيد القيمة المرتبطة بالمفتاح من المستوى الحالي للسياق. عند استدعاء `anotherCtx.Value` من أجل مفتاح `myKey`، سيعيد القيمة `anotherValue` لأنها القيمة المغلّفة للسياق، وبالتالي يتجاوز أي قيم أخرى مُغلّفة للمفتاح، وعند استدعاء `anotherCtx` داخل `doSomething` للمرة الثانية، لن تغلِّف `anotherCtx` السياق `ctx`، وستُعاد القيمة الأصلية `myValue`.

> السياق أداة قوية يمكن من خلالها تخزين جميع القيم التي نريد الاحتفاظ بها. قد يبدو من المغري وضع جميع البيانات في السياق واستخدام تلك البيانات في الدوال بدلًا من المعاملات، ولكن يمكن أن يؤدي ذلك إلى شيفرة يصعب قراءتها والحفاظ عليه. يجب تحقيق التوازن بين البيانات المُخزنة في السياق والبيانات المُمررة إلى دالةٍ ما مثل معاملات.

القاعدة الأساسية المناسبة هي أن أي بيانات مطلوبة لتشغيل دالة ما يجب أن تُمرر مثل معاملات. مثلًا قد يكون من المفيد الاحتفاظ بأسماء المستخدمين ضمن السياق لاستخدامها عند تسجيل المعلومات لاحقًا، إلا أنه من الممكن أن تكون هناك حاجة لاستخدام اسم المستخدم في تحديد ما إذا كانت الدالة يجب أن تعرض بعض المعلومات المحددة المرتبطة به، وبالتالي من الأفضل تضمينها مثل معامل للدالة حتى لو كانت متاحةً في السياق، لكي يسهل معرفة البيانات التي تُستخدم ضمن الدالة لنا وللآخرين.

حدّثنا في هذا القسم برنامجنا لتخزين قيمة في سياق، ثم تغليف السياق لتجاوز هذه القيمة. هذه ليست الأداة الوحيدة التي يمكن للسياقات توفيرها، إذ يمكن للسياقات أن تُستخدم للإشارة إلى أجزاء أخرى من البرنامج عندما ينبغي التوقف عن المعالجة لتجنب هدر الموارد الحاسوبية.

36.4 إنهاء سياق

إحدى الأدوات الأخرى التي يُقدمها السياق `context.Context` هي إمكانية الإشارة إلى أي دالة تستخدمه بأن السياق قد انتهى ويجب عدّه مكتملًا، وبالتالي يمكن للدوال إيقاف تنفيذ أي عمل يؤدونه. هذا يسمح للبرنامج أن يكون أكثر كفاءة، فهو أصبح يعرف متى يجب أن يتوقف عن معالجة طلب ما عندما تنتهي الحاجة إليه.

مثلًا، إذا أرسل المستخدم طلبًا للحصول على صفحة ويب من الخادم وليكن عن طريق الخطأ، ثم ضغط على زر "إيقاف" أو إغلاق المتصفح قبل انتهاء تحميل الصفحة، في هذه الحالة إن لم يُدرك الخادم أن المستخدم قرر

إلغاء العملية، سيعالج هذا الطلب دون جدوى، فالمستخدم لن يرى النتيجة لأنه لا يريدها بعد الآن، وبالتالي تُهدر الموارد على طلب دون فائدة، ولا سيما إذا كانت الصفحة المطلوبة تتطلب تنفيذ بعض الاستعلامات من قاعدة البيانات؛ أما إذا كانت الدالة التي تُخدّم الطلب تستخدم سياقًا، فإنها ستعرف وستُخبر بقية الدوال ذات الصلة بأن السياق انتهى لأن الخادم ألغاه، وبالتالي يمكنهم تخطي تنفيذ أية استعلامات مُتبقية من قاعدة البيانات. يؤدي ذلك إلى الحد من هدر الموارد من خلال إتاحة وقت المعالجة هذا إلى طلب آخر ربما يكون منتظرًا.

سنعدّل في هذا القسم البرنامج ليكون قادرًا على معرفة وقت انتهاء السياق، وسنتعرّف على ثلاثة توابع لإنهاء السياق.

36.4.1 تحديد انتهاء السياق

يحدث تحديد ما إذا كان السياق قد انتهى بنفس الطريقة، وذلك بصرف النظر عن السبب؛ إذ يوفر النوع `context.Context` تابعًا يُسمى `Done` للتحقق من انتهاء سياق ما. يُعيد هذا التابع قناةً channel تُغلق حالما ينتهي السياق، وأي دالة تُتابع هذه القناة توقف تنفيذ أي عملية ذات صلة في حال أُغلقت القناة. لا يُكتب على هذه القناة أية قيم، وبالتالي عند إغلاق القناة تُعيد القيمة `nil` إذا حاولنا قراءتها. إذًا، يمكننا من خلال هذه القناة، إنشاء دوال يمكنها معالجة الطلبات ومعرفة متى يجب أن تكمل المعالجة ومتى يجب أن تتوقف من خلال التحقق الدوري من حالة القناة، كما أن الجمع بين معالجة الطلبات والفحص الدوري لحالة القناة وتعليمة `select` يمكن أن يقدم فائدةً أكبر من خلال السماح بإرسال أو استقبال البيانات من قنوات أخرى في نفس الوقت، إذ تُستخدم التعليمة `select` للسماح للبرنامج بالكتابة أو القراءة من عدة قنوات بصورة متزامنة. يمكن تنفيذ عملية واحدة خاصة بقناة في وقت واحد ضمن تعليمة `select`، لذا نستخدم حلقة for على تعليمة `select` كما سنرى في المثال التالي، لإجراء عدة عمليات على القناة.

يُمكن إنشاء تعليمة `select` من خلال الكلمة المفتاحية `select` متبوعةً بقوسين `{}` مع تعليمة `case` واحدة أو أكثر ضمن القوسين. يمكن أن تكون كل تعليمة `case` عملية قراءة أو كتابة على قناة، وتنتظر تعليمة `select` حتى تُنفّذ إحدى حالتها (تبقى منتظرة حتى تُنفذ إحدى تعليمات `case`)، وفي حال أردنا ألا تنتظر، يمكننا أن نستخدم التعليمة `default`، وهي الحالة الافتراضية التي تُنفّذ في حال عدم تحقق شرط تنفيذ إحدى الحالات الأخرى (تشبه تعليمة `switch`).

توضّح الشيفرة التالية كيف يمكن استخدام تعليمة `select` ضمن دالة، بحيث تتلقى نتائج من قناة وتراقب انتهاء السياق من خلال التابع Done:

```
ctx := context.Background()
resultsCh := make(chan *WorkResult)
for {
  select {
```

```
    case <- ctx.Done():
          // The context is over, stop processing results
          return
    case result := <- resultsCh:
            // عالج النتائج
    }
  }
```

تُمـرر عـادةً قيم كـل من `ctx` و `resultsCh` إلى دالـة مثـل معـاملات، إذ يكـون `ctx` سـياقًا من النـوع `context.Context`، بينما `resultsCh` هي قيمة من قناة يمكننا القراءة منها فقط داخـل الدالـة، وغالبًـا مـا تكون هذه القيمة نتيجة من عامل worker (أو خيوط معالجة جو) في مكانٍ ما. في كل مـرة تُنفـذ فيهـا تعلميـة `select` سيوقف جو تنفيذ الدالة ويراقب جميع تعليمـات `case`، وحالمـا تكـون هنـاك إمكانيـة لتنفيـذ أحـدها (قراءة من قناة كما في في حالة `resultsCh`، أو كتابة أو معرفة حالة القناة عبر `Done`) يُنفذ فـر ع هـذه الحالـة، ولا يمكن التنبؤ بترتيب تنفيذ تعلميات `case` هذه، إذ من الممكن أن تُنفذ أكثر من حالة بالتزامن.

نلاحظ في الشيفرة أعلاه أنه يمكننا استمرار تنفيذ الحلقة إلى الأبد طالما أن السـياق لم ينتهي، أي طالمـا أن `ctx.Done` لم تُشـر إلى إغلاق القنـاة، حيث لا توجـد أي تعليمـات `break` أو `return` إلا داخـل عبـارة `case`. وعلى الرغم من عدم إسناد الحالة `case <- ctx.Done` أي قيمة لأي متغير، إلا أنه سيظل بالإمكان تنفيـذها عند إغلاق `ctx.Done`، لأن القناة تحتوي على قيمة يمكن قراءتها حتى لو لم نُسنِد تلك القيمة وتجاهلناها. إذا لم تُغلق القناة `ctx.Done` (أي لم يتحقق فر ع الحالة `case <- ctx.Done`)، فسوف تنتظـر تعليمـة `select` حتى تُغلق أو حتى يُصبح بالإمكان قراءة قيمة من `resultsCh`.

إذا كانت `resultsCh` تحمل قيمة يمكن قراءتها يُنفّذ فر ع الحالـة الـذي يتضـمنها ويجـري انتظارهـا ريثمـا تنتهي من التنفيذ، وبعدها يجري الدخول في تكرار آخر للحلقة (تُنفّذ حالة واحـدة في الاسـتدعاء الواحـد لتعليمـة `select`)، وكما ذكرنا سابقًا إذا كان بالإمكان تنفيذ الحالتين، يجري الاختيار بينهما عشوائيًا.

في حال وجود تعليمة `default`، فإن الأمر الوحيد الذي يتغـير هـو أنـه يُنفـذ حـالًا في حـال لم تكن إحـدى الحالات الأخرى قابلة للتنفيذ، وبعد تنفيـذ `default` يجـري الخـروج من `select` ثم يجـري الـدخول إليهـا مـرةً أخرى بسبب وجود الحلفة. يؤدي هذا إلى تنفيذ حلقة `for` بسرعة كبيرة لأنها لن تتوقف أبـدًا وتنتظـر القـراءة من قناة. تُسمى الحلقة في هذه الحالة "حلقة مشغولة busy loop" لأنـه بـدلًا من انتظـار حـدوث شـيء مـا، تكـون الحلقة مشغولة بالتكرار مرارًا وتكرارًا. يستهلك ذلك الكثير من دورات وحدة المعالجة المركزية CPU، لأن البرنامج لا يحصل أبدًا على فرصة للتوقف عن التشغيل للسماح بتنفيذ التعليمات البرمجيـة الأخـرى. عمومًـا تكـون هـذه العملية مفيدة أحيانًا، فمثلًا إذا كنا نريد التحقق مما إذا كانت القناة جاهزة لفعـل شـيء مـا قبـل الـذهاب لإجـراء عملية أخرى غير متعلقة بالقناة.

كما ذكرنا، تتجلى الطريقة الوحيدة للخروج من الحلقة في هذا المثال بإغلاق القناة المُعادة من التـابع Done، والطريقة الوحيدة لإغلاق هذه القناة هي إنهاء السياق، بالتالي نحن بحاجة إلى طريقة لإنهائه. تـوفر لغـة جـو عـدة طرق لإجراء ذلك وفقًا للهدف الذي نبتغيه، والخيار المباشر هو استدعاء دالة "إلغاء cancel" السياق.

36.4.2 إلغاء السياق

يعد إلغاء السـياق context Canceling أكـثر طريقـة مباشـرة ويمكن التحكم بهـا لإنهـاء السـياق. يمكننـا -بأسلوب مشابه لتضمين قيمة في سياق باستخدام دالة context.WithValue- ربط دالـة إلغـاء سـياق مـع سياق باستخدام دالة context.WithCancel، إذ تقبل هذه الدالة السـياق الأب مثـل معامـل وتعيـد سـياقًا جديدًا إضافةً إلى دالة يمكن استخدامها لإلغاء السياق المُعاد؛ وكذلك يؤدي استدعاء دالة الحـذف المُعـادة فقـط إلى إلغاء السياق الذي أُعيد مع جميع السياقات الأخرى التي تستخدمه مثل سياق أب، وذلـك بطريقـة مشـابهة أيضًا لأسلوب عمل دالة context.WithValue. هذا لا يعني طبعًا أنه لا يمكن إلغاء السياق الأب الذي مررناه إلى دالة context.WithCancel، بل يعني أنه لا يُلغى إذا استدعيت دالة الإلغاء بهذا الشكل.

لنفتح ملف main.go لنرى كيف نستخدم context.WithCancel:

```
package main
import (
  "context"
  "fmt"
  "time"
)
func doSomething(ctx context.Context) {
  ctx, cancelCtx := context.WithCancel(ctx)
  printCh := make(chan int)
  go doAnother(ctx, printCh)
  for num := 1; num <= 3; num++ {
  printCh <- num
  }
  cancelCtx()
  time.Sleep(100 * time.Millisecond)
  fmt.Printf("doSomething: finished\n")
}
func doAnother(ctx context.Context, printCh <-chan int) {
  for {
        select {
```

```
            case <-ctx.Done():
                  if err := ctx.Err(); err != nil {
                        fmt.Printf("doAnother err: %s\n", err)
                  }
                  fmt.Printf("doAnother: finished\n")
                  return
            case num := <-printCh:
                  fmt.Printf("doAnother: %d\n", num)
            }
      }
}
...
```

أضفنا استيرادًا للحزمة `time` وجعلنا الدالة `doAnother` تقبل قناةً جديدة لطباعة أرقام على شاشة الخـرج. استخدمنا تعليمة `select` ضمن حلقة `for` للقراءة من تلك القناة والتابع `Done` الخاص بالسياق. أنشأنا ضـمن الدالـة `doSomething` سـياقًا يمكن إلغـاؤه إضـافةً إلى قنـاة لإرسـال الأرقـام إليهـا. أضـفنا اسـتدعاءً للدالـة `doAnother` سبقناه بالكلمة المفتاحية `go` ليُنفّذ مثل خيوط معالجة جـو goroutine، ومررنـا لـه السـياق `ctx` والقناة `printCh`. أخيرًا أرسلنا بعض الأرقام إلى القناة ثم ألغينا السياق.

لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
doAnother: 1
doAnother: 2
doAnother: 3
doAnother err: context canceled
doAnother: finished
doSomething: finished
```

تعمل الدالة `doSomething` في الشيفرة السابقة على إرسال العمل إلى خيط معالجة واحد أو أكثر يقـرؤون الأرقام من القناة ويطبعوها، وتكون في هذه الحالة الدالة `doAnother` هي العامل worker وعملهـا هـو طباعـة الأرقام، وحالما يبدأ خيط معالجة جو `doAnother`، تبدأ دالة `doSomething` بإرسال الارقام للطباعة.

تنتظـر تعليمـة `select` -داخـل الدالـة `doAnother`- إغلاق قنـاة `ctx.Done` أو اسـتقبال رقم على القنـاة `printCh`. تبدأ الدالة `doSomething` عملية إرسال الأرقام إلى القناة بعد بدء تنفيذ `doAnother` كما نـرى في

الشيفرة أعلاه، إذ ترسل 3 أرقام إلى القناة، وبالتالي تُفعّل 3 عمليات طباعة `fmt.Printf` لكل رقم (فرع الحالة الثانية داخل تعليمة `select`)، ثم تستدعي دالة `cancelCtx` لإلغاء السياق. بعد أن تقرأ الدالة `doAnother` الأرقام الثلاثة من القناة، ستنتظر العملية التالية من القناة، أي تبقى منتظرة ولن يُنفّذ الفرع المقابل في `select`، وبما أن `doSomething` في هذه الأثناء استدعت `cancelCtx`، بالتالي يُستدعى فرع `ctx.Done`، الذي يستخدم الدالة `Err` التي يوفرها النوع `context.Context` لتحديد كيفية إنهاء السياق. بما أننا ألغينا السياق باستخدام `cancelCtx`، بالتالي سيكون الخطأ الذي نراه هو `context canceled`.

> إذا سبق وشغّلت برامج لغة جو من قبل ونظرت إلى الخرج، فربما سبق وشاهدت الخطأ `context canceled` من قبل، فهو خطأ شائع عند استخدام حزمة `http` من لغة جو، ويهدف لمعرفة وقت قطع اتصال العميل بالخادم، قبل أن يعالج الخادم الاستجابة.

أخيرًا، تستخدم الدالة `doSomething` بعد ذلك الدالة `time.Sleep` للانتظار لفترة قصيرة من الوقت، لتعطي بذلك الدالة `doAnother` وقتًا لمعالجة حالة السياق المُلغى وإنهاء التنفيذ، ثم تطبع الدالة `doSomething` رسالة تُشير إلى انتهاء التنفيذ. تجدر الملاحظة إلى أنه لا داعٍ إلى استخدام دالة `time.Sleep` غالبًا، لكنها ضرورية عندما تنتهي الشيفرة من التنفيذ بسرعة، وسينتهي بدونها البرنامج دون رؤية كامل الخرج على الشاشة.

تكون الدالة `context.WithCancel` ودالة الإلغاء التي تُعيدها مفيدةً أكثر عندما يتطلب الأمر قدرًا كبيرًا من التحكم عند إنهاء السياق، لكن في كثير من الأحيان قد لا نحتاج إلى هذا القدر من التحكم. الدالة التالية المتاحة لإنهاء السياقات في حزمة السياق `context` هي `Context.WithDeadline`، إذ تنهي هذه الدالة السياق تلقائيًا نيابةً عنا.

36.4.3 إعطاء السياق مهلة زمنية للانتهاء

يمكننا تحديد مهلة زمنية Deadline يجب خلالها أن ينتهي السياق باستخدام الدالة `Context.WithDeadline`، وبعد انتهاء هذه المهلة سوف ينتهي السياق تلقائيًا. الأمر أشبه بأن نكون في امتحان، ويكون هناك مهلة محددة لحل الأسئلة، وتُسحب الورقة منا عند انتهائها تلقائيًا، حتى لو لم ننتهي من حلها. لتحديد موعد نهائي لسياق ما، نستخدم الدالة `Context.WithDeadline` مع تمرير السياق الأب وقيمة زمنية من النوع `time.Time` تُشير إلى الموعد النهائي. تُعيد هذه الدالة سياقًا جديدًا ودالة لإلغاء السياق، وكما رأينا في `Context.WithCancel` تُطبق عملية الإلغاء على السياق الجديد وعلى أبنائه (السياقات التي تستخدمه). يمكننا أيضًا إلغاء السياق يدويًا عن طريق استدعاء دالة الإلغاء كما في دالة `context.WithCancel`.

الآن لنفتح ملف البرنامج ونحدثه كي نستخدم دالة `Context.WithDeadline` بدلًا من استخدام الدالة `context.WithCancel`:

```
...
func doSomething(ctx context.Context) {
  deadline := time.Now().Add(1500 * time.Millisecond)
  ctx, cancelCtx := context.WithDeadline(ctx, deadline)
  defer cancelCtx()
  printCh := make(chan int)
  go doAnother(ctx, printCh)
  for num := 1; num <= 3; num++ {
        select {
        case printCh <- num:
              time.Sleep(1 * time.Second)
        case <-ctx.Done():
              break
        }
  }
  cancelCtx()
  time.Sleep(100 * time.Millisecond)
  fmt.Printf("doSomething: finished\n")
}
...
```

تستخدم الشيفرة الآن الدالة `context.WithDeadline` ضمن الدالة `Context.WithDeadline` لإلغاء السياق تلقائيًا بعد 1500 ميلي ثانية (1.5 ثانية) من بدء تنفيذ الدالة، إذ حددنا الوقت من خلال دالة `time.Now`. إضافةً إلى استخدام الدالة `Context.WithDeadline`، أجرينا بعض التعديلات الأخرى؛ فنظرًا لأنه من المحتمل أن ينتهي البرنامج الآن عن طريق استدعاء `cancelCtx` مباشرةً أو الإلغاء التلقائي وفقًا للموعد النهائي، حدّثنا دالة `doSomething`، بحيث نستخدم تعليمة `select` لإرسال الأرقام على القناة. بالتالي، إذا كانت `doAnother` لا تقرأ من `printCh` وكانت قناة `ctx.Done` مُغلقة، ستلاحظ ذلك `doSomething` وتتوقف عن محاولة إرسال الأرقام.

نلاحظ أيضًا استدعاء `cancelCtx` مرتين، مرة عبر تعليمة `defer` ومرة كما في السابق. وجود الاستدعاء الأول غير ضروري طالما أن الاستدعاء الثاني موجود وسيُنفذ دومًا، ولكن من المهم وجوده لو كانت هناك تعليمة `return` أو حدث ما يُمكن أن يتسبب في عدم تنفيذ الاستدعاء الثاني. على الرغم من إلغاء السياق بعد انقضاء المهلة الزمنية، إلا أننا نستدعي دالة الإلغاء، وذلك من أجل تحرير أية موارد مُستخدمة بمثابة إجراء أكثر أمانًا.

لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
doAnother: 1
doAnother: 2
doAnother err: context deadline exceeded
doAnother: finished
doSomething: finished
```

نلاحظ هذه المرة إلغاء السياق بسبب خطأ تجاوز الموعد النهائي`deadline exceeded` قبل طباعة جميع الأرقام الثلاثة، وهذا منطقي فالمهلة الزمنية هي 1.5 ثانية بدءًا من لحظة تنفيذ `doSomething`، وبما أن `doSomething` تنتظر ثانية واحدة بعد إرسال رقم، فستنتهي المهلة قبل طباعة الرقم الثالث. بمجرد انقضاء المهلة الزمنية، ينتهي تنفيذ كل من `doSomething` و `doAnother`، وذلك لأنهما يراقبان لحظة إغلاق قناة `ctx.Done`. لو عدّلنا مدة المهلة وجعلناها أكثر من 3 ثوان، فربما سنشاهد الخطأ `context canceled` يظهر من جديد، وذلك لأن المهلة طويلة.

قد يكون الخطأ `context deadline exceeded` مألوفًا أيضًا، ولا سيما للمبرمجين الذين يستخدمون تطبيقات لغة جو ويقرؤون رسائل الخطأ التي تظهر. هذا الخطأ شائع في خوادم الويب التي تستغرق وقتًا في إرسال الاستجابات إلى العميل. مثلًا، إذا استغرق استعلام من قاعدة البيانات أو عملية ما وقتًا طويلًا، فقد يتسبب ذلك بإلغاء سياق الطلب وظهور هذا الخطأ لأن الخادم لا يسمح بتجاوز مهلة معينة في معالجة طلب ما. يسمح لنا إلغاء السياق باستخدام `context.WithCancel` بدلًا من `cont4ext.WithCancel` بإلغاء السياق تلقائيًا بعد انتهاء مهلة نتوقع أنها كافية، دون الحاجة إلى تتبع ذلك الوقت. بالتالي: إذا كنا نعرف متى يجب أن ينتهي السياق (أي نعرف المهلة الكافية)، ستكون هذه الدالة خيارًا مناسبًا.

أحيانًا ربما لا نهتم بالوقت المحدد الذي ينتهي فيه السياق، وتريد أن ينتهي بعد دقيقة واحدة من بدئه. هنا يمكننا أيضًا استخدام الدالة `context.WithDeadline` مع بعض توابع الحزمة `time` لتحقيق الأمر، لكن لغة جو توفر لنا الدالة `context.WithTimeout` لتبسيط الأمر.

36.4.4 إعطاء السياق وقت محدد

تؤدي دالة `context.WithTimeout` نفس المهمة التي تؤديها الدالة السابقة، والفرق الوحيد هو أننا في `context.WithDeadline` نمرر قيمة زمنية محددة من النوع `time.Time` لإنهاء السياق، أما في `context.WithTimeout` نحتاج فقط إلى تمرير المدة الزمنية، أي قيمة من النوع `time.Duration`.

إذًا، يمكننا استخدام `context.WithDeadline` إذا أردنا تحديد وقت معين `time.Time`. ستحتاج -بدون `context.WithTimeout`- إلى استخدام الدالة `time.Now()` والتابع `Add` لتحديد المهلة الزمنية، أما مع `context.WithTimeout`، فيمكنك تحديد المهلة مباشرةً.

لنفتح ملف البرنامج مجددًا ونعدله، بحيث نستخدم `context.WithTimeout` بدلًا من استخدام `context.WithDeadline`:

```
...
func doSomething(ctx context.Context) {
  ctx, cancelCtx := context.WithTimeout(ctx, 1500*time.Millisecond)
  defer cancelCtx()
  ...
}
...
```

لنُشغّل ملف البرنامج `main.go` من خلال الأمر `go run`:

```
$ go run main.go
```

ليكون الخرج على النحو التالي:

```
doAnother: 1
doAnother: 2
doAnother err: context deadline exceeded
doAnother: finished
doSomething: finished
```

نلاحظ أن الخرج ورسالة الخطأ هي نفسها التي حصلنا عليها في الخرج السابق عندما استخدمنا الدالة `context.WithDeadline`، ورسالة الخطأ أيضًا نفسها التي تظهر أن `context.WithTimeout` هي فعليًا مغلّف يجري عمليات رياضية نيابةً عنك.

استخدمنا في هذا القسم ثلاث طرق مختلفة لإنهاء السياق `context.Context`.؛ إذ بدأنا بالدالة `context.WithCancel` التي تسمح لنا باستدعاء دالة لإلغاء السياق؛ ثم الدالة `context.WithDeadline` مع قيمة `time.Time` لإنهاء السياق في وقت محدد؛ ثم الدالة `context.WithTimeout` مع قيمة `time.Duration` لإنهاء السياق بعد مدة معينة.

نضمن من خلال استخدام هذه الدوال أن البرنامج لن يستهلك موارد الحاسب أكثر مما تحتاجه، كما يُسهّل فهم الأخطاء التي تُسببها السياقات عملية استكشاف الأخطاء وإصلاحها في برامج جو.

36.5 الخاتمة

أنشأنا خلال هذا الفصل برنامجًا يستخدم حزمة السياق `context` التي تقدمها لغة البرمجة جو بطرق مختلفة. أنشأنا دالةً تقبل سياقًا `context.Context` مثل معامل، واستخدمنا كلًا من الدالتين `context.TODO` و `context.Background` لإنشاء سياق فارغ.

بعد ذلك، استخدمنا الدالة `context.WithValue` لإنشاء سياق جديد يُغلّف سياقًا آخر ويحمل قيمة جديدة، وتعرّفنا على كيفية قراءة هذه القيمة لاحقًا من خلال التابع `Value` من داخل الدوال الأخرى التي تستخدم هذا السياق.

بعدها، تعرفنا على التابع `Done` الذي يُساعدنا في معرفة الوقت الذي تنتفي فيه الحاجة إلى إبقاء السياق وتعلمنا أيضًا كيف نلغي السياق بطرق مختلفة من خلال الدوال التالية `context.WithCancel` و `context.WithDeadline` و `context.WithTimeout` وكيف نضع حدًا للمدة التي يجب أن تُنفّذ بها التعليمات البرمجية التي تستخدم تلك السياقات.

37. كيفية استخدام صيغة JSON

إحدى الخصائص المهمة في البرامج الحديثة هو إمكانية التواصل مع البرامج الأخرى، فسواء كان برنامج جـو يتحقق فيما إذا كان لدى المستخدم حق الوصول إلى برنامج آخر، أو برنامج جافـا سـكريبت JavaScript يحصـل على قائمة بالطلبات السابقة لعرضها على موقع ويب، أو برنامج رست Rust يقرأ نتائج اختبار من ملف، فهنـاك حاجة إلى طريقة نزوّد البرامج من خلالها بالبيانات.

لدى أغلب لغات البرمجة طريقة خاصة في تخزين البيانات داخليًّا، والتي لا تفهمها اللغات البرمجية الأخرى. للسماح لهذه اللغات بالتفاعل مع بعضها، يجب تحويل البيانـات إلى تنسـيق أو صـيغة مشـتركة يمكنهم فهمهـا جميعًا. إحدى هذه الصيغ هي صيغة جسون JSON، إنها وسيلة شائعة لنقل البيانـات عـبر الإنـترنت وكـذلك بين البرامج في نفس النظام. تمتلك لغة جو والعديـد من لغـات البرمجـة الأخـرى طريقـة لتحويـل البيانـات من وإلى صيغة JSON في مكتباتها القياسية.

سنُنشئ في هذا الفصل برنامجًا يستخدم حزمة `encoding/json` لتحويل صيغة البيانـات من النـوع `map` إلى بيانات `json`، ثم من النوع `struct` إلى `json`، كما سنتعلم كيفية إجراء تحويل عكسي لهما.

37.1 المتطلبات

- إصدار مُثبّت من جو 1.16 أو أعلى، ويمكنك الاستعانة بالتعليمات الواردة في الفصل الأول من الكتاب، لتثبيت لغة جو Go وإعداد بيئة تطوير محلية بحسب نظام تشغيلك.
- معرفة بكيفية التعامل مع التاريخ والوقت في لغة جو. يمكنك الاطلاع على فصـل التعامـل مـع التـاريخ والوقت في لغة جو Go.
- معرفة مسبقة بصيغة جسون JSON.

- معرفة مسبقة بكيفية التعامل مع وسوم البنية Struct tags لتخصيص حقول البنية.

37.2 استخدام الخرائط Maps لتوليد بيانات بصيغة JSON

توفر حزمة `encoding/json` بعض الدوال لتحويل البيانات من وإلى جسون JSON. الدالة الأولى هي `json.Marshal`. التنظيم Marshaling أو المعروف أيضًا باسم السلسلة Serialization، هو عملية تحويل بيانات البرنامج من الذاكرة إلى تنسيق يمكن نقله أو حفظه في مكان آخر، وهذا ما تفعله الدالة `json.Marshal` في لغة جو، إذ تحول بيانات البرنامج قيد التشغيل (الموجود في الذاكرة) إلى بيانات جسون. تقبل هذه الدالة أي قيمة من النوع واجهة `interface{}` لتنظيمها بصيغة جسون، لذلك يُسمح بتمرير أي قيمة مثل معامل، لتُعيد الدالة البيانات ممثلة بصيغة جسون في النتيجة.

سننشئ في هذا القسم برنامجًا يستخدم دالة `json.Marshal` لإنشاء ملف جسون يحتوي أنواعًا مختلفة من البيانات من قيم `map`، ثم سنطبع هذه القيم على شاشة الخرج. تُمثّل بيانات جسون غالبًا على شكل كائن مفاتيحه من سلاسل نصيّة وقيمه من أنواع مختلفة، وهذا يُشبه آلية تمثيل البيانات في خرائط جو، لذا فإن الطريقة الأفضل لإنشاء بيانات جسون في لغة جو هي وضع البيانات ضمن خريطة `map` مع مفاتيح من النوع `string` وقيم من النوع `interface{}`. وتُفسّر مفاتيح `map` مباشرةً على أنها مفاتيح جسون JSON، ويمكن أن تكون قيم النوع `interface` من أي نوع بيانات في لغة البرمجة Go، مثل النوع `int`، أو `string`، أو حتى `map[string]interface{}`.

لبدء استخدام الحزمة `encoding/json`، وكما هو معتاد، سنحتاج لبدء إنشاء برامجنا إلى إنشاء مجلد للعمل ووضع الملفات فيه، ويمكن وضع المجلد في أي مكان على الحاسب، إذ يكون للعديد من المبرمجين عادةً مجلدٌ يضعون داخله كافة مشاريعهم. سنستخدم في هذا الفصل مجلدًا باسم `projects`، لذا فلننشئ هذا المجلد وننتقل إليه:

```
$ mkdir projects
$ cd projects
```

سنشغّل من داخل هذا المجلد، الأمر `mkdir` لإنشاء مجلد `jsondata` ثم سنستخدم `cd` للانتقال إليه:

```
$ mkdir jsondata
$ cd jsondata
```

يمكننا الآن فتح ملف main.go باستخدام محرر نانو nano أو أي محرر آخر تريده:

```
$ nano main.go
```

نضيف داخل ملف main.go دالة main لتشغيل البرنامج، ثم نضيف قيمة map[string]interface{} مع مفاتيح وقيم من أنواع مختلفة، ثم نستخدم الدالة json.Marshal لتحويل بيانات map إلى بيانات جسون:

```
package main
import (
  "encoding/json"
  "fmt"
)
func main() {
  data := map[string]interface{}{
        "intValue":  1234,
        "boolValue":  true,
        "stringValue": "hello!",
        "objectValue": map[string]interface{}{
              "arrayValue": []int{1, 2, 3, 4},
        },
  }
  jsonData, err := json.Marshal(data)
  if err != nil {
        fmt.Printf("could not marshal json: %s\n", err)
        return
  }
  fmt.Printf("json data: %s\n", jsonData)
}
```

نلاحظ في المتغير data أن كل قيمة لديها مفتاح string، لكن قيم هذه المفاتيح تختلف، فأحدها int وأحدها bool والأخرى هي خريطة map[string]interface{} مع قيم int بداخلها. عند تمرير المتغير data إلى json.Marshal، ستنظر الدالة إلى جميع القيم التي تتضمنها الخريطة وتحدد نوعها وكيفية تمثيلها في جسون، وإذا حدثت مشكلة في تفسير بيانات الخريطة، ستُعيد خطأ يصف المشكلة.

إذا نجحت العملية، سيتضمّن المتغير jsonData بيانات من النوع []byte تُمثّل البيانات التي جرى تنظيمها إلى صيغة جسون. وبما أن بالإمكان تحويل []byte إلى قيمة string باستخدام التعليمة myString := string(jsonData) أو العنصر النائب %s ضمن تنسيق سلسلة، فيمكننا طباعة بيانات جسون على شاشة الخرج باستخدام دالة الطباعة fmt.Printf.

بعد حفظ وإغلاق الملف، لنُشغّل ملف البرنامج main.go من خلال الأمر go run:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
json data: {"boolValue":true,"intValue":1234,"objectValue":
{"arrayValue":[1,2,3,4]},"stringValue":"hello!"}
```

نلاحظ من الخرج أن قيمة جسون هي كائن مُمثّل بأقواس معقوصة curly braces `{}` تحيط به، وأن جميع قيم المتغير `data` موجودة ضمن هذين القوسين. نلاحظ أيضًا أن خريطة المفتاح `objectValue` الذي هـو `{}` `map[string]interface` قد جرى تفسيره إلى كائن جسون آخر مُمثّل بأقواس معقوصة `{}` تحيط به أيضًـا، ويتضمن أيضًا المفتاح `arrayValue` بداخله مع مصفوفة القيم المقابلة `[1،2،3،4]`.

37.2.1 ترميز البيانات الزمنية في JSON

لا تقتصر قـدرات الحزمـة `encoding/json` على إمكانيـة تمثيـل البيانـات من النـوع `string` و `int`، إذ يمكنها التعامل مع أنواع أعقد مثل البيانات الزمنية من النوع `time.Time` من الحزمة `time`.

لنفتح ملــف البرنــامج main.go مــرةً أخــرى ونضــيف قيمــة من النــوع `time.Time` باســتخدام الدالة `time.Date`:

```
package main
import (
  "encoding/json"
  "fmt"
  "time"
)
func main() {
  data := map[string]interface{}{
        "intValue":  1234,
        "boolValue":  true,
        "stringValue": "hello!",
        "dateValue":  time.Date(2022, 3, 2, 9, 10, 0, 0, time.UTC),
        "objectValue": map[string]interface{}{
              "arrayValue": []int{1, 2, 3, 4},
        },
  }
  ...
}
```

يؤدي هذا التعديل إلى ضبط التاريخ على March 2, 2022 والوقت على 9:10:00 AM في المنطقة الزمنية UTC وربطهم بالمفتاح `dateValue`.

لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيعطي الخرج التالي:

```
json data: {"boolValue":true,"dateValue":"2022-03-
02T09:10:00Z","intValue":1234,"objectValue":{"arrayValue":
[1,2,3,4]},"stringValue":"hello!"}
```

نلاحظ هذه المرة في الخرج الحقل `dateValue` ضمن بيانات جسون، وأن الوقت مُنسّقٌ وفقًا لتنسيق RFC 3339، وهو تنسيق شائع يُستخدم لنقل التواريخ والأوقات على أنها قيم نصية `string`.

37.2.2 ترميز قيم Null في JSON

قد نحتاج إلى التعامل مع قيم `null`، وتحويلها إلى صيغة جسون أيضًا. يمكن لحزمة `encoding/json` تولي هذه المهمة أيضًا، إذ يمكننا التعامل مع قيم `nil` (تُقابل قيم `null`) مثل أي قيمة من نوع آخر ضمن الخريطة `map`.

لنفتح ملف main.go ولنضع قيمتي `null` ضمن `map` كما يلي:

```
...
func main() {
  data := map[string]interface{}{
        "intValue":  1234,
        "boolValue":  true,
        "stringValue": "hello!",
        "dateValue":  time.Date(2022, 3, 2, 9, 10, 0, 0, time.UTC),
        "objectValue": map[string]interface{}{
              "arrayValue": []int{1, 2, 3, 4},
        },
        "nullStringValue": nil,
        "nullIntValue":  nil,
  }
  ...
}
```

وضعنا في الشيفرة أعلاه قيمتي `null` مع مفتاحين مختلفين، هما `nullStringValue` و `nullIntValue` على التوالي، وعلى الرغم من أن أسماء المفاتيح تُشير إلى قيم `string` و `int`، لكن هي ليست كذلك (مجرد أسماء). طبعًا كل القيم ضمن الخريطة `map` مُشتقة من النوع `interface{}` والقيمة `nil` هي قيمة مُحتملة لهذا النوع وبالتالي تُفسّر على أنها `null` فقط، وهذا كل شيء. لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج كما يلي:

```
json data: {"boolValue":true,"dateValue":"2022-03-
02T09:10:00Z","intValue":1234,"nullIntValue":null,"nullStringValue":nu
ll,"objectValue":{"arrayValue":[1,2,3,4]},"stringValue":"hello!"}
```

نلاحظ في الخرج أن الحقلين `nullIntValue` و `nullStringValue` مُضمنان مع قيمة `null` لكل منهما، بالتالي تمكنا من استخدام `map[string]interface{}` مع قيم `null` دون أية مشكلات.

أنشأنا في هذا القسم برنامجًا يمكنه تحويل قيم بيانات من النوع `map[string]interface{}` إلى بيانات جسون. أضفنا بعد ذلك حقلًا يستخدم بيانات زمنية من النوع `time.Time` ضمن الخريطة، وأخيرًا أضفنا حقلين يستخدمان القيمة `null`.

بالرغم من مرونة استخدام `map[string]interface{}` لتحويل البيانات إلى بيانات جسون، إلا أنه قد يكون عُرضةً لحدوث أخطاء غير مقصودة، ولا سيما إذا كنا بحاجة إلى إرسال نفس البيانات إلى عدة أماكن. إذا أرسلنا نُسخًا من هذه البيانات إلى أكثر من مكان ضمن الشيفرة، فربما نُغيّر عن طريق الخطأ اسم حقل أو نضع قيمة غير صحيحة ضمن حقل. في هكذا حالات ربما يكون من المفيد استخدام النوع `struct` لتمثيل البيانات التي نُريد تحويلها إلى جسون.

37.3 استخدام البنى Structs لتوليد بيانات بصيغة جسون

تُعَدّ جو لغةً ثابتة الأنواع statically-typed language مثل لغة C أو جافا Java أو C++، وهذا يعني أن كل تعليمة في البرنامج تُفحَص في وقت التصريف. وتتمثل فائدة ذلك في السماح للمُصرّف باستنتاج نوع المتغيرات والتحقق منها وفرض التناسق بين قيم المتغيرات.

تستفيد الحزمة `encoding/json` من بنية `struct` وذلك من خلال تعريف بنية struct تُمثّل بيانات جسون. يمكننا التحكم في كيفية تفسير البيانات التي تتضمنها البنية باستخدام وسوم البنية Struct tags. سنعدّل البرنامج السابق خلال هذا القسم، لاستخدام بنية `struct` بدلًا من خريطة `map`، لتوليد بيانات جسون.

عند استخدام `struct` لتعريف بيانات جسون، يجب علينا تصدير أسماء الحقول (وليس اسم النوع `struct` نفسه) التي نريد تحويلها إلى جسون، وذلك بأن نبدأ أسماء الحقول بحرف كبير (أي بدلًا من كتابة `intValue` نكتب `IntValue`) وإلا لن تكون الحزمة `encoding/json` قادرةً على الوصول إلى هذه الحقول لتحويلها إلى جسون.

الآن بالنسبة لأسماء الحقول، إذا لم نستخدم وسوم البنية للتحكم في تسمية هذه الحقول، ستُفسّر كما هي مباشرةً ضمن البنية. قد يكون استخدام الأسماء الافتراضية هو ما نريده في بيانات جسون، وذلك وفقًا للطريقة التي نرغب بها بتنسيق بياناتنا، وبذلك لن نحتاج في هذه الحالة إلى أية وسوم. يستخدم العديد من المبرمجين تنسيقات أسماء، مثل `intValue`، أو `int_value` مع حقول البيانات، وسنحتاج في هذه الحالة إلى وسوم البنية للتحكم في كيفية تفسير هذه الأسماء.

سيكون لدينا في المثال التالي بنية `struct` مع حقل وحيد اسمه `IntValue` وسنحول هذه البنية إلى صيغة جسون:

```
type myInt struct {
  IntValue int
}
data := &myInt{IntValue: 1234}
```

إذا حوّلنا المتغير `data` إلى صيغة جسون باستخدام الدالة `json.Marshal`، سنرى الخرج التالي:

```
{"IntValue":1234}
```

لكن لو كنا نريد أن يكون اسم الحقل في ملف جسون هو `intValue` بدلًا من `IntValue`، سنحتاج إلى إخبار `encoding/json` بذلك. بما أن `json.Marshal` لا تعرف ماذا نتوقع أن يكون اسم بيانات جسون، سنحتاج إلى إخبارها من خلال إضافة وسم البنية `json` بعد اسم الحقل مباشرةً مع إرفاقه بالاسم الذي نريد أن يظهر به في صيغة جسون. إذًا، من خلال إضافة هذا الوسم إلى الحقل `IntValue` مع الاسم الذي نريد يظهر به `intValue`، ستستخدم الدالة `json.Marshal` الاسم الذي نريده اسمًا للحقل ضمن صيغة جسون:

```
type myInt struct {
  IntValue int `json:"intValue"`
}
data := &myInt{IntValue: 1234}
```

إذا حوّلنا المتغير `data` إلى صيغة جسون باستخدام الدالة `json.Marshal`، سنرى الخرج التالي، وكما نلاحظ فإنه يستخدم اسم الحقل الذي نريده:

```
{"intValue":1234}
```

سنعدل البرنامج الآن لتعريف نوع بيانات `struct` يُمكن تحويله إلى بيانات جسون. سنضيف بنيةً باسم `myJSON` لتمثيل البيانات بطريقة يمكن تحويلها إلى جسون ونضيف البنية `myObject` التي ستكون قيمة للحقل `ObjectValue` ضمن البنية `myJSON`. سنضيف أيضًا وسمًا لكل اسم حقل ضمن البنية `myJSON` لتحديد الاسم الذي نريد أن يظهر به الحقل ضمن بيانات جسون. يجب أيضًا أن نحدّث الإسناد الخاص بالمتغير `data` بحيث نسند له بنية `myJSON` مع التصريح عنه بنفس الطريقة التي نتعامل مع بنى جو الأخرى.

```
...
type myJSON struct {
  IntValue    int    `json:"intValue"`
  BoolValue    bool   `json:"boolValue"`
  StringValue   string  `json:"stringValue"`
  DateValue    time.Time `json:"dateValue"`
  ObjectValue   *myObject `json:"objectValue"`
  NullStringValue *string  `json:"nullStringValue"`
  NullIntValue  *int   `json:"nullIntValue"`
}
type myObject struct {
  ArrayValue []int `json:"arrayValue"`
}
func main() {
  otherInt := 4321
  data := &myJSON{
        IntValue:  1234,
        BoolValue:  true,
        StringValue: "hello!",
        DateValue:  time.Date(2022, 3, 2, 9, 10, 0, 0, time.UTC),
        ObjectValue: &myObject{
              ArrayValue: []int{1, 2, 3, 4},
        },
        NullStringValue: nil,
        NullIntValue:  &otherInt,
  }
  ...
}
```

تُشبه معظم التغييرات في الشيفرة أعلاه ما فعلناه في المثال السابق مع الحقل `IntValue`، إلا أن هناك بعض الأشياء تستحق الإشارة إليها. أحد هذه الأشياء هو الحقل `ObjectValue` الذي يستخدم قيمةً مرجعية

`*myObject` لإخبار دالة `json.Marshal` -التي تؤدي عملية التنظيم- إلى وجود قيمة مرجعية من النوع `myObject` أو قيمة `nil`. بهذه الطريقة نكون قد عرّفنا كائن جسون بأكثر من طبقة، وفي حال كانت هذه الطريقة مطلوبة، سيكون لدينا بنيةً أخرى من نوع `struct` داخل النوع `myObject`، وهكذا، وبالتالي نلاحظ أنه بإمكاننا تعريف كائنات جسون أعقد وأعقد باستخدام أنواع `struct` وفقًا لحاجتنا.

واحد من الأشياء الأخرى التي تستحق الذكر هي الحقلين `NullStringValue` و `NullIntValue`، وعلى عكس `StringValue` و `IntValue`؛ أنواع هذه القيم هي أنواع مرجعية `*int` و `*string`، وقيمها الافتراضية هي قيم صفريّة أي `nil` وهذا يُقابل القيمة 0 لنوع البيانات `int` والسلسلة الفارغة `''` لنوع البيانات `string`. يمكننا من الكلام السابق أن نستنتج أنه في حال أردنا التعبير عن قيمة من نوع ما تحتمل أن تكون `nil`، فيجب أن نجعلها قيمةً مرجعية. مثلًا لو كنا نريد أن نعبر عن قيمة حقل تُمثّل إجابة مُستخدم عن سؤال ما، فهنا قد يُجيب المُستخدم عن السؤال أو قد لا يُجيب (نضع `niil`).

نُعدّل قيمة الحقل `NullIntValue` بضبطه على القيمة `4321` لنُظهر كيف يمكن إسناد قيمة لنوع مرجعي مثل `*int`. تجدر الإشارة إلى أنه في لغة جو، يمكننا إنشاء مراجع لأنواع البيانات الأولية primitive types فقط، مثل `int` و `string` باستخدام المتغيرات. إذًا، لإسناد قيمة إلى الحقل `NullIntValue`، نُسند أولًا قيمةً إلى متغير آخر `otherInt`، ثم نحصل على مرجع منه `&otherInt` (بدلًا من كتابة `&4321` مباشرةً).

لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
json data:
{"intValue":1234,"boolValue":true,"stringValue":"hello!","dateValue":"
2022-03-02T09:10:00Z","objectValue":{"arrayValue":
[1,2,3,4]},"nullStringValue":null,"nullIntValue":4321}
```

نلاحظ أن هذا الناتج هو نفسه عندما استخدمنا `map[string]interface{}`، باستثناء أن قيمة `nullIntValue` هذه المرة هي `4321` لأن هذه هي قيمة `otherInt`.

يستغرق الأمر في البداية بعض الوقت لتعريف البنية `struct` وإعداد حقولها، ولكن يمكننا بعد ذلك استخدامها مرارًا وتكرارًا في الشيفرة، وستكون النتيجة هي نفسها بغض النظر عن مكان استخدامها، كما يمكننا تعديلها من مكان واحد بدلًا من تعديلها في كل مكان تتواجد نسخة منها فيه كما في حالة الخرائط maps.

تتيح لنا الدالة `json.Marshal` إمكانية تحديد الحقول التي نُريد تضمينها في جسون، في حال كانت قيمة تلك الحقول صفريّة (أي Null وهذا يُكافئ 0 في حالة `int` و `false` في حالة `bool` والسلسلة الفارغة في حالة `string` ..إلخ). قد يكون لدينا أحيانًا كائن جسون كبير أو حقول اختيارية لا نريد تضمينها دائمًا في بيانات

جسون، لذا يكون تجاهل هذه الحقول مفيدًا. يكون التحكم في تجاهل هذه الحقول -عندما تكون قيمها صفريّة أو غير صفريّة- باستخدام الخيار `omitempty` ضمن وسم بنية `json`.

لنُحدّث البرنامج السابق لإضافة الخيار `omitempty` إلى حقل `NullStringValue` وإضافة حقل جديد يسمى `EmptyString` مع نفس الخيار:

```
...
type myJSON struct {
  ...
  NullStringValue *string  `json:"nullStringValue,omitempty"`
  NullIntValue  *int   `json:"nullIntValue"`
  EmptyString   string  `json:"emptyString,omitempty"`
}
...
```

بعد تحويل البنية `myJSON` إلى بيانات جسون، سنلاحظ أن الحقل `EmptyString` والحقل `NullStringValue` غير موجودان في بيانات جسون، لأن قيمهما صفريّة.

لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
json data:
{"intValue":1234,"boolValue":true,"stringValue":"hello!","dateValue":"
2022-03-02T09:10:00Z","objectValue":{"arrayValue":
[1,2,3,4]},"nullIntValue":4321}
```

نلاحظ أن الحقل `nullStringValue` لم يعد موجودًا في الخرج، لأنه يُعد حقلًا بقيمة `nil`، بالتالي فإن الخيار `omitempty` استبعده من الخرج. نفس الأمر بالنسبة للحقل `emptyString`، لأن قيمته صفريّة (القيمة الصفريّة تُكافئ `nil` كما سبق وذكرنا).

استخدمنا في هذا القسم أسلوب البُنى `struct` بدلًا من الخرائط maps، لتمثيل كائن جسون في برنامجنا، ثم حولناها إلى بيانات جسون باستخدام الدالة `json.Marshal`. تعلمنا أيضًا كيفية تجاهل القيم الصفريّة من بيانات جسون.

هذه البيانات (بعد تحويلها لصيغة جسون) قد تُرسل إلى برامج أخرى، وبالتالي إذا كان البرنامج الآخر مكتوب بلغة جو، يجب أن نعرف كيف يمكننا قراءة هذا النوع من البيانات. يمكنك أن تتخيل الأمر على أنه برنامجي جو A و B أحدهما مُخدم والآخر عميل. يُرسل A طلبًا إلى B، فيعالجه ويرسله بصيغة جسون إلى A، ثم يقرأ A هذه

البيانات. لأجل ذلك توفر الحزمة `encoding/json` طريقةً لفك ترميز بيانات جسون وتحويلها إلى أنواع جو المقابلة (مجرد عملية عكسية). سنتعلم في القسم التالي كيفية قراءة بيانات جسون وتحويلها إلى خرائط maps.

37.4 تحليل بيانات جسون باستخدام الخرائط

بطريقة مشابهة لما فعلناه عندما استخدمنا `map[string]interface{}` مثل طريقة مرنة لتوليد بيانات جسون، يمكننا استخدامها أيضًا مثل طريقة مرنة لقراءة بيانات جسون. تعمل الدالة `json.Unmarshal` بطريقة معاكسة للدالة `json.Marshal`، إذ تأخذ بيانات جسون وتحولها إلى بيانات جو، وتأخذ أيضًا متغيرًا لوضع البيانات التي جرى فك تنظيمها فيه، وتعيد إما خطأ `error` في حال فشل عملية التحليل أو `nil` في حال نجحت.

سنعدّل برنامجنا في هذا القسم، بحيث نستخدم الدالة `json.Unmarshal` لقراءة بيانات جسون من سلسلة وتخزينها في متغير من النوع `map`، وطباعة الخرج على الشاشة. لنعدّل البرنامج إذًا، بحيث نفك تنظيم البيانات باستخدام الدالة السابقة ونحولها إلى خريطة `map[string]interface{}`. لنبدأ باستبدال المتغير `data` الأصلي بمتغير `jsonData` يحتوي على سلسلة جسون، ثم نُصرّح عن متغير `data` جديد على أنه `map[string]interfac{}` لتلقي بيانات جسون، ثم نستخدم الدالة `json.Unmarshal` مع هذه المتغيرات للوصول إلى بيانات جسون:

```
...
func main() {
  jsonData := `
        {
              "intValue":1234,
              "boolValue":true,
              "stringValue":"hello!",
              "dateValue":"2022-03-02T09:10:00Z",
              "objectValue":{
                    "arrayValue":[1,2,3,4]
              },
              "nullStringValue":null,
              "nullIntValue":null
        }
  `
  var data map[string]interface{}
  err := json.Unmarshal([]byte(jsonData), &data)
  if err != nil {
```

```
        fmt.Printf("could not unmarshal json: %s\n", err)
        return
    }
    fmt.Printf("json map: %v\n", data)
}
```

جرى إسناد قيمة المتغير jsonData في الشيفرة أعلاه من خلال سلسلة نصية أولية، وذلك للسماح بكتابة البيانات ضمن أسطر متعددة لتسهيل القراءة. بعد التصريح عن المتغير data على أنه متغير من النوع map[string]interface{}، نمرر jsonData والمتغير data إلى الدالة json.Unmarshal لفك تنظيم بيانات جسون وتخزين النتيجة في data. يُمرَّر المتغير jsonData إلى دالة فك التنظيم على شكل مصفوفة بايت []byte، لأن الدالة تتطلب النوع []byte، والمتغير jsonData عُرّف على أنه قيمة من نوع سلسلة نصية string. طبعًا هذا الأمر ينجح، ففي لغة جو، يمكن تحويل string إلى []byte والعكس. بالنسبة للمتغير data، ينبغي تمريره مثل مرجع، لأن الدالة تتطلب معرفة موقع المتغير في الذاكرة. أخيرًا، يجري فك تنظيم البيانات وتخزين النتيجة في المتغير data، لنطبع النتيجة بعدها باستخدام دالة fmt.Printf.

لنُشغّل ملف البرنامج main.go من خلال الأمر go run:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
json map: map[boolValue:true dateValue:2022-03-02T09:10:00Z
intValue:1234 nullIntValue:<nil> nullStringValue:<nil>
objectValue:map[arrayValue:[1 2 3 4]] stringValue:hello!]
```

يُظهر الخرج السابق نتيجة تحويل بيانات جسون إلى خريطة map. نلاحظ أن جميع الحقول من بيانات جسون موجودة، بما في ذلك القيم الصفريّة null. وبما أن بيانات جو الآن مُخزنة في قيمة من النوع map[string]interface{}، فسيكون لدينا القليل من العمل مع بياناتها؛ إذ نحتاج إلى الحصول على القيمة من الخريطة باستخدام قيمة مفتاح string معينة، وذلك للتأكد من أن القيمة التي تلقيناها هي القيمة التي نتوقعها، لأن القيمة المعادة هي قيمة من النوع interface{}.

لنفتح ملف main.go ونُحدّث البرنامج لقراءة حقل dateValue:

```
...
func main() {
    ...
    fmt.Printf("json map: %v\n", data)
    rawDateValue, ok := data["dateValue"]
```

```
    if !ok {
        fmt.Printf("dateValue does not exist\n")
        return
    }
    dateValue, ok := rawDateValue.(string)
    if !ok {
        fmt.Printf("dateValue is not a string\n")
        return
    }
    fmt.Printf("date value: %s\n", dateValue)
}
```

استخدمنا في الشيفرة أعلاه المفتاح `dateValue` لاستخراج قيمة من الخريطة `map` بكتابة `data["dateValue"]`، وخزّنا النتيجة في `rawDateValue` ليكون قيمةً من النوع `interface{}`، واستخدمنا المتغير `ok` للتأكد من أن الحقل ذو المفتاح `dateValue` موجود ضمن الخريطة. استخدمنا بعدها توكيد النوع type assertion، للتأكد من أن `rawDateValue` هو قيمة `string`، وأسندناه إلى المتغير `dateValue`. استخدمنا بعدها المتغير `ok` للتأكد من نجاح عملية التوكيد. أخيرًا، طبعنا `dateValue` باستخدام دالة الطباعة `fmt.Printf`.

لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج كما يلي:

```
json map: map[boolValue:true dateValue:2022-03-02T09:10:00Z
intValue:1234 nullIntValue:<nil> nullStringValue:<nil>
objectValue:map[arrayValue:[1 2 3 4]] stringValue:hello!]
date value: 2022-03-02T09:10:00Z
```

يمكننا أن نلاحظ في السطر الأخير من الخرج استخراج قيمة الحقل `dateValue` من الخريطة `map` وتغير نوعها إلى `string`.

استخدمنا في هذا القسم الدالة `json.Unmarshal` لفك تنظيم unmarshal بيانات جسون وتحويلها إلى بيانات في برنامج جو بالاستعانة بمتغير من النوع `map[string]interface{}`. بعد ذلك استخرجنا قيمة الحقل `dateValue` من الخريطة التي وضعنا فيها بيانات جسون وطبعناها على الشاشة.

الجانب السيء في استخدام النوع `map[string]interface{}` في عملية فك التنظيم، هو أن مُفسّر اللغة لا يعرف أنواع الحقول التي جرى فك تنظيمها؛ فكل ما يعرفه أنها من النوع `interface{}`، وبالتالي لا

يمكنـه أن يفعـل شـيئًا أكـثر من تخمين الأنـواع. بالتـالي لن يجـري فـك تنظيم أنـواع البيانـات المعقـدة، مثـل `time.Time` في الحقـل `dateValue` إلى بيانـات من النـوع `time.Time`، وإنمـا تُفسّـر على أنهـا `string`. تحدث مشكلة مماثلة إذا حاولنا الوصول إلى أي قيمـة رقميـة number في الخريطـة بهـذه الطريقـة، لأن الدالـة `json.Unmarshal` لا تعرف مـا إذا كـان الـرقم من النـوع `int` أو `float` أو `int64` ..إلخ. لـذا يكـون التخمين الأفضل هو عدّ الرقم من النوع `float64` لأنه النوع الأكثر مرونة.

نستنتج مما سبق جانب إيجابي للخرائط وهو مرونة استخدامها، وجانب سيئ يتجلى بالمشـكلات السـابقة. هنا تأتي ميزة استخدام البنى لحل المشكلات السابقة. بطريقة مشابهة لآليـة تنظيم البيانـات من بنيـة `struct` باسـتخدام الدالـة `json.Marshal` لإنتـاج بيانـات جسـون، يمكن إجـراء عمليـة معاكسـة باسـتخدام `json.Unmarshal` أيضًا كما سبق وفعلنا مع الخرائط. يمكننا باستخدام البـنى الاسـتغناء عن تعقيـدات توكيـد النوع التي عانينا منها مع الخرائط، من خلال تعريف أنواع البيانات في حقول البنية لتحديد أنواع بيانـات جسـون التي يجري فك تنظيمها. هذا ما سنتحدث عنه في القسم التالي.

37.5 تحليل بيانات جسون باستخدام البنى

عند قراءة بيانات جسون، هناك فرصة جيدة لمعرفة أنواع البيانات الـتي نتلقاهـا من خلال اسـتخدام البُـنى؛ فمن خلال استخدام البُنى، يمكننا منح مُفسّـر اللغـة تلميحـات تُسـاعده في تحديـد شـكل ونـوع البيانـات الـتي يتوقعها. عرّفنا في المثال السابق البنيتين `myJSON` و `myObject` وأضفنا وسوم `json` لتحديد أسـماء الحقـول بعد تحويلها إلى جسون. يمكننا الآن استخدام قيم البنية `struct` نفسها لفك ترميز سلسلة جسون المُستخدمة، وهذا ما قد يكون مفيدًا لتقليل التعليمات البرمجية المكررة في البرنامج عند تنظم أو فـك تنظيم بيانـات جسـون نفسها. فائدة أخرى لاستخدام بنية في فك تنظيم بيانات جسون هي إمكانية إخبار المُفسّر بنوع بيانات كل حقل، وهناك فائدة أخرى تأتي من استخدام مُفسّر اللغة للتحقـق من اسـتخدام الأسـماء الصـحيحة للحقـول، وبالتـالي تجنب أخطاء قد تحدث في أسماء الحقول (من النوع `string`) عند استخدام الخرائط.

لنفتح ملـف main.go، ونعـدّل تصـريح المتغـير `data` لاسـتخدام مرجـع للبنيـة `myJSON` ونضـيف بعض تعليمات الطباعة `fmt.Printf` لإظهار بيانات الحقول المختلفة في `myJSON`:

```
...
func main() {
  ...
  var data *myJSON
  err := json.Unmarshal([]byte(jsonData), &data)
  if err != nil {
        fmt.Printf("could not unmarshal json: %s\n", err)
        return
```

```
    }
    fmt.Printf("json struct: %#v\n", data)
    fmt.Printf("dateValue: %#v\n", data.DateValue)
    fmt.Printf("objectValue: %#v\n", data.ObjectValue)
}
```

نظرًا لأننا عرّفنا سابقًا أنواع البُنى، فلن نحتاج إلا إلى تحديث نوع الحقل `data` لدعم عملية فـك التنظيم في بنية. تُظهر بقية التحديثات بعض البيانـات الموجـودة في البنيـة نفسـها. لنُشـغّل ملـف البرنـامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

ويظهر لدينا الخرج التالي:

```
json struct: &main.myJSON{IntValue:1234, BoolValue:true,
StringValue:"hello!", DateValue:time.Date(2022, time.March, 2, 9, 10,
0, 0, time.UTC), ObjectValue:(*main.myObject)(0x1400011c180),
NullStringValue:(*string)(nil), NullIntValue:(*int)(nil),
EmptyString:""}
dateValue: time.Date(2022, time.March, 2, 9, 10, 0, 0, time.UTC)
objectValue: &main.myObject{ArrayValue:[]int{1, 2, 3, 4}}
```

هنـاك شـيئان يجب أن نشـير لهمـا في الخـرج السـابق، إذ نلاحـظ أولًا في سـطر `json struct` وسـطر `dateValue`، أن قيمة التاريخ والوقت من بيانات جسون جرى تحويلها إلى قيمة من النوع `time.Time` (يظهر النوع `time.Date` عند استخدام العنصر النـائب `%#v`). بمـا أن مُفسّـر جـو كـان قـادرًا على التعـرّف على النـوع `time.Time` في حقل `DateValue`، فهو قادر أيضًا على تحليل قيم النوع `string`.

الشيء الثاني الذي نلاحظه هو أن `EmptyString` يظهر على سطر `json struct` على الـرغم من أنـه لم يُضمّن في بيانات جسون الأصلية. إذا جـرى تضـمين حقـل في بنيـة مُسـتخدمة في عمليـة فـك تنظيم بيانـات جسون، وكان هذا الحقل غير موجود في بيانات جسون الأصـلية، فإنـه هـذا الحقـل يُضـبط بالقيمـة الافتراضـية لنوعه ويجري تجاهله. يمكننا بهذه الطريقة تعريف جميع الحقول المحتملة التي قد تحتوي عليها بيانـات جسـون بأمان، دون القلق بشأن حدوث خطأ إذا لم يكن الحقل موجودًا في أي من جانبي العملية.

ضُـبط كـل من الحقلين `NullStringValue` و `NullIntValue` على قيمتهمـا الافتراضـية `nil`، لأن بيانات جسون تقول أن قيمهما `null`، لكن حتى لو لم يكونا ضمن بيانات جسون، سـيأخذان نفس القيمـة. على غرار الطريقة التي تجاهلت بها الدالة `json.Unmarshal` حقل `EmptyString` في البنية `struct` عندما كان حقل `emptyString` مفقودًا من بيانات جسون، فـإن العكس هـو الصـحيح أيضًـا؛ فـإذا كـان هنـاك حقـل في بيانات جسون ليس له ما يقابله في البنية `struct`، سـيتجاهل مفسّـر اللغـة هـذا الحقـل، وينتقـل إلى الحقـل

التالي لتحليله. بالتالي إذا كانت بيانات جسون التي نقرأها كبيرة جدًا وكان البرنامج يحتاج عـددًا صـغيرًا من تلـك الحقول، يمكننا إنشاء بنية تتضمن الحقول التي نحتاجها فقط، إذ يتجاهـل مُفسّـر اللغـة أيـة حقـول من بيانـات جسون غير موجودة في البنية.

لنفتح ملف main.go ونعدّل `jsonData` لتضمين حقل غير موجود في `myJSON`:

```
...
func main() {
  jsonData := `
        {
              "intValue":1234,
              "boolValue":true,
              "stringValue":"hello!",
              "dateValue":"2022-03-02T09:10:00Z",
              "objectValue":{
                    "arrayValue":[1,2,3,4]
              },
              "nullStringValue":null,
              "nullIntValue":null,
              "extraValue":4321
        }
  `
...
}
```

لنُشغّل ملف البرنامج main.go من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
json struct: &main.myJSON{IntValue:1234, BoolValue:true,
StringValue:"hello!", DateValue:time.Date(2022, time.March, 2, 9, 10,
0, 0, time.UTC), ObjectValue:(*main.myObject)(0x14000126180),
NullStringValue:(*string)(nil), NullIntValue:(*int)(nil),
EmptyString:""}
dateValue: time.Date(2022, time.March, 2, 9, 10, 0, 0, time.UTC)
objectValue: &main.myObject{ArrayValue:[]int{1, 2, 3, 4}}
```

نلاحظ عدم ظهور الحقل الجديد `extraValue` الذي أضفناه إلى بيانات جسون ضمن الخرج، إذ تجاهله مُفسّر اللغة، لأنّه غير موجود ضمن البنية `myJSON`.

استخدمنا في هذا الفصل أنواع البُنى `struct` المُعرّفة مسبقًا في عملية فك تنظيم بيانات جسون. رأينا كيف أن ذلك يُمكّن مُفسّر اللغة من تحليل قيم الأنواع المعقدة مثل `time.Time`، ويسمح بتجاهل الحقل `EmptyString` الموجود ضمن البنية وغير موجود ضمن بيانات جسون. رأينا أيضًا كيف يمكن التحكم في الحقول التي نستخرجها من بيانات جسون وتحديد ما نريده من حقول بدقة.

37.6 الخاتمة

أنشأنا في هذا الفصل برنامجًا يستخدم الحزمة `encoding/json` من مكتبة لغة جو القياسية. استخدمنا بدايةً الدالة `json.Marshal` مع النوع `map[string]interface{}` لإنشاء بيانات جسون بطريقة مرنة. عدّلنا بعد ذلك البرنامج لاستخدام النوع `struct` مع وسوم `json` لإنشاء بيانات جسون بطريقة متسقة وموثوقة باستخدام الدالة `json.Marshal`. استخدمنا بعد ذلك الدالة `json.Unmarshal` مع النوع `map[string]interface{}` لفك ترميز سلسلة جسون وتحويلها إلى بيانات يمكن التعامل معها في برنامج جو. أخيرًا، استخدمنا نوع بنية `struct` مُعرّف مسبقًا في عملية فك تنظيم بيانات جسون باستخدام دالة `json.Unmarshal` للسماح لمُفسّر اللغة بإجراء التحليل واستنتاج أنواع البيانات وفقًا لحقول البنية التي عرّفناها.

يمكننا من خلال الحزمة `encoding/json` التفاعل مع العديد من واجهات برمجة التطبيقات APIs المتاحة على الإنترنت لإنشاء عمليات متكاملة مع مواقع الويب الأخرى. يمكننا أيضًا تحويل بيانات جو في برامجنا إلى تنسيق يمكن حفظه ثم تحميله لاحقًا للمتابعة من حيث توقف البرنامج (لأن عملية السلسلة Serialization تحفظ البيانات قيد التشغيل في الذاكرة). تتضمن الحزمة `encoding/json` أيضًا دوالًا أخرى مُفيدة للتعامل مع بيانات جسون، مثل الدالة `json.MarshalIndent` التي تساعدنا في عرض بيانات جسون بطريقة مُرتبة وأكثر وضوحًا للاطلاع عليها ومساعدتنا في استكشاف الأخطاء وإصلاحها.

دورة تطوير التطبيقات
باستخدام لغة بايثون

احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن

38. كيفية إنشاء خادم HTTP

يكرّس العديد من المطورين جزءًا من وقتهم لبناء خوادم تُسهّل توزيع المحتوى عبر الإنترنت. يُعد بروتوكول النقل التشعبي Hypertext Transfer Protocol أو اختصارًا HTTP من أهم الوسائل المستخدمة لتوزيع المحتوى مهما كان نوع البيانات عبر الإنترنت. تتضمّن مكتبة لغة جو القياسية وظائفًا مدمجة لإنشاء خادم HTTP لتخديّم محتوى الويب،أو إنشاء طلبات HTTP للتواصل مع هذه الخوادم.

سنتعلم في هذا الفصل كيفية إنشاء خادم HTTP باستخدام مكتبة لغة جو القياسية، وكيفية توسيع وظائف الخادم لاستخراج البيانات من أجزاء مختلفة من طلب HTTP، مثل سلسلة الاستعلام والمتن Body وبيانات النموذج في الطلب. سنتعلّم أيضًا كيفية تعديل استجابة الخادم عن طريق إضافة ترويسات HTTP ورموز حالة مخصصة status codes، وبالتالي السماح للمطورين بتخصيص سلوك الخادم الخاص بهم.

38.1 توضيح المصطلحات المتعلقة بخادم HTTP

في سياق إنشاء خادم HTTP باستخدام مكتبة جو القياسية، تشير المصطلحات "سلسلة استعلام الطلب" و"المتن" و"بيانات النموذج" إلى أجزاء مختلفة من طلب HTTP.

1. **سلسلة الاستعلام Query String**: سلسلة الاستعلام هي جزء من عنوان URL الذي يأتي بعد رمز علامة الاستفهام (?). يحتوي عادةً على أزواج ذات قيمة مفتاح مفصولة بعلامات &.
2. **المتن أو النص الأساس Request Body**: يحتوي متن طلب HTTP على البيانات التي يرسلها العميل إلى الخادم، مثل JSON أو XML أو النص العادي. يُستخدم بكثرة في طلبات من نوع POST و PUT و PATCH لإرسال البيانات إلى الخادم.

3. **بيانات النموذج Form Data**: تُرسل بيانات النموذج مثل جزء من طلب `POST` مع ضبط ترويسة "نوع المحتوى" على `application/x-www-form-urlencoded` أو `multipart/form-data`. وهو يتألف من أزواج مفتاح - قيمة على غرار سلسلة الاستعلام ولكن يُرسل في متن الطلب.

38.2 بروتوكول HTTP

هو بروتوكول يعمل على مستوى التطبيقات، ويستخدم لنقل مستندات الوسائط التشعبية، مثل صفحات HTML عبر الإنترنت. يعمل HTTP بمثابة أساس لاتصالات البيانات على شبكة الويب العالمية.

يتبع HTTP نموذج خادم-العميل، إذ يرسل العميل (عادةً متصفح ويب) طلبًا إلى الخادم، ويستجيب الخادم بالمعلومات المطلوبة. كما تُستخدم بعض الدوال، مثل `GET` و `POST` و `PUT` و `DELETE` من أجل تسهيل عملية الاتصال بين العميل والخادم.

ويمكنك مطالعة مقال مدخل إلى HTTP على أكاديمية حسوب لمزيد من المعلومات عن هذا البروتوكول.

38.3 المتطلبات الأولية

لمتابعة هذا الفصل التعليمي، سنحتاج إلى:

- إصدار مُثبّت من جو 1.16 أو أعلى، ارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبت لغة جو Go وقم بإعداد بيئة تطوير محلية بحسب نظام تشغيلك.
- القدرة على استخدام أداة curl لإجراء طلبات ويب.
- إلمام بكيفية استخدام جسون JSON في لغة جو.
- معرفة بكيفية استخدام السياقات Contexts في لغة جو Go.
- فهم لتنظيمات جو goroutines والقنوات channels. يمكنك الاطلاع على فصل تشغيل عدة دوال على التساير في لغة جو Go.
- الإلمام بكيفية إنشاء طلبات HTTP وإرسالها (موصى به).

38.4 إعداد المشروع

تتوفّر معظم وظائف HTTP التي تسمح لنا بإجراء الطلبات من خلال حزمة net/http الموجودة في المكتبة القياسية في لغة جو، بينما تتولى حزمة net بقية عمليات الاتصال بالشبكة. توفّر حزمة `net/http` أيضًا خادم HTTP يمكن استخدامه لمعالجة تلك الطلبات.

سننشئ في هذا القسم برنامجًا يستخدم الدالة http.ListenAndServe لتشغيل خادم HTTP يستجيب للطلبات ذات المسارات / و /hello، ثم سنوسّع البرنامج لتشغيل خوادم HTTP متعددة في نفس البرنامج.

كما هو معتاد، سنحتاج لبدء إنشاء برامجنا إلى إنشاء مجلد للعمل ووضع الملفات فيه، ويمكن وضع المجلد في أي مكان على الحاسب، إذ يكون للعديد من المبرمجين عادةً مجلدٌ يضعون داخله كافة مشاريعهم. سنستخدم في هذا الفصل مجلدًا باسم projects، لذا فلننشئ هذا المجلد وننتقل إليه:

```
$ mkdir projects
$ cd projects
```

الآن من داخل هذا المجلد، سنشغّل الأمر mkdir لإنشاء مجلد باسم httpserver ثم سنستخدم cd للانتقال إليه:

```
$ mkdir httpserver
$ cd httpserver
```

الآن، بعد أن أنشأنا مجلدًا للبرنامج وانتقلنا إليه، يمكننا البدء في تحقيق خادم HTTP.

38.5 الاستماع إلى الطلبات وتقديم الردود

يتضمّن مخدّم HTTP في لغة جو مكونين رئيسيين: المخدّم الذي يستمع إلى الطلبات القادمة من العميل الذي يرسل طلبات HTTP (عميل HTTP أو عملاء HTTP) ومُعالج طلبات (أو أكثر) يستجيب لتلك الطلبات. سنبدأ حاليًا في استخدام الدالة http.HandleFunc التي تخبر المُخدّم بالدالة التي يجب استدعاؤها لمعالجة الطلب، ثم سنستخدم الدالة http.ListenAndServe لتشغيل الخادم وإخباره بالتحضير للاستماع إلى طلب HTTP جديد وتخديمه من خلال معالجته بدوال المعالجة Handler functions التي نُنشئها مُسبقًا.

بما أننا الآن داخل مجلد httpserver، يمكننا فتح ملف main.go باستخدام محرر نانو nano أو أي محرر آخر تريده:

```
$ nano main.go
```

سننشئ داخل هذا الملف دالتين getRoot و getHello سيمثلان دوال المعالجة الخاصة بنا، ثم سننشئ دالةً رئيسية main() لاستخدامها في إعداد معالجات الطلبات من خلال الدالة http.HandleFunc، وذلك بتمرير المسار / الخاص بالدالة getRoot والمسار /hello الخاص بالدالة getHello. بمجرد أن ننتهي من إعداد دوال المعالجة الخاصة بنا، يمكننا استدعاء http.ListenAndServe لبدء تشغيل المخدّم والاستماع للطلبات. دعنا نضيف التعليمات البرمجية التالية إلى الملف لبدء تشغيل البرنامج وإعداد المعالجات:

```
package main
```

```
import (
  "errors"
  "fmt"
  "io"
  "net/http"
  "os"
)
func getRoot(w http.ResponseWriter, r *http.Request) {
  fmt.Printf("got / request\n")
  io.WriteString(w, "This is my website!\n")
}
func getHello(w http.ResponseWriter, r *http.Request) {
  fmt.Printf("got /hello request\n")
  io.WriteString(w, "Hello, HTTP!\n")
}
```

أعـددنا في البدايـة الحزمـة الخاصـة بالبرنـامج `package` مـع اسـتيراد الحـزم المطلوبـة من خلال التعليمـة `import`، كما أنشأنا الدالتين `getRoot` و `getHello`، ونلاحـظ أن لهمـا نفس البصـمة Signature، إذ تقبلان نفس الوسـيطين وهمـا: قيمـة `http.ResponseWriter` وقيمـة `*http.Request`. هاتـان الـدالتان لهمـا بصمة تطابق النوع http.HandlerFunc، والـذي يشـيع اسـتخدامه لتعريـف دوال معالجـة HTTP. عنـد تقـديم طلب إلى الخادم، فإنه يُزوِّد هاتين القيمتين بمعلومـات حـول الطلب الحـالي، ثم يسـتدعي دالـة المعـالج الـتي تتوافق مع هذه القيم.

تُستخدم القيمة http.ResponseWriter (المُسماة `w`) ضـمن `http.HandlerFunc` للتحكّم بمعلومـات الاستجابة التي يُعاد كتابتها إلى العميـل الـذي قـدّم الطلب، مثـل متن الاسـتجابة response body (جـزء من استجابة HTTP ويحمل الحمولة الفعلية أو المعلومات التي طلبها العميل أو التي يوفرها الخـادم) أو رمـوز الحالـة status codes، وهي معلومات حول نتيجة الطلب والحالة الحالية للخادم، وهي جزء من بروتوكول HTTP يجري تضمينها في ترويسة الاستجابة، وتشير إلى ما إذا كان الطلب ناجحًا أو واجه خطـأً أو يتطلب إجـراءً إضـافيًا. بعـد ذلك، تُستخدم القيمة http.Request (المسماة `r`) للحصول على معلومات حول الطلب الذي جـاء إلى الخـادم، مثل المتن المُرسل في حالة طلب `POST` أو معلومات حول العميل الذي أجرى الطلب.

في كل من معالجات HTTP التي أنشأناها، يمكننا اسـتخدام الدالـة `fmt.Printf` للطباعـة، وذلـك عنـدما يـأتي طلب لدالـة المعالجـة، ثم نسـتخدم `http.ResponseWriter` لإرسـال نص مـا إلى متن الاسـتجابة. `http.ResponseWriter` هي واجهة في حزمة `http` تمثل الاستجابة التي سترسـل مـرةً أخـرى إلى العميـل عنـد تقـديم طلب إلى الخـادم، وهي io.Writer ممـا يعـني أنهـا تـوفر إمكانيـة كتابـة البيانـات. نسـتخدم

`http.ResponseWriter` في الشيفرة السابقة مثل وسيط `w` في دوال المعالجة `getRoot` و `getHello`، للسماح بالتحكم في الرد المرسل إلى العميل، وبالتالي إمكانية كتابة متن الاستجابة، أو ضبط الترويسات headers، أو تحديد رمز الحالة باستخدامها. نستخدم الدالة io.WriteString لكتابة الاستجابة ضمن متن الرسالة.

لنضف الآن الدالة `main` إلى الشيفرة السابقة:

```
...
func main() {
  http.HandleFunc("/", getRoot)
  http.HandleFunc("/hello", getHello)
  err := http.ListenAndServe(":3333", nil)
...
```

تكون الدالة الرئيسية `main` في جزء الشيفرة السابق، مسؤولةً عن إعداد خادم HTTP وتحديد معالجات الطلب. هناك استدعاءان إلى الدالة `http.HandleFunc`، بحيث يربط كل استدعاء لها دالة معالجة من أجل مسار طلب محدد ضمن مجمّع الخادم الافتراضي default server multiplexer. يتطلب الأمر وسيطين: الأول هو مسار الطلب (في هذه الحالة `/` ثم `/hello`) ودالة المعالجة (`getRoot` ثم `getHello` على التوالي). الدالتان `getRoot` و `getHello` هما دوال المعالجة التي سيجري استدعاؤها عند تقديم طلب إلى المسارات المقابلة (`/` و `/hello`). للدالتين توقيع مماثل للدالة `http.HandlerFunc` التي تقبل `http.ResponseWriter` و `*http.Request` كوسطاء.

تُستخدم الدالة `http.ListenAndServe` لبدء تشغيل خادم HTTP للاستماع إلى الطلبات الواردة. يتطلب الأمر وسيطين، هما: عنوان الشبكة للاستماع عليه (في هذه الحالة `:3333`) ومعالج اختياري `http.Handler`. يحدد `:3333` في برنامجنا أن الخادم يجب أن يستمع إلى المنفذ `3333`، ونظرًا لعدم تحديد عنوان IP، سيستمع إلى جميع عناوين IP المرتبطة بالحاسب. يمثّل منفذ الشبكة network port طريقةً تمكّن جهاز الحاسوب من أن يكون لديه عدة برامج تتواصل مع بعضها بنفس الوقت، بحيث يستخدم كل برنامج منفذه المخصص، وبالتالي عند اتصال العميل مع منفذ معين يعلم الحاسوب إلى أي منفذ سيُرسل. وإذا كنت تريد قصر الاتصالات على المضيف المحلي `localhost` فقط، فيمكنك استخدام `127.0.0.1:3333`.

تمرّر الدالة `http.ListenAndServe` قيمة `nil` من أجل المعامل `http.Handler`، وهذا يخبر دالة `ListenAndServe` بأنك تريد استخدام مجمّع الخادم الافتراضي وليس أي مجمّع ضبطه سابقًا.

الدالة `ListenAndServe` هي استدعاء "حظر"، مما يعني أنها ستمنع تنفيذ التعليمات البرمجية الأخرى حتى يُغلق الخادم. تُعيد هذه الدالة خطًأ إذا فشلت عملية بدء تشغيل الخادم أو إذا حدث خطأ أثناء التشغيل. من المهم تضمين عملية معالجة الأخطاء بعد استدعاء `http.ListenAndServe`، وذلك لأن الدالة يمكن أن

تفشل، بالتالي من الضروري التعامل مع الأخطاء المحتملة. يُستخدم المتغير `err` لالتقاط أي خطأ يُعاد بواسطة `ListenAndServe`. يمكننا إضافة شيفرة معالجة الأخطاء بعد هذا السطر لمعالجة أي أخطاء محتملة قد تحدث أثناء بدء تشغيل الخادم أو تشغيله كما سنرى.

لنضف الآن شيفرة معالجة الأخطاء إلى دالة `ListenAndServe` ضمن دالة `main` الرئيسية كما يلي:

```
...
func main() {
  ...
  err := http.ListenAndServe(":3333", nil)
 if errors.Is(err, http.ErrServerClosed) {
        fmt.Printf("server closed\n")
  } else if err != nil {
        fmt.Printf("error starting server: %s\n", err)
        os.Exit(1)
  <^>}
}
```

بعد استدعاء `http.ListenAndServe`، يجري تخزين الخطأ المُعاد في المتغير `err`. تجري عملية فحص الخطأ الأولى باستخدام `errors.Is(err, http.ErrServerClosed)`، إذ يجري التحقق ما إذا كان الخطأ هو `http.ErrServerClosed`، والذي يُعاد عندما يُغلق الخادم أو يجري إيقاف تشغيله. يعني ظهور هذا الخطأ أن الخادم قد أُغلق بطريقة متوقعة، بالتالي طباعة الرسالة "server closed".

يُنجز فحص الخطأ الثاني باستخدام `Err != nil`. يتحقق هذا الشرط مما إذا كان الخطأ ليس `nil`، مما يشير إلى حدوث خطأ أثناء بدء تشغيل الخادم أو تشغيله. إذا تحقق الشرط، فهذا يعني حدوث خطأ غير متوقع، وبالتالي طباعة رسالة خطأ مع تفاصيل الخطأ باستخدام `fmt.Printf`، كما يجري إنهاء البرنامج مع شيفرة الخطأ `1` باستخدام `os.Exit (1)` للإشارة إلى حدوث خطأ.

تجدر الإشارة إلى أن أحد الأخطاء الشائعة التي قد تواجهها هو أن العنوان قيد الاستخدام فعلًا `address already in use`. يحدث هذا عندما تكون الدالة `ListenAndServe` غير قادرة على الاستماع إلى العنوان أو المنفذ المحدد للخادم لأنه قيد الاستخدام فعلًا من قبل برنامج آخر، أي إذا كان المنفذ شائع الاستخدام أو إذا كان برنامج آخر يستخدم نفس العنوان أو المنفذ.

> عند ظهور هذا الخطأ، يجب التأكد من إيقاف أي مثيلات سابقة للبرنامج ثم محاولة تشغيله مرة أخرى. إذا استمر الخطأ يجب علينا محاولة استخدام رقم منفذ مختلف لتجنب التعارضات، فمن المحتمل أن برنامجًا آخر يستخدم المنفذ المحدد. يمكن اختيار رقم منفذ مختلف (أعلى من 1024 وأقل من 65535) وتعديل الشيفرة وفقًا لذلك.

على عكس برامج لغة جو الأخرى؛ لن يُنهى البرنامج فورًا من تلقاء نفسه. لنُشـغّل ملـف البرنـامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

بما أن البرنامج يبقى قيد التشغيل في الطرفية الحالية، فسنحتاج إلى فتح نافذة أخرى للطرفية لكي نتفاعـل مع الخادم (ستظهر الأوامر بلون مختلف عن الطرفية الأولى).

ضمن الطرفية الثانية التي فتحناها، نستخدم الأمر `curl` لتقديم طلب HTTP إلى خادم HTTP الخـاص بنـا. `curl` هي أداة مُثبّتة افتراضيًا على العديد من الأنظمة التي يمكنها تقـديم طلبـات للخـوادم من أنـواع مختلفـة، وسنستخدمها في هذا الفصل لإجراء طلبـات HTTP. يسـتمع الخـادم إلى الاتصـالات على المنفـذ `3333` لجهـاز الحاسوب، لذا يجب تقديم الطلب للمضيف المحلي على نفس المنفذ:

```
$ curl http://localhost:3333
```

سيكون الخرج على النحو التالي:

```
This is my website!
```

العبارة `This is my website` ناتجة عن الدالة `getRoot`، وذلك لأنك استخدمت المسار `/` على خادم HTTP. دعونا الآن نستخدم المسار `/hello` على نفس المضيف والمنفذ، وذلك بإضافة المسـار إلى نهايـة أمـر `curl`:

```
$ curl http://localhost:3333/hello
```

ليكون الخرج هذه المرة:

```
Hello, HTTP!
```

نلاحظ أن الخرج السابق كان نتيجةً لاستدعاء الدالة `getHello`. إذا عدنا إلى المحطـة الطرفيـة الأولى الـتي يعمـل عليهـا خـادم HTTP، نلاحـظ وجـود سـطرين أنتجهمـا الخـادم الخـاص بنـا: واحـد للطلب `/` والآخـر للطلب `/hello`.

```
got / request
got /hello request
```

سيستمر البرنامج بالعمل، لذا يجب علينا إيقافه يدويًا من خلال الضغط على المفتاحين Ctrl+C.

لقد أنشأنا برنامجًا يمثل خادم HTTP، لكنه يستخدم مجمّع خـادم افتراضـي وخـادم HTTP افتراضـي أيضًـا. يمكن أن يؤدي الاعتماد على القيم الافتراضية أو العامة Global إلى حـدوث أخطـاء يصـعب تكرارهـا أو يصـعب

إنتاجها باستمرار Reproduce consistently، إذ يمكن أن تُعدِّل أجزاءً مختلفة من البرنامج هذه القيم العامة في أوقات مختلفة، مما يؤدي إلى حالة غير صحيحة أو غير متسقة. يصبح تحديد مثل هذه الأخطاء أمرًا صعبًا لأنها قد تحدث فقط في ظل ظروف معينة أو إذا جرى استدعاء وظائف معينة بترتيب معين.

38.6 معالجات طلبات التجميع

عند بدء تشغيل خادم HTTP سابقًا؛ استخدمنا مجمّع خادم افتراضي عن طريق تمرير قيمة صفرية (أي nil) للمعامل http.Handler في دالة ListenAndServe. رأينا أيضًا أن هناك بعض المشاكل التي قد تطرأ في حالة استخدام المعاملات الافتراضية. بما أن http.Handler هو واجهة interface، فهذا يعني أنه لدينا الخيار لإنشاء بنية مخصصة تحقق هذه الواجهة. هناك طبعًا حالات نحتاج فيها فقط إلى http.Handler الافتراضي الذي يستدعي دالة واحدة لمسار طلب معين، أي كما في حالة مجمّع الخادم الافتراضي، لكن هناك حالات قد تتطلّب أكثر من ذلك؛ هذا ما نناقشه تاليًا.

لنعدّل البرنامج الآن لاستخدام http.ServeMux، إنها أداة تعمل مثل مجمّع للخادم، وهي مسؤولة عن التوجيه والتعامل مع طلبات HTTP الواردة بناءً على مساراتها. تُحقّق http.ServeMux الواجهة http.Handler المؤمَّنة من قِبل حزمة net/http، مما يعني قدرتها على التعامل مع طلبات HTTP وإنشاء الاستجابات المناسبة، بالتالي مزيد من التحكم في التوجيه والتعامل مع مسارات الطلبات المختلفة، وإتاحة الفرصة لتحديد دوال أو معالجات محددة لكل مسار. بالتالي نكون قد اتبعنا نهجًا أكثر تنظيمًا وقابلية للتخصيص للتعامل مع طلبات HTTP في البرنامج.

يمكن تهيئة بنية http.ServeMux بطريقة مشابهة للمجمّع الافتراضي، لذا لن نحتاج إلى إجراء العديد من التغييرات على البرنامج لبدء استخدام مجمّع الخادم المخصّص بدلًا من الافتراضي. لتحديث البرنامج وفقًا لذلك، نفتح ملف main.go مرةً أخرى ونجري التعديلات اللازمة لاستخدام http.ServeMux:

```
...
func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", getRoot)
  mux.HandleFunc("/hello", getHello)
  err := http.ListenAndServe(":3333", mux)
  ...
}
```

أنشأنا http.ServeMux جديد باستخدام باني http.NewServeMux وأسندناه إلى المتغير mux، ثم عدّلنا استدعاءات http.HandleFunc لاستخدام المتغير mux بدلًا من استدعاء حزمة http مباشرة. أخيرًا، عدّلنا استدعاء http.ListenAndServe لتزويده بالمعالج http.Handler الذي أنشأناه mux بدلًا من nil.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

سيستمر البرنامج في العمل كما في المرة السابقة، لذا نحتاج إلى تشغيل أوامر في طرفية أخرى للتفاعل مع الخادم. أولًا، نستخدم `curl` لطلب المسار `/`:

```
$ curl http://localhost:3333
```

سيكون الخرج على النحو التالي:

```
This is my website!
```

الخرج كما هو في المرة السابقة. دعونا الآن نستخدم المسار `/hello` على نفس المضيف والمنفذ، وذلك بإضافة المسار إلى نهاية أمر `curl`:

```
$ curl http://localhost:3333/hello
```

سيكون الخرج كما يلي:

```
Hello, HTTP!
```

نلاحظ أن الخرج السابق كان كما المرة السابقة أيضًا.

إذا عدنا الآن إلى الطرفية الأولى، فسنرى مخرجات كل من `/` و `hello /` كما كان من قبل:

```
got / request
got /hello request
```

لاحظ أن هذه التعديلات لا تغيّر وظيفة البرنامج بل تبدل فقط مجمّع الخادم الافتراضي بآخر مخصص.

سيستمر البرنامج بالعمل، لذا يجب علينا إيقافه يدويًا من خلال الضغط على المفتاحين Ctrl+C.

38.7 تشغيل عدة خوادم في وقت واحد

سنجري خلال هذا القسم تعديلات على البرنامج لاستخدام عدة خوادم HTTP في نفس الوقت باستخدام http.Server التي توفرها حزمة `net/http`. يمكننا بالحالة الافتراضية تشغيل خادم HTTP واحد فقط في البرنامج، لكن قد تكون هناك سيناريوهات نحتاج فيها إلى تخصيص سلوك الخادم أو تشغيل عدة خوادم في نفس الوقت، مثل استضافة موقع ويب عام Public website، وموقع ويب إداري خاص Private admin website داخل نفس البرنامج. لأجل ذلك سنعدّل ملف main.go لإنشاء نسخ متعددة من `http.Server`

لأغراض مختلفة، إذ سيكون لكل خادم التهيئة والإعدادات الخاصة به. يتيح لك هذا مزيدًا من التحكم في سلوك الخادم ويمكّنك من التعامل مع وظائف خادم متعددة داخل نفس البرنامج.

سنعدّل أيضًا دوال المعالجة لتحقيق إمكانية الوصول إلى Context.Context المرتبط مع *http.Request؛ أي إمكانية الوصول إلى سياق الطلبات الواردة، إذ يمكننا من خلال هذا السياق تمييز الخادم الذي يأتي الطلب منه. إذًا من خلال تخزين هذه المعلومات في متغير السياق، يصبح بمقدورنا استخدامها داخل دالة المعالجة لتنفيذ إجراءات محددة أو تخصيص الاستجابة بناءً على الخادم الذي أنشأ الطلب.

لنفتح ملف main.go ونعدّله بالتالي:

```
package main
import (
// os لاحظ أننا حذفنا استيراد
  "context"
  "errors"
  "fmt"
  "io"
  "net"
  "net/http"
)
const keyServerAddr = "serverAddr"
func getRoot(w http.ResponseWriter, r *http.Request) {
  ctx := r.Context()
  fmt.Printf("%s: got / request\n", ctx.Value(keyServerAddr))
  io.WriteString(w, "This is my website!\n")
}
func getHello(w http.ResponseWriter, r *http.Request) {
  ctx := r.Context()

  fmt.Printf("%s: got /hello request\n", ctx.Value(keyServerAddr))
  io.WriteString(w, "Hello, HTTP!\n")
}
```

عدّلنا بيان الاستيراد import لتضمين الحزم المطلوبة، ثم أنشأنا سلسلة نصية ثابتة const string تسمى keyServerAddr لتعمل مثل مفتاح لقيمة عنوان خادم HTTP في سياق http.Request، ثم عدّلنا دوال المعالجة getRoot و getHello للوصول إلى قيمة Context.Context التابعة إلى

`http.Request`. بعد الحصول على القيمة يمكننا تضمين عنوان خـادم HTTP في خـرج `fmt.Printf` حـتى نتمكن من معرفة أي من الخادمين تعامل مع طلب HTTP.

لنعدّل الآن الدالة `main` بإضافة قيمتي `http.Server`:

```
...
func main() {
  ...
  mux.HandleFunc("/hello", getHello)
  ctx, cancelCtx := context.WithCancel(context.Background())
  serverOne := &http.Server{
        Addr:  ":3333",
        Handler: mux,
        BaseContext: func(l net.Listener) context.Context {
              ctx = context.WithValue(ctx, keyServerAddr,
l.Addr().String())
              return ctx
        },
  }
```

التغيـير الأول الـذي أجرينـاه هـو إنشـاء قيمـة `Context.Context` جديـدة مـع دالـة متاحـة هي الدالـة `cancelCtx`. هذا يسمح لنا بإلغاء السياق عند الحاجة.

عرّفنا أيضًا نسخةً تسمى `serverOne` من `http.Server`، وهو مشابه لخادم HTTP الذي كنا نسـتخدمه، ولكن بـدلًا من تمريـر العنـوان والمعـالج مباشـرةً إلى `http.ListenAndServe`، يمكن إسـنادهما مثـل قيم `Addr` و `Handler` في بنية `http.Server`.

التعديل الآخر كان بإضافة دالة `BaseContext`، وهي دالة تسمح بتعديل أجـزاء من `Context.Context` الذي جرى تمريره إلى دوال المعالجة عند استدعاء التابع `Context` من `*http.Request`. أضفنا في الشـيفرة السابقة عنوان الاستماع الخاص بالخادم إلى السياق باستخدام المفتاح `serverAddr`، وهـذا يعـني أن العنـوان الذي يستمع فيه الخادم للطلبـات الـواردة مرتبـط بمفتـاح `serverAddr` في السـياق. عنـدما نسـتدعي الدالـة `BaseContext`، فإنها تتلقى `net.Listener`، والذي يمثل مستمع الشبكة الأساسي الذي يستخدمه الخادم.

بالنسبة لبرنامجنا فإننا من خلال استدعاء `l.Addr(). String()`، نكون قد حصـلنا على عنـوان شـبكة المستمع. يتضمن هذا عادةً عنوان IP ورقم المنفذ الذي يستمع الخـادم عليـه. بعـد ذلـك نضـيف العنـوان الـذي حصلنا عليه إلى السياق باستخدام الدالة `Context.WithValue`، والتي تتيح لنا تخزين أزواج المفتاح والقيمـة في السـياق. في هـذه الحالـة يكـون المفتـاح هـو `serverAddr`، والقيمـة المرتبطـة بـه هي عنـوان الاسـتماع الخاص بالخادم.

لنعرّف الآن الخادم الثاني `serverTwo`:

```
...
func main() {
  ...
  serverOne := &http.Server {
        ...
  }
  serverTwo := &http.Server{
        Addr:  ":4444",
        Handler: mux,
        BaseContext: func(l net.Listener) context.Context {
              ctx = context.WithValue(ctx, keyServerAddr,
l.Addr().String())
              return ctx
        },
  }
```

نعرّف الخادم الثاني بنفس طريقة تعريف الأول، لكن نضع حقل العنوان على `:4444` بدلًا من `:3333` وذلك لكي يستمع الخادم الأول على للاتصالات على المنفذ 3333 ويستمع الخادم الثاني على المنفذ 4444.

لنعدّل الآن البرنامج من أجل استخدام الخادم الأول `serverOne` وجعله يعمل مثل تنظيم جو goroutine:

```
...
func main() {
  ...
  serverTwo := &http.Server {
        ...
  }
  go func() {
        err := serverOne.ListenAndServe()
        if errors.Is(err, http.ErrServerClosed) {
              fmt.Printf("server one closed\n")
        } else if err != nil {
              fmt.Printf("error listening for server one: %s\n", err)
        }
        cancelCtx()
  }()
```

نستخدم تنظيم goroutine لبدء تشغيل الخادم الأول `serverOne` باستخدام الدالـة `ListenAndServe` كما فعلنا سـابقًا، لكن هـذه المـرة دون أي معـاملات لأن قيم `http.Server` جـرى تهيئتهـا مسـبقًا باسـتخدام العنوان والمعالج المطلوبين.

تجري عملية معالجة الأخطاء كما في السابق؛ فإذا كان الخادم مغلقًا، فإنه يطبع رسـالة تشـير إلى أن الخـادم الأول أصبح مغلقًا؛ وإذا كان هناك خطأ آخر بخلاف `http.ErrServerClosed`، فستظهر رسالة خطأ.

تُستدعى أخيرًا الدالة `CancelCtx` لإلغاء السـياق الـذي قـدمناه لمُعالجـات HTTP ودوال `BaseContext` لكلا الخادمين. بالتالي إنهاء أي عمليات جارية تعتمد عليه بأمـان. هـذا يضـمن أنـه إذا انتهى الخـادم لأي سـبب، فسيُنهى أيضًا السياق المرتبط به.

لنعدّل الآن البرنامج لاستخدام الخادم الثاني، بحيث يعمل مثل تنظيم جو :

```
...
func main() {
  ...
  go func() {
        ...
  }()
  go func() {
        err := serverTwo.ListenAndServe()
        if errors.Is(err, http.ErrServerClosed) {
              fmt.Printf("server two closed\n")
        } else if err != nil {
              fmt.Printf("error listening for server two: %s\n", err)
        }
        cancelCtx()
  }()
  <-ctx.Done()
}
```

تنظيم جو هنا هو نفسه الأول من الناحية الوظيفية، فهو يُشغّل `serverTwo` فقـط بـدلًا من `serverOne`. بعد بدء تشغيل `serverTwo`، يصل التنظيم الخاص بالدالة `main` إلى السـطر `ctx.Done`. ينتظـر هـذا السـطر إشارةً من قناة `ctx.Done`، والتي تُعاد عند إلغاء السياق أو الانتهاء منه. من خلال هذا الانتظار نكـون قـد منعنـا تنظيم الدالة الرئيسـية من الخـروج من الدالـة `main` حـتى تنتهي تنظيمـات جـو لكلا الخـادمين من العمـل. إذًا، الغرض من هذا الأسلوب هو التأكد من استمرار تشغيل البرنامج حتى انتهاء كلا الخادمين أو مواجهة خطأ.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

نُشغّل الآن أوامر `curl` (في الطرفية الثانية) لطلب المسار `/` والمسار`/hello` من الخادم الذي يستمع على 3333، مثل الطلبات السابقة:

```
$ curl http://localhost:3333
$ curl http://localhost:3333/hello
```

سيكون الخرج كما في المرات السابقة:

```
This is my website!
Hello, HTTP!
```

لنشغّل الأوامر نفسها مرة أخرى، ولكن هذه المرة مع المنفذ 4444 الذي يتوافق مع `serverTwo`:

```
$ curl http://localhost:4444
$ curl http://localhost:4444/hello
```

سيبقى الخرج نفسه كما في المرة السابقة أيضًا:

```
This is my website!
Hello, HTTP!
```

لنلقي نظرةً الآن على الطرفية الأولى حيث يعمل الخادم:

```
[::]:3333: got / request
[::]:3333: got /hello request
[::]:4444: got / request
[::]:4444: got /hello request
```

الخرج مشابه لما رأيناه من قبل، لكنه يعرض هذه المرة الخادم الذي استجاب للطلب. يظهِر الطلبان الأولان أنهما جاءا من الخادم الذي يستمع على المنفذ 3333 أي `serverOne`، والطلبان الثانيان جاءا من الخادم الذي يستمع على المنفذ 4444 أي `serverTwo`. إنها القيم التي جرى استردادها من قيمة `serverAddr` في `BaseContext`. قد يكون الخرج مختلفًا قليلًا عن الخرج أعلاه اعتمادًا على ما إذا كان جهاز الحاسب المُستخدم يستخدم IPv6 أم لا. إذا كان الحاسب يستخدم IPv6 فسيكون الخرج كما أعلاه، وإلا سنرى `0.0.0.0` بدلًا من `[::]`. السبب في ذلك هو أن جهاز الحاسب سيتواصل مع نفسه عبر IPv6، و `[::]` هو تدوين IPv6 والذي يُقابل `0.0.0.0` في IPv4. بعد الانتهاء نضغط على مفتاحي Ctrl+C لإنهاء الخادم.

تعرفنا في هذا القسم على عملية إنشاء برنامج خادم HTTP وتحسينه تدريجيًا للتعامل مع السيناريوهات المختلفة. بدأنا بتهيئة الخادم الافتراضي باستخدام `http.HandleFunc` و `http.ListenAndServe`. بعد

ذلك عدّلناه لاستخدام `http.ServeMux` مثل مجمّع خادم، مما يسمح لنا بالتعامل مع معالجات طلبات متعددة لمسارات مختلفة. وسّعنا البرنامج أيضًا لاستخدام `http.Server`، مما يمنحنا مزيدًا من التحكم في تهيئة الخادم ويسمح لنا بتشغيل خوادم HTTP متعددة في نفس الوقت داخل نفس البرنامج.

على الرغم من أن البرنامج يعمل، إلا أنه يفتقر إلى التفاعل الذي يتجاوز تحديد المسارات المختلفة. لمعالجة هذا القيد، يركز القسم التالي من هذا الفصل على دمج قيم سلسلة الاستعلام query string values في وظائف الخادم، مما يتيح للمستخدمين التفاعل مع الخادم باستخدام معاملات الاستعلام.

38.8 فحص سلسلة الاستعلام الخاصة بالطلب

ينصب التركيز في هذا القسم على دمج قيم سلسلة الاستعلام في وظائف خادم HTTP. سلسلة الاستعلام هي مجموعة من القيم الملحقة بنهاية عنوان URL، تبدأ بالمحرف `?` وتستخدم المُحدّد `&` للقيم الإضافية. توفر قيم سلسلة الاستعلام وسيلة للمستخدمين للتأثير على الاستجابة التي يتلقونها من خادم HTTP عن طريق تخصيص النتائج أو تصفيتها، فمثلًا قد يستخدم أحد الخوادم قيمة `results` للسماح للمستخدم بتحديد شيء مثل `results = 10` ليقول إنه يرغب في رؤية 10 عناصر في قائمة النتائج.

لتحقيق هذه الميزة نحتاج إلى تحديث دالة المعالجة `getRoot` في ملف main.go للوصول إلى قيم سلسلة الاستعلام `*http.Request` باستخدام التابع `r.URL.Query`، ثم طباعتها بعد ذلك على الخرج. نزيل أيضًا `serverTwo` وكل الشيفرات المرتبطة به من الدالة `main`، لأنها لم تعد مطلوبة للتغييرات القادمة:

```
...
func getRoot(w http.ResponseWriter, r *http.Request) {
  ctx := r.Context()
  hasFirst := r.URL.Query().Has("first")
  first := r.URL.Query().Get("first")
  hasSecond := r.URL.Query().Has("second")

  second := r.URL.Query().Get("second")
  fmt.Printf("%s: got / request. first(%t)=%s, second(%t)=%s\n",
        ctx.Value(keyServerAddr),
        hasFirst, first,
        hasSecond, second)
  io.WriteString(w, "This is my website!\n")
}
...
```

يمكننا في دالة `getRoot` المحدَّثة استخدام الحقل `r.URL` الذي يتبع إلى `*http.Request` من أجل الوصول إلى الخصائص المتعلقة بعنوان URL المطلوب. باستخدام التابع `Query` في الحقل `r.URL`، يمكننا الوصول إلى قيم سلسلة الاستعلام المرتبطة بالطلب. هناك طريقتان يمكن استخدامهما للتفاعل مع بيانات سلسلة الاستعلام:

1. يتحقق التابع `Has` ما إذا كانت سلسلة الاستعلام تحتوي على قيمة بمفتاح معين، مثل"first" أو "second". يعيد التابع قيمة بوليانية `bool` تشير إلى وجود المفتاح.
2. يسترد التابع `Get` القيمة المرتبطة بمفتاح معين من سلسلة الاستعلام وتكون من نوع `string`، وإذا لم يعثر على المفتاح، يعيد عادةً سلسلة فارغة.

نظريًّا يمكننا دائمًا استخدام `Get` لاسترداد قيم سلسلة الاستعلام لأنها ستعيد دائمًا إما القيمة الفعلية للمفتاح المحدد أو سلسلة فارغة إذا كان المفتاح غير موجود. هذا جيد في كثير من الحالات، ولكن في بعض الحالات الأخرى، قد نرغب في معرفة الفرق بين المستخدم الذي يقدم قيمة فارغة أو عدم تقديم قيمة إطلاقًا. إذًا، بناءً على حالة الاستخدام الخاصة بنا، قد يكون من المهم التمييز بين المستخدم الذي يقدم قيمة فارغة للمرشّح أو عدم تقديم مرشح `filter` إطلاقًا:

1. يُقدّم المستخدم قيمة فارغة للمرشح `filter`: هنا يحدد المستخدم صراحة قيمة المرشح، ولكنه يضبطها على قيمة فارغة. قد يشير هذا إلى اختيار متعمد لاستبعاد نتائج معينة أو تطبيق شرط مرشح`filter` معين. مثلًا، إذا كانت لدينا دالة بحث وقدم المستخدم قيمة فارغة للمرشح، فيمكننا تفسيرها على أنها "إظهار كافة النتائج".
2. لا يوفر المستخدم مرشّح إطلاقًا: هنا لم يقدّم المستخدم مرشّحًا في الطلب. هذا يعني عادةً أن المستخدم يريد استرداد جميع النتائج دون تطبيق أي تصفية محددة. يمكن عدّه سلوكًا افتراضيًا، إذ لا تُطبّق شروط ترشيح محددة.

يسمح لك استخدام `Has` و `Get` بالتمييز بين الحالات التي يقدم فيها المستخدم صراحة قيمةً فارغة والحالات التي لا تُقدّم فيها قيمة إطلاقًا. بالتالي إمكانية التعامل مع السيناريوهات المختلفة اعتمادًا على حالة الاستخدام المحددة الخاصة بنا.

يمكنك تحديث دالة `getRoot` لعرض قيم `Has` و `Get` من أجل قيمتي سلسلة الاستعلام `first` و `second`.

لنعدّل الدالة `main` بحيث نستخدم خادم واحد مرة أخرى:

```
...
func main() {
  ...
  mux.HandleFunc("/hello", getHello)
  ctx := context.Background()
  server := &http.Server{
        Addr:  ":3333",
        Handler: mux,
        BaseContext: func(l net.Listener) context.Context {
              ctx = context.WithValue(ctx, keyServerAddr,
l.Addr().String())
              return ctx
        },
  }
  err := server.ListenAndServe()
  if errors.Is(err, http.ErrServerClosed) {
        fmt.Printf("server closed\n")
  } else if err != nil {
        fmt.Printf("error listening for server: %s\n", err)
  }
}
```

نلاحظ ضمن الدالة `main()` إزالة المراجع والشيفرات المرتبطة بالخادم الثاني `serverTwo`، لأننا لم نعد بحاجة إلى خوادم متعددة. نقلنا أيضًا تنفيذ الخادم (`serverOne` سابقًا) خارج التنظيم goroutine وإلى الدالة `main()`. هذا يعني أنه سيجري بدء تشغيل الخادم بطريقة متزامنة، و ينتظر التنفيذ حتى يُغلق الخادم قبل المتابعة.

هنالك تغيير آخر باستخدام الدالة `server.ListenAndServe`؛ فبدلًا من أن نستخدم `http.ListenAndServe`، يمكننا الآن استخدام `server.ListenAndServe` لبدء الخادم. يتيح ذلك الاستفادة من تهيئة `http.Server` وأي تخصيصات أجريناها. كذلك أضفنا شيفرة لمعالجة الأخطاء، وذلك للتحقق ما إذا كان الخادم مغلقًا أو واجه أي خطأ آخر أثناء الاستماع. إذا كان الخطأ هو `http.ErrServerClosed`، فهذا يعني أن عملية الإغلاق عن قصد (إغلاق طبيعي)، وخلاف ذلك ستجري طباعة الخطأ. بإجراء هذه التغييرات سيُشغّل برنامجنا الآن خادم HTTP واحد باستخدام وفقًا لتهيئة `http.Server` ليبدأ في الاستماع للطلبات الواردة، ولن تحتاج إلى تحديثات أخرى من أجل تخصيصات للخادم مستقبلًا.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

نُشغّل الآن أوامر `curl` (في الطرفية الثانية). هنا نحتاج إلى إحاطة عنوان URL بعلامات اقتباس مفردة (')، وإلا فقد تُفسر صدفة الطرفية (أي Shell) الرمز `&` في سلسلة الاستعلام على أنه ميزة "تشغيل الأمر في الخلفية". ضمن عنوان URL نضيف `first=1` إلى `first` و `second=` إلى `second`:

```
$ curl 'http://localhost:3333?first=1&second='
```

سيكون الخرج على النحو التالي:

```
This is my website!
```

لاحظ أن الخرج لم يتغير عن المرة السابقة، لكن إذا عدنا إلى خرج برنامج الخادم، فسنرى أن الخرج الجديد يتضمن قيم سلسلة الاستعلام:

```

[::]:3333: got / request. first(true)=1, second(true)=
```

يظهِر خرج قيمة سلسلة الاستعلام `first` أن التابع `Has` أعاد `true` لأن `first` لها قيمة، وأيضًا التابع `Get` أعاد القيمة `1`. يُظهر ناتج `second` أنه قد أعاد `true` لأننا ضمّنّا `second`، لكن التابع `Get` لم يُعيد أي شيء إلا سلسلة فارغة. يمكننا أيضًا محاولة إجراء طلبات مختلفة عن طريق إضافة وإزالة `first` و `second` أو إسناد قيم مختلفة لنرى كيف تتغير النتائج.

سيستمر البرنامج بالعمل، لذا يجب علينا إيقافه يدويًا من خلال الضغط على المفتاحين Ctrl+C.

حدّثنا في هذا القسم البرنامج لاستخدام `http.Server` واحد فقط مرةً أخرى، لكن أضفنا أيضًا دعمًا لقراءة قيم `first` و `second` من سلسلة الاستعلام لدالة المعالجة `getRoot`.

ليس استخدام سلسلة الاستعلام الطريقة الوحيدة للمستخدمين لتقديم مدخلات إلى خادم HTTP، فهناك طريقة أخرى شائعة لإرسال البيانات إلى الخادم وهي تضمين البيانات في متن الطلب. سنعدّل في القسم التالي البرنامج لقراءة نص الطلب من بيانات `*http.Request`.

38.9 قراءة متن الطلب

عند إنشاء واجهة برمجة تطبيقات مبنية على HTTP، مثل واجهة برمجة تطبيقات REST، قد تكون هناك حالات تتجاوز فيها البيانات المُرسلة قيود تضمينها في عنوان URL نفسه، مثل الطول الأعظمي للعنوان. قد نحتاج أيضًا إلى تلقي بيانات لا تتعلق بكيفية تفسير البيانات، وهذا يشير إلى الحالات التي لا ترتبط فيها

البيانات المرسلة في متن الطلب ارتباطًا مباشرًا بالمحتوى نفسه. بمعنى آخر: لا توفر هذه البيانات الإضافية إرشادات أو بيانات وصفية حول المحتوى، ولكنها تتضمن معلومات تكميلية تحتاج إلى المعالجة بطريقة منفصل. تخيل صفحة بحث، يمكن فيها للمستخدمين إدخال كلمات للبحث عن عناصر محددة. تمثل كلمات البحث نفسها تفسير المحتوى المقصود، ومع ذلك قد تكون هناك بيانات إضافية في متن الطلب، مثل تفضيلات المستخدم أو الإعدادات، والتي لا ترتبط مباشرةً باستعلام البحث نفسه ولكنها لا تزال بحاجة إلى النظر فيها أو معالجتها بواسطة الخادم. للتعامل مع مثل هذه السيناريوهات، يمكننا تضمين البيانات في متن طلب HTTP باستخدام توابع مثل `POST` أو `PUT`.

تُستخدم قيمة `*http.Request` في `http.HandlerFunc` للوصول إلى معلومات متعلقة بالطلب الوارد، بما في ذلك متن الطلب، والذي يمكن الوصول إليه من خلال حقل `Body`.

سنعدّل في هذا القسم دالة المعالجة `getRoot` لقراءة نص الطلب. لنفتح ملف `main.go` ونعدّل `getRoot` لاستخدام `ioutil.ReadAll` لقراءة حقل`r.Body` للطلب:

```
package main
import (
   ...
   "io/ioutil"
   ...
)
...
func getRoot(w http.ResponseWriter, r *http.Request) {
   ...
   second := r.URL.Query().Get("second")
   body, err := ioutil.ReadAll(r.Body)
   if err != nil {
         fmt.Printf("could not read body: %s\n", err)
   }
   fmt.Printf("%s: got / request. first(%t)=%s, second(%t)=%s, body:\n
%s\n",
         ctx.Value(keyServerAddr),
         hasFirst, first,
         hasSecond, second,
         body)
   io.WriteString(w, "This is my website!\n")
}
...
```

تُستخدم الدالة `ioutil.ReadAll` لقراءة `r.Body` من `*http.Request` لاسترداد بيانات متن الطلب. `ioutil.ReadAll` هي أداة مساعدة تقرأ البيانات من `io.Reader` حتى تنتهي من القراءة أو ظهور خطأ. نظرًا لأن `r.Body` هو io.Reader، فيمكن استخدامه لقراءة متن الطلب. نعدّل العبارة `fmt.Printf` بعد قراءة النص لتضمين محتوى المتن في الخرج.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

نُشغّل الآن أوامر `curl` (في الطرفية الثانية)، لتقديم طلب `POST` مع خيار `-X POST` وتحديد أن طريقة الطلب يجب أن تكون `POST`، ومتن طلب باستخدام الخيار `-b`. يُضبط متن الطلب على السلسلة النصيّة المقدمة، والتي في هذه الحالة هي "This is the body":

```
$ curl -X POST -d 'This is the body' 'http://localhost:3333?
first=1&second='
```

ليكون الخرج:

```
This is my website!
```

خرج دالة المعالجة هو نفسه، لكن سنرى تسجيلات الخادم الخاصة بك قد حُدّثت.

سيُرسل طلب `POST` عندما نشغّل هذا الأمر إلى الخادم المحلي الذي يعمل على المنفذ `3333` مع محتوى المتن المحدد ومعلمات سلسلة الاستعلام. بالتالي يعالج الخادم الطلب ونرى الخرج في سجلات الطرفية للخادم، بما في ذلك قيم سلسلة الاستعلام ومتن الطلب.

```
[::]:3333: got / request. first(true)=1, second(true)=, body:
This is the body
```

سيستمر البرنامج بالعمل، لذا يجب علينا إيقافه يدويًا من خلال الضغط على المفتاحين Ctrl+C.

عدّلنا في هذا القسم البرنامج لقراءة وطباعة متن الطلب. تفتح هذه الإمكانية إمكانيات التعامل مع أنواع مختلفة من البيانات، مثل جسون encoding/json، وتتيح إنشاء واجهات برمجة تطبيقات يمكنها التفاعل مع بيانات المستخدم بطريقة مألوفة.

تجدر الملاحظة إلى أنه لا تأتي جميع بيانات المستخدم في شكل واجهات برمجة التطبيقات. تتضمن العديد من مواقع الويب نماذج يملؤها المستخدمون، والتي ترسل البيانات إلى الخادم مثل بيانات نموذج أو استمارة form. سنعمل على تحسين البرنامج في القسم التالي ليشمل القدرة على قراءة بيانات النموذج ومعالجتها، بالإضافة إلى جسم الطلب وبيانات سلسلة الاستعلام التي كنا نعمل معها سابقًا.

38.10 استرجاع بيانات النموذج

لطالما كان إرسال البيانات باستخدام النماذج أو الاستمارات هو الطريقة القياسية للمستخدمين لإرسال البيانات إلى خادم HTTP والتفاعل مع مواقع الويب، وعلى الرغم من انخفاض شعبية النماذج بمرور الوقت، إلا أنها لا تزال تخدم أغراضًا مختلفة لتقديم البيانات. توفر قيمة `*http.Request` في `http.HandlerFunc` طريقةً للوصول إلى هذه البيانات، بطريقة مشابهة للطريقة التي توفر بها الوصول إلى سلسلة الاستعلام ومتن الطلب.

سنعدّل في هذا القسم الدالة `getHello` لتلقي اسم مستخدم من نموذج والرد بتحية شخصي، وللقيام بذلك سنفتح ملف main.go ونعدّل `getHello` لاستخدام التابع `PostFormValue` من `*http.Request`:

```
...
func getHello(w http.ResponseWriter, r *http.Request) {
  ctx := r.Context()
  fmt.Printf("%s: got /hello request\n", ctx.Value(keyServerAddr))
  myName := r.PostFormValue("myName")
  if myName == "" {
        myName = "HTTP"
  }
  io.WriteString(w, fmt.Sprintf("Hello, %s!\n", myName))
}
...
```

الآن في دالة `getHello` المحدَّثة، تجري قراءة قيم النموذج المرسلة إلى دالة المعالجة والبحث عن قيمة تسمى `myName`. إذا لم يُعثر على القيمة أو كانت سلسلة فارغة، تُضبط القيمة الافتراضية `HTTP` إلى المتغير `myName` لمنع عرض اسم فارغ على الصفحة. نعدّل أيضًا الخرج المرسل إلى المستخدم لكي يعرض الاسم الذي قدمه أو `HTTP` إذا لم يُقدّم أي اسم.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

نُشغّل الآن أوامر `curl` (في الطرفية الثانية)، لتقديم طلب `POST` مع خيار `-X POST`، وبدلًا من استخدام الخيار `-b`، نستخدم الخيار `-F 'myName=Sammy'` لتضمين بيانات النموذج مع حقل `myName` والقيمة `Sammy`:

```
curl -X POST -F 'myName=Sammy' 'http://localhost:3333/hello'
```

سيكون الخرج على النحو التالي:

```
Hello, Sammy!
```

يُستخدم التابع `r.PostFormValue` في `getHello` لاسترداد قيمة حقل النموذج `myName`. يبحث هذا التابع تحديدًا عن القيم المنشورة (البيانات المرسلة من العميل إلى الخادم) في متن الطلب. يمكن أيضًا استخدام التابع `r.FormValue`، والذي يتضمن كلًا من متن النموذج وأي قيم أخرى في سلسلة الاستعلام. إذا استخدمنا `r.FormValue("myName")` وأزلنا الخيار `-F`، فيمكن تضمين `myName = Sammy` في سلسلة الاستعلام لرؤية القيمة `Sammy`.

يوصى عمومًا بأن نكون أكثر صرامة وأن نستخدم الدالة `r.PostFormValue` إذا كنا نريد تحديدًا استرداد القيم من متن النموذج، إذ تتجنب هذه الدالة التعارضات أو الأخطاء المحتملة التي قد تنشأ عن خلط قيم النموذج من مصادر مختلفة. إذًا، تعد الدالة `r.PostFormValue` خيارًا أكثر أمانًا عند التعامل مع بيانات النموذج، لكن إذا كنا نحتاج إلى المرونة في وضع البيانات في كل من متن النموذج وسلسلة الاستعلام، فربما نضطر إلى الدالة الأخرى.

عند النظر إلى سجلات الخادم، سنرى أن طلب `/hello` قد جرى تسجيله بطريقة مشابهة للطلبات السابقة:

```
[::]:3333: got /hello request
```

سيستمر البرنامج بالعمل، لذا يجب علينا إيقافه يدويًا من خلال الضغط على المفتاحين Ctrl+C.

عدّلنا في هذا القسم البرنامج لقراءة اسم من بيانات النموذج المنشورة على الصفحة ثم إعادة هذا الاسم إلى المستخدم.

يمكن أن تسوء بعض الأشياء في هذه المرحلة من البرنامج، لا سيما عند التعامل مع أحد الطلبات، ولن يجري إعلام المستخدمين. لنعدّل في القسم التالي دوال المعالجة لإرجاع رموز حالة HTTP والترويسات headers.

38.11 الرد باستجابة تتضمن الترويسات ورمز الحالة

هناك بعض الميزات المستخدمة خلف الكواليس ضمن بروتوكول HTTP لتسهيل الاتصال الفعّال بين المتصفحات والخوادم. إحدى هذه الميزات هي "رموز الحالة"، والتي تعمل مثل وسيلة للخادم لتوضّح للعميل ما إذا كان الطلب ناجحًا أو واجه أي مشكلات في أي من الطرفين.

آلية اتصال أخرى تستخدمها خوادم وعملاء HTTP هي استخدام "حقول الترويسة header fields"، التي تتكون من أزواج ذات قيمة مفتاح يجري تبادلها بين العميل والخادم لنقل المعلومات عن أنفسهم. يمتلك بروتوكول HTTP عدة ترويسات معرّفة مسبقًا، مثل ترويسة `Accept`، التي يستخدمها العميل لإعلام الخادم

بنوع البيانات التي يمكنه التعامل معها. يمكن أيضًا تعريف ترويسات خاصة باستخدام البادئة `x-` متبوعة بالاسم المطلوب.

سنعمل في هذا القسم على تحسين البرنامج بجعل حقل النموذج `myName` في دالة المعالجة `getHello` حقلًا إلزاميًا. بالتالي، إذا لم تُعطى قيمة للحقل `myName`، فسيستجيب الخادم للعميل برمز الحالة "Bad Request طلب غير صالح" وتضمين الترويسة `x-missing-field` في الاستجابة، والتي تُعلم العميل بالحقل المفقود من الطلب.

لنفتح ملف main.go ونعدّل الدالة `getHello` وفقًا لما ذُكر:

```
...
func getHello(w http.ResponseWriter, r *http.Request) {
  ctx := r.Context()
  fmt.Printf("%s: got /hello request\n", ctx.Value(keyServerAddr))
  myName := r.PostFormValue("myName")
  if myName == "" {
        w.Header().Set("x-missing-field", "myName")
        w.WriteHeader(http.StatusBadRequest)
        return
  }
  io.WriteString(w, fmt.Sprintf("Hello, %s!\n", myName))
}
…
```

سابقًا: إذا كان حقل `myName` فارغًا، كان يُسند إليه قيمة افتراضية، أما الآن ضمن الدالة `getHello` المُحدَّثة، نُرسل رسالة خطأ إلى العميل. يُستخدم بدايةً التابع `w.Header().Set` لضبط الترويسة `x-missing-field` بقيمة `myName` في ترويسة الاستجابة، ثم التابع `w.WriteHeader` لكتابة ترويسات الاستجابة ورمز الحالة "طلب غير صالح" إلى العميل. أخيرًا تُستخدم تعليمة `return` لضمان إنهاء الدالة وعدم إرسال استجابة إضافية.

من الضروري التأكد من ضبط الترويسات وإرسال رمز الحالة بالترتيب الصحيح، إذ يجب إرسال جميع الترويسات في بروتوكول HTTP قبل الجسم، مما يعني أنه يجب إجراء أي تعديلات على `w.Header()` قبل استدعاء `w.WriteHeader`. عند استدعاء `w.WriteHeader` يُرسل رمز الحالة والترويسات، ويمكن كتابة المتن بعد ذلك حصرًا. يجب اتباع هذا الأمر لضمان حسن سير استجابة HTTP.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

نُشغّل الآن الأمر `curl -X POST` (في الطرفية الثانية) مع المسار `/hello` وبدون تضمين `-F` لإرسال بيانات النموذج. نحتاج أيضًا إلى تضمين الخيار `-v` لإخبار `curl` بإظهار الخرج المطوّل حتى نتمكن من رؤية جميع الترويسات والخرج للطلب:

```
curl -v -X POST 'http://localhost:3333/hello'
```

سنرى هذه المرة الكثير من المعلومات بسبب استخدامنا الخيار `-v`:

```
\*  Trying ::1:3333...
\* Connected to localhost (::1) port 3333 (#0)
\> POST /hello HTTP/1.1
\> Host: localhost:3333
\> User-Agent: curl/7.77.0
\> Accept: */*
\>
\* Mark bundle as not supporting multiuse
< HTTP/1.1 400 Bad Request
< X-Missing-Field: myName
< Date: Wed, 02 Mar 2022 03:51:54 GMT
< Content-Length: 0
<
\* Connection #0 to host localhost left intact
```

تشير الأسطر الأولى إلى أن `curl` تحاول إنشاء اتصال بالخادم عند منفذ المضيف المحلي `3333`. الأسطر المسبوقة بالرمز `>` تمثل الطلب المُرسل بواسطة `curl`، إذ يُرسل طلب `POST` إلى العنوان أو المسار `/hello` باستخدام بروتوكول HTTP 1.1. يتضمن الطلب ترويسات مثل `User-Agent` و `Accept` و `Host`. والجدير بالذكر أن الطلب لا يتضمن متنًا، كما هو موضح في السطر الفارغ.

تظهر استجابة الخادم بالسابقة `<` بعد إرسال الطلب. يشير السطر الأول إلى أن الخادم قد استجاب برمز حالة "طلب غير صالح" (المعروفة برمز الحالة 400)، كما تُضمّن الترويسة `X-Missing-Field` التي جرى ضبطها في ترويسة استجابة الخادم، مع تحديد أن الحقل المفقود هو `myName`. ينتهي الرد بدون أي محتوى في المتن، وهذا ما يتضح من طول المحتوى `0`.

إذا نظرنا مرةً أخرى إلى خرج الخادم، فسنرى طلب `/hello` للخادم الذي جرت معالجته في الخرج:

```
[::]:3333: got /hello request
```

سيستمر البرنامج بالعمل، لذا يجب علينا إيقافه يدويًا من خلال الضغط على المفتاحين Ctrl+C.

عدّلنا في هذا القسم خادم HTTP ليشمل التحقق من صحة إرسال الطلب `/hello`. إذا لم يُقدّم اسم في في الطلب، تُضبط ترويسة باستخدام التابع `w.Header().Set` للإشارة إلى الحقل المفقود. يُستخدم التابع `w.WriteHeader` بعد ذلك من أجل كتابة الترويسات إلى العميل، جنبًا إلى جنب مع رمز الحالة التي تشير إلى "طلب غير صالح".

إذًا يُخبر الخادم العميل بوجود مشكلة في الطلب من خلال ضبط الترويسة ورمز الحالة. يسمح هذا الأسلوب بمعالجة الأخطاء بطريقة صحيحة ويوفر ملاحظات للعميل فيما يتعلق بحقل النموذج المفقود.

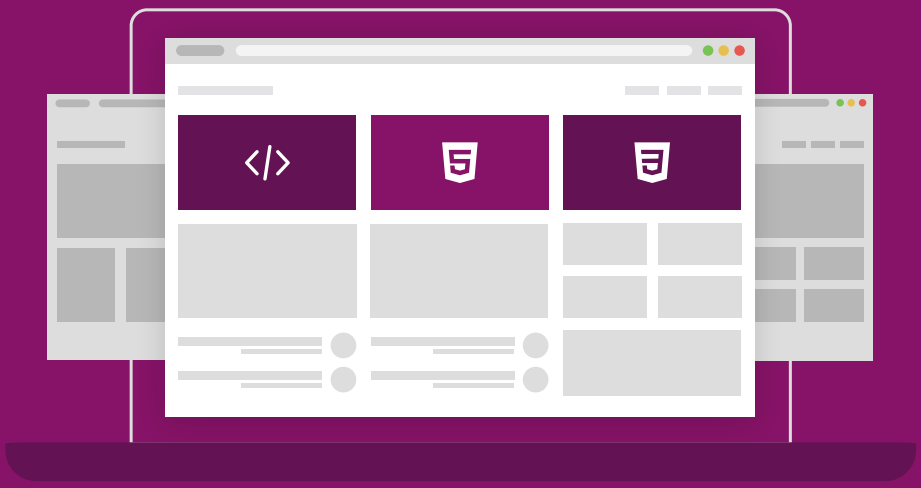
38.12 الخاتمة

تعلمنا في هذا الفصل كيفية إنشاء خادم HTTP في لغة جو باستخدام حزمة `net/http`، وطوّرنا الخادم باستخدام مجمّع مخصص للخادم واستخدام نسخ `http.Server` متعددة. اسكتشفنا أيضًا طرقًا مختلفة للتعامل مع مُدخلات المستخدم، مثل قيم سلسلة الاستعلام ومتن الطلب وبيانات النموذج. تحققنا أيضًا من صحة الطلب من خلال إعادة ترويسات HTTP مخصصة ورموز حالة إلى العميل.

تتمثل إحدى نقاط القوة في نظام بروتوكول HTTP في توافقه مع العديد من الأطر التي تتكامل بسلاسة مع حزمة `net/http`. توضح مشاريع مثل github.com/go-chi/chi ذلك من خلال توسيع وظائف خادم http القياسي من خلال تحقيق الواجهة `http.Handler`. يتيح ذلك للمطورين الاستفادة من البرامج الوسيطة والأدوات الأخرى دون الحاجة لإعادة كتابة المنطق الخاص الخادم، وهذا يسمح بالتركيز على إنشاء برمجيات وسيطة middleware وأدوات أخرى لتحسين الأداء بدلًا من التعامل مع الوظائف الأساسية فقط.

توفر حزمة `net/http` المزيد من الإمكانات التي لم نتناولها في هذا الفصل، مثل العمل مع ملفات تعريف الارتباط وخدمة حركة مرور HTTPS.

دورة تطوير واجهات المستخدم



ابدأ مسارك المهني كمطور واجهات المواقع والمتاجر الإلكترونية
فور انتهائك من الدورة

التحق بالدورة الآن

39. كيفية إنشاء طلبات HTTP

يُعد بروتوكول HTTP الخيار الأفضل غالبًا عندما يحتاج المطورون لإنشاء اتصال بين البرامج. وتقدم لغة جو دعمًا قويًا لبروتوكول HTTP من خلال حزمة `net/http` الموجودة في المكتبة القياسية. إذ لا تتيح هذه الحزمة إنشاء خوادم HTTP فحسب، بل تتيح أيضًا إجراء طلبات عميل HTTP.

سننشئ في هذا الفصل برنامجًا يتفاعل مع خادم HTTP من خلال إجراء أنواع مختلفة من الطلبات. سنبدأ بطلب `GET` باستخدام عميل HTTP الافتراضي في لغة جو، ثم سنعمل على تحسين البرنامج ليشمل طلب `POST` مع متن الطلب. أخيرًا نُخصص طلب `POST` من خلال دمج ترويسة HTTP وتحقيق آلية المهلة، للتعامل مع الحالات التي تتجاوز فيها مدة الطلب حدًا معينًا.

39.1 المتطلبات الأولية

لمتابعة هذا الفصل من الكتاب، ستحتاج إلى:

- إصدار مُثبّت من جو 1.16 أو أعلى، بحسب التعليمات الواردة في الفصل الأول من الكتاب.
- القدرة على استخدام أداة curl لإجراء طلبات ويب.
- إلمام بكيفية استخدام جسون JSON في لغة جو.
- معرفة بكيفية استخدام السياقات Contexts في لغة جو Go.
- فهم لتنظيمات جو goroutines والقنوات channels الموضحة في فصل تشغيل عدة دوال على التساير في لغة جو Go.
- الإلمام بكيفية إنشاء طلبات HTTP وإرسالها (موصى به).

39.2 تقديم طلب GET

نناقش في هذه الفقرة كيفية تقديم طلب GET باستخدام حزمة net/http مثل عميل، إذ توفّر لنا هذه الحزمة طرقًا وخيارات متنوعة للتفاعل مع موارد HTTP. أحد الخيارات الشائعة هو استخدام عميل "HTTP عام" مع دوال مثل http.Get، التي تسمح بإنشاء طلب GET سريعًا باستخدام عنوان URL ومتن فقط. يمكننا أيضًا إنشاء http.Request للحصول على مزيد من التحكم وتخصيص جوانب معينة من الطلب. سنبدأ في هذا القسم بإنشاء برنامج أولي يستخدم http.Get لتقديم طلب HTTP، ونعدّله لاحقًا لاستخدام http.Request مع عميل HTTP الافتراضي.

39.2.1 استخدام دالة http.Get لتقديم طلب

نستخدم الدالة http.Get في الإصدار الأولي من البرنامج لإرسال طلب إلى خادم HTTP داخل البرنامج، إذ تُعد هذه الدالة مناسبة لأنها تتطلب الحد الأدنى من التهيئة، أي أنها لا تحتاج إلى الكثير من التفاصيل والإعدادات، وهذا ما يجعلها مثالية لتقديم طلب واحد مباشر. بالتالي، يكون استخدام الدالة http.Get هو الأسلوب الأنسب عندما نحتاج إلى تنفيذ طلب سريع لمرة واحدة فقط.

كما هو معتاد، سنحتاج لبدء إنشاء برامجنا إلى إنشاء مجلد للعمل ووضع الملفات فيه، ويمكن وضع المجلد في أي مكان على الحاسب، إذ يكون للعديد من المبرمجين عادةً مجلدٌ يضعون داخله كافة مشاريعهم. سنستخدم في هذا الفصل مجلدًا باسم projects، لذا فلننشئ هذا المجلد وننتقل إليه:

```
$ mkdir projects
$ cd projects
```

ننشئ مجلدًا للمشروع وننتقل إليه. لنسميه مثلًا httpclient:

```
$ mkdir httpclien
$ cd httpclient
```

نستخدم الآن محرر نانو nano أو أي محرر آخر تريده لفتح ملف main.go:

```
$ nano main.go
```

نضع بداخله الشيفرة التالية:

```
package main
import (
  "errors"
  "fmt"
```

```
    "net/http"
    "os"
    "time"
)
const serverPort = 3333 // 3333 ضبط منفذ الخادم على
```

أضفنا في الشيفرة السابقة الحزمة `main` لضمان إمكانية تصريف البرنامج وتنفيذه. نستورد عدة حزم لاستخدامها في البرنامج. نُعرّف ثابت `const` اسمه `serverPort` بقيمة `3333`، سيجري استخدامه مثل منفذ لخادم HTTP والعميل.

ننشئ دالة `main` ضمن الملف main.go ونبدأ بإعداد خادم HTTP مثل تنظيم goroutine:

```
func main() {
    // نبدأ تشغيل خادم HTTP مثل تنظيم جو
    go func() {
        // نُنشئ جهاز توجيه جديد
        mux := http.NewServeMux()
        // معالجة مسار الجذر "/" وطباعة معلومات الطلب
        mux.HandleFunc("/", func(w http.ResponseWriter, r
*http.Request) {

            fmt.Printf("Server: %s /\n", r.Method)

        })
        // تهيئة خادم HTTP
        server := http.Server{
            Addr:  fmt.Sprintf(":%d", serverPort),
            Handler: mux,
        }
        // بدء تشغيل الخادم، ومعالجة الأخطاء المحتملة
        if err := server.ListenAndServe(); err != nil {
            if !errors.Is(err, http.ErrServerClosed) {
                fmt.Printf("Error running HTTP server: %s\n", err)
            }
        }
    }()
    // الانتظار لمدة قصيرة للسماح للخادم بالبدء
```

```
    time.Sleep(100 * time.Millisecond)
}
```

تعمل الدالة `main` بمثابة نقطة دخول للبرنامج، ونستخدم الكلمة المفتاحية `go` للإشارة إلى أن خـادم HTTP سـيجري تشـغيله ضـمن تنظيم جـو goroutine. نعـالج مسـار الجـذر `/`، ونطبـع معلومـات الطلب باسـتخدام `fmt.Printf`، ونُهيّئ خادم HTTP بالعنوان والمعالج المحددين.

يبدأ الخادم بالاستماع إلى الطلبات باستخدام الدالة `ListenAndServe`، ويجـري التعامـل مـع أيـة أخطـاء محتملة؛ فإذا حدث خطأ ولم يكن هذا الخطأ هـو `http.ErrServerClosed` (إغلاق طـبيعي تحـدثنا عنـه في الفصل السابق)، ستُطبع رسالة خطأ. نستخدم الدالة `time.Sleep` للسماح للخادم بوقت كـافٍ لبـدء التشـغيل قبل تقديم الطلبات إليه.

نجري الآن بعض التعديلات الإضافية على الدالة `main()` من خلال إعداد عنوان URL للطلب، وذلك بـدمج اسم المضيف `http://localhost` مع قيمة `serverPort` باستخدام `fmt.Sprintf`. نستخدم بعد ذلـك الدالة `http.Get` لتقديم طلب إلى عنوان URL هذا:

```
...
    requestURL := fmt.Sprintf("http://localhost:%d", serverPort)
    res, err := http.Get(requestURL)
    if err != nil {
          fmt.Printf("error making http request: %s\n", err)
          os.Exit(1)
    }
    fmt.Printf("client: got response!\n")
    fmt.Printf("client: status code: %d\n", res.StatusCode)
}
```

يرسـل البرنـامج طلب HTTP باسـتخدام عميـل HTTP الافتراضـي إلى عنـوان URL المحـدد عنـد اسـتدعاء `http.Get`، ويعيد http.Response في حالة نجاح الطلب أو ظهور قيمة خطأ في حالة فشل الطلب. في حال حدوث خطأ، يطبع البرنـامج رسـالة الخطـأ ويخـرج من البرنـامج باسـتخدام os.Exit مـع شـيفرة خطـأ 1. إذا نجح الطلب، يطبع البرنامج رسالةً تشير إلى تلقي استجابة، جنبًا إلى جنب مع شيفرة حالة HTTP للاستجابة.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج على النحو التالي:

```
server: GET /
client: got response!
client: status code: 200
```

يشير السطر الأول من الخرج إلى أن الخادم تلقى طلب GET من العميل على المسار /. يشير السطران التاليان إلى أن العميل قد تلقى استجابة من الخادم بنجاح وأن شيفرة حالة الاستجابة كان 200.

على الرغم من أن الدالة http.Get ملائمة لإجراء طلبات HTTP سريعة مثل تلك الموضحة في هذا القسم، لكن استخدام http.Request يوفر نطاقًا أوسع من الخيارات لتخصيص الطلب.

39.2.2 استخدام دالة http.Request لتقييم طلب

توفّر لنا http.Request مزيدًا من التحكم في الطلب أكثر من مجرد استخدام تابع HTTP (على سبيل المثال، GET و POST) وعنوان URL فقط. على الرغم من أننا لن نستخدم ميزات إضافية في الوقت الحالي، يسمح لنا استخدام http.Request بإضافة تخصيصات في أقسام لاحقة من هذا الفصل.

يتضمن التحديث الأولي تعديل معالج خادم HTTP لإرجاع استجابة تتضمّن بيانات جسون JSON وهمية باستخدام fmt.Fprintf. تُنشأ هذه البيانات ضمن خادم HTTP مكتمل باستخدام حزمة encoding/json. لمعرفة المزيد حول العمل مع بيانات جسون في لغة جو، يمكن الرجوع إلى الفصل كيفية استخدام جسون JSON في لغة جو. نحتاج أيضًا إلى استيراد io/ioutil لاستخدامه في تحديث لاحق.

نفتح ملف main.go لتضمين http.Request كما هو موضح أدناه:

```
package main
import (
   ...
   "io/ioutil"
   ...
)
...
func main() {
   ...
   mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
         fmt.Printf("server: %s /\n", r.Method)
         fmt.Fprintf(w, `{"message": "hello!"}`)
   })
   ...
```

هـدفنا هـو اسـتبدال اسـتخدام `http.Get`، من خلال اسـتخدام كـل من الدالـة `http.NewRequest` من الحزمـة `net/http`-لإنشـاء `http.Request` جديـد، والـذي يمثـل طلب HTTP يمكن إرسـاله إلى الخـادم و `http.DefaultClient` ليكـون عميـل HTTP افتراضـي يـوفر لنـا اسـتخدام التـابع `Do` لإرسـال `http.Request` إلى الخادم واسترداد `http.Response` المقابل. يسـمح هـذا التغيـير بمزيـد من التحكم في الطلب وإجراء تعديلات عليه قبل الإرسال إلى الخادم:

```
...
  requestURL := fmt.Sprintf("http://localhost:%d", serverPort)
  req, err := http.NewRequest(http.MethodGet, requestURL, nil)

  if err != nil {
        fmt.Printf("client: could not create request: %s\n", err)
        os.Exit(1)
  }
  res, err := http.DefaultClient.Do(req)
  if err != nil {
        fmt.Printf("client: error making http request: %s\n", err)
        os.Exit(1)
  }
  fmt.Printf("client: got response!\n")
  fmt.Printf("client: status code: %d\n", res.StatusCode)
  resBody, err := ioutil.ReadAll(res.Body)

  if err != nil {
        fmt.Printf("client: could not read response body: %s\n", err)
        os.Exit(1)
  }
  fmt.Printf("client: response body: %s\n", resBody)
}
```

نبدأ باستخدام الدالة `http.NewRequest` لإنشاء قيمة `http.Request`. نحـدد تـابع HTTP للطلب على أنـه `GET` باسـتخدام `http.MethodGet`. نُنشـئ أيضًـا عنـوان URL للطلب من خلال دمج اسـم المضـيف `http://localhost` مع قيمة `serverPort` باستخدام `fmt.Sprintf`. نفحص أيضًا متن الطلب إذا كـان فارغًا (أي قيمة `nil`) لنتعامل أيضًا مع أي خطأ محتمل قد يحدث أثناء إنشاء الطلب.

بخلاف `http.Get` الذي يرسل الطلب على الفور، فإن`http.NewRequest` يُجهّز الطلب فقط ولا يرسـله مباشرةً، ويتيح لنا ذلك تخصيص الطلب كما نريد قبل إرساله فعليًا.

بمجرد إعداد `http.Request`، نستخدم`http.DefaultClient.Do(req)` لإرسال الطلب إلى الخادم باستخدام عميل HTTP الافتراضي `http.DefaultClient`. يبدأ التابع `Do` الطلب ويعيد الاستجابة المستلمة من الخادم مع أي خطأ محتمل.

بعد الحصول على الرد نطبع معلومات عنه، إذ نعرض أن العميل قد تلقى استجابة بنجاح، إلى جانب شيفرة حالة HTTP للاستجابة.

نقرأ بعد ذلك متن استجابة HTTP باستخدام الدالة ioutil.ReadAll. يُمثّل متن الاستجابة على أنه `io.ReadCloser`، والذي يجمع بين `io.Reader` و `io.Closer`. نستخدم `ioutil.ReadAll` لقراءة جميع البيانات من متن الاستجابة حتى النهاية أو حدوث خطأ. تُعيد الدالة البيانات بقيمة `[]byte`، والتي نطبعها مع أي خطأ مصادف.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

الخرج مشابه لما سبق مع إضافة بسيطة:

```
server: GET /
client: got response!
client: status code: 200
client: response body: {"message": "hello!"}
```

يُظهر السطر الأول أن الخادم لا يزال يتلقى طلب `GET` إلى المسار `/`. يتلقى العميل استجابة من الخادم برمز الحالة `200`.حيث يقرأ العميل متن استجابة الخادم ويطبعها أيضًا، والتي تكون في هذه الحالة على النحو التالي `{" !message ":" hello "}`. يمكننا معالجة استجابة جسون هذه باستخدام حزم `encoding/json` أيضًا لكن لن ندخل في هذه التفاصيل الآن.

طورّنا في هذا القسم برنامج باستخدام خادم HTTP وقدمنا طلبات HTTP إليه باستخدام طرق مختلفة. استخدمنا في البداية الدالة `http.Get` لتنفيذ طلب `GET` للخادم باستخدام عنوان URL الخاص بالخادم فقط. حدّثنا بعد ذلك البرنامج لاستخدام `http.NewRequest` لإنشاء قيمة `http.Request`، ثم استخدمنا التابع `Do` لعميل HTTP الافتراضي`http.DefaultClient`، لإرسال الطلب وطباعة متن الاستجابة.

تُعد طلبات `GET` مفيدةً لاسترداد المعلومات من الخادم، لكن بروتوكول HTTP يوفر طرقًا أخرى متنوعة للاتصال بين البرامج. إحدى هذه الطرق هي طريقة `POST`، والتي تتيح لك إرسال معلومات من برنامجك إلى الخادم.

39.3 إرسال طلب POST

يُستخدم طلب GET في واجهة برمجة تطبيقات REST، فقط لاسترداد المعلومات من الخادم، لذلك لكي يُشارك البرنامج بالكامل في REST، يحتاج أيضًا إلى دعم إرسال طلبات POST، وهو عكس طلب GET تقريبًا، إذ يرسل العميل البيانات إلى الخادم داخل متن الطلب.

سنُعدّل في هذا القسم البرنامج لإرسال طلب POST بدلًا من طلب GET، إذ سيتضمن طلب POST متنًا للطلب، وسنُعدّل الخادم لتوفير معلومات أكثر تفصيلًا حول الطلبات الواردة من العميل.

نفتح ملف main.go ونُضمّن الحزم الإضافية التالية:

```
...
import (
  "bytes"
  "errors"
  "fmt"
  "io/ioutil"
  "net/http"
  "os"
  "strings"
  "time"
)
...
```

ثم نُعدّل دالة المعالجة لطباعة معلومات متنوعة حول الطلب الوارد، مثل قيم سلسلة الاستعلام وقيم الترويسة ومتن الطلب:

```
...
 mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("server: %s /\n", r.Method)
    fmt.Printf("server: query id: %s\n", r.URL.Query().Get("id"))
    fmt.Printf("server: content-type: %s\n", r.Header.Get("content-
type"))

    fmt.Printf("server: headers:\n")

    for headerName, headerValue := range r.Header {
```

```
            fmt.Printf("\t%s = %s\n", headerName,
strings.Join(headerValue, ", "))
    }
    reqBody, err := ioutil.ReadAll(r.Body)
    if err != nil {
                fmt.Printf("server: could not read request body: %s\n",
err)
    }
    fmt.Printf("server: request body: %s\n", reqBody)
    fmt.Fprintf(w, `{"message": "hello!"}`)
 })
...
```

أضفنا في هذا التحديث لمعالج طلب HTTP الخاص بالخادم تعليمات `fmt.Printf` لتوفير مزيد من المعلومات حول الطلب الوارد. يُستخدم التابع `r.URL.Query().Get` لاسترداد قيمة سلسلة الاستعلام المسماة `id`. يُستخدم التابع `r.Header.Get` للحصول على قيمة الترويسة `content-type`. تتكرر حلقة `for` مع `r.Header` فوق كل ترويسة HTTP يستقبلها الخادم وتطبع اسمها وقيمتها. يمكن أن تكون هذه المعلومات ذات قيمة لأغراض استكشاف الأخطاء وإصلاحها في حالة ظهور أي مشكلات مع سلوك العميل أو الخادم. تُستخدم أيضًا الدالة `ioutil.ReadAll` لقراءة متن الطلب من `r.Body`.

بعد تحديث دالة المعالجة للخادم، نُعدّل شيفرة الطلب في الدالة `main`، بحيث ترسل طلب `POST` مع متن الطلب:

```
...
 time.Sleep(100 * time.Millisecond)
 jsonBody := []byte(`{"client_message": "hello, server!"}`)
 bodyReader := bytes.NewReader(jsonBody)
 requestURL := fmt.Sprintf("http://localhost:%d?id=1234", serverPort)
 req, err := http.NewRequest(http.MethodPost, requestURL, bodyReader)
...
```

في هذا التعديل قُدّم متغيران جديدان: يمثل الأول `jsonBody` بيانات جسون مثل قيم من النوع `[]byte` بدلًا من سلسلة `string`. يُستخدم هذا التمثيل لأنه عند ترميز بيانات جسون باستخدام حزمة `encoding/json`، فإنها تُرجع `[]byte` بدلًا من سلسلة.

المتغير الثاني `bodyReader`، هو `bytes.Reader` مُغلّف ببيانات `jsonBody`. يتطلب `http.Request` أن يكون متن الطلب من النوع `io.Reader`. نظرًا لأن قيمة `jsonBody` هي `[]byte` ولا تُحقق `io.Reader`

مباشـرةً، يُسـتخدم `bytes.Reader` لتوفـير واجهـة `io.Reader` الضـرورية، ممـا يسـمح باسـتخدام قيمـة `jsonBody` على أنها متن الطلب.

نُعدّل أيضًا المتغير `requestURL` لتضميـن قيمة سـلسـلة الاستعلام `id = 1234`، وذلك لتوضيح كيف يمكن تضمين سلسلة استعلام في عنوان URL للطلب إلى جانب مكونات عنوان URL القياسـية الأخـرى. أخـيرًا نُعدّل اسـتدعاء الدالـة `http.NewRequest` لاسـتخدام التـابع `POST` مـع `http.MethodPost`، وضـبط متن الطلب على `bodyReader`، وهو قارئ بيانات جسون.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج بالتأكيد أطول من السابق، بسبب طباعة معلومات إضافية:

```
server: POST /
server: query id: 1234
server: content-type:
server: headers:
    Accept-Encoding = gzip
    User-Agent = Go-http-client/1.1
    Content-Length = 36
server: request body: {"client_message": "hello, server!"}
client: got response!
client: status code: 200
client: response body: {"message": "hello!"}
```

يُظهر الخرج السلوك المُحدّث للخادم والعميل بعد إجـراء التغيـيرات لإرسـال طلب `POST`؛ إذ يشـير السـطر الأول إلى أن الخادم قد تلقى طلب `POST` للمسار `/`؛ ويعرض السطر الثاني قيمـة سلسـلة الاسـتعلام `id`، والـتي تحمل القيمة `1234` في هذا الطلب؛ بينما يعرض السطر الثالث قيمـة ترويسـة `Content-Type`، وهي فارغـة في هذه الحالة.

قد يختلف ترتيب الترويسات المطبوعة من `r.Headers` أثناء التكرار عليها باستخدام `range` في كـل مـرة نُشغّل فيها البرنامج، فبدءًا من إصدار جو 1.12، لا يمكن ضمان أن يكون الترتيب الذي يجـري فيـه الوصـول إلى العناصر هو نفس الترتيب الذي تُدرج في العناصر في الخريطة `map`، وهذا لأن لغة جو تُحقق الخرائـط باسـتخدام بنية لا تحافظ على ترتيب الإدراج، لذا قد يختلف ترتيب طباعة الترويسات عن الترتيب الذي جـرى اسـتلامها بـه فعليًـا في طلب HTTP. الترويسـات الموضـحة في المثـال هي `Accept-Encoding` و `User-Agent`

و `Content-Length`. قد نرى أيضًا قيمة مختلفة للترويسة `User-Agent` عما رأيناه أعلاها اعتمادًا على إصدار لغة جو المُستخدم.

يعرض السطر التالي متن الطلب الذي استلمه الخادم، وهو بيانات جسون التي يرسلها العميل. يمكن للخادم بعد ذلك استخدام حزمة `encoding/json` لتحليل بيانات جسون هذه التي أرسلها العميل وصياغة استجابة. تمثل الأسطر التالية خرج العميل، مما يشير إلى أنه تلقى استجابةً مع رمز الحالة `200`. يحتوي متن الاستجابة المعروض في السطر الأخير على بيانات جسون أيضًا.

أجرينا في القسم السابق العديد من التحديثات لتحسين البرنامج. أولًا، استبدلنا طلب `GET` بطلب `POST`، مما يتيح للعميل إرسال البيانات إلى الخادم في متن الطلب، إذ أنجزنا هذا عن طريق تحديث شيفرة الطلب وتضمين متن الطلب باستخدام `[]byte`. عدّلنا أيضًا دالة معالجة الطلب الخاصة بالخادم لتوفير معلومات أكثر تفصيلًا حول الطلبات الواردة، مثل قيمة سلسلة الاستعلام `id` وترويسة `Content-Type` وجميع الترويسات المستلمة من العميل.

تجدر الإشارة إلى أنه في الخرج المعروض، لم تكن ترويسة `Content-Type` موجودة في طلب HTTP، مما يشير عادةً إلى نوع المحتوى المُرسل في المتن. سنتعلم في القسم التالي المزيد عن كيفية تخصيص طلب HTTP، بما في ذلك ضبط ترويسة `Content-Type` لتحديد نوع البيانات المُرسلة.

39.4 تخصيص طلب HTTP

يسمح تخصيص طلب HTTP بنقل مجموعة واسعة من أنواع البيانات بين العملاء والخوادم. كان بإمكان عملاء HTTP سابقًا افتراض أن البيانات المستلمة من خادم HTTP هي HTML غالبًا، أما اليوم يمكن أن تشمل البيانات تنسيقات مختلفة، مثل جسون والموسيقى والفيديو وغيرهم. لنقل معلومات إضافية متعلقة بالبيانات المرسلة، يشتمل بروتوكول HTTP على العديد من الترويسات، أهمها الترويسة `Content-Type`، التي تُخبر الخادم (أو العميل، اعتمادًا على اتجاه البيانات) بكيفية تفسير البيانات التي يتلقاها.

سنعمل في القسم التالي على تحسين البرنامج عن طريق ضبط الترويسة `Content-Type` في طلب HTTP للإشارة إلى أن الخادم يجب أن يتوقع بيانات جسون. ستتاح لنا الفرصة أيضًا لاستخدام عميل HTTP مخصص بدلًا من `http.DefaultClient` الافتراضي، مما يتيح لنا تخصيص إرسال الطلب وفقًا للمتطلبات المحددة.

نفتح ملف main.go لإجراء التعديلات الجديدة على الدالة `main`:

```
...
 req, err := http.NewRequest(http.MethodPost, requestURL, bodyReader)
 if err != nil {
```

```
            fmt.Printf("client: could not create request: %s\n", err)
            os.Exit(1)
    }
    req.Header.Set("Content-Type", "application/json")
    client := http.Client{
      Timeout: 30 * time.Second,
    }
  res, err := client.Do(req)
    if err != nil {
      fmt.Printf("client: error making http request: %s\n", err)
      os.Exit(1)
    }
  ...
```

بدايـةً، جـرى النفـاذ إلى ترويسـات `http.Request` باسـتخدام `req.Header` وضـبطنا ترويسـة `Content-Type` على `application/json`، إذ يُمثّل ضبط هـذه الترويسـة إشـارةً للخـادم إلى أن البيانـات الموجـودة في متن الطلب مُنسّـقة بتنسـيق جسـون، وهـذا يسـاعد الخـادم في تفسـير متن الطلب ومعالجتـه بطريقة صحيحة.

أنشأنا أيضًا نسخة خاصة من البنية `http.Client` في المتغير `client`. يتيح لنـا هـذا مزيـدًا من التحكم في سلوك العميل. يمكننا في هذه الحالة ضبط قيمة المهلة `Timeout` للعميل على 30 ثانية؛ وهذا يعني أنه إذا لم يجري تلقي الرد في غضون 30 ثانية، سيستسلم العميل ويتوقـف عن الانتظـار. هـذا مهم لمنـع البرنـامج من إهدار الموارد بإبقاء الاتصالات مفتوحة إلى أجل غير مسمى.

من خلال إنشاء `http.Client` خاص بنا مع مهلة محددة، فإننا نتأكد من أن البرنامج لديه حد محدد للمدة التي سينتظرها الرد، ويختلف هـذا عن اسـتخدام `http.DefaultClient` الافتراضـي، والـذي لا يحـدد مهلـة وسينتظر إلى أجل غير مسمى. لذلك يعد تحديد المهلة أمرًا بالغ الأهمية لإدارة الموارد بفعالية وتجنب مشـكلات الأداء المحتملة.

أخيرًا عدّلنا الطلب لاستخدام التابع `Do` من المتغير `client`. هذا التغيير واضح ومباشـر لأننـا كنـا نسـتخدم التابع `Do` في جميع أنحاء البرنامج، سواء مع العميل الافتراضي أو العميل المخصص. الاختلاف الوحيـد الآن هـو أننا أنشأنا صراحةً متغيرًا من `http.Client`.

توفر هـذه التحـديثات مزيـدًا من التحكم والمرونـة لبرنامجنـا، ممـا يسـمح بضـبط ترويسـات محـددة وإدارة المهلات الزمنية بالطريقة المناسبة. من خلال فهم وتخصيص هذه الجوانب لطلب HTTP، يمكننـا ضـمان الأداء الأمثل والتواصل الموثوق مع الخادم.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

يجب أن تكون مخرجاتك مشابهة جدًا للإخراج السابق ولكن مع مزيد من المعلومات حول نوع المحتوى:

```
server: POST /
server: query id: 1234
server: content-type: application/json
server: headers:
    Accept-Encoding = gzip
    User-Agent = Go-http-client/1.1
    Content-Length = 36
    Content-Type = application/json
server: request body: {"client_message": "hello, server!"}
client: got response!
client: status code: 200
client: response body: {"message": "hello!"}
```

يمكننا ملاحظة أن الخادم قد استجاب بقيمة الترويسة `Content-Type` المُرسلة من قِبل العميل وهي من النوع `application/json`، ونلاحظ القيمة `content-type` وهي أيضًا `application/json`.

يسمح وجود `Content-Type` بالمرونة في التعامل مع أنواع مختلفة من واجهات برمجة التطبيقات في وقت واحد، إذ يمكّن الخادم من التمييز بين الطلبات التي تحتوي على بيانات جسون والطلبات التي تحتوي على بيانات XML. يصبح هذا التمييز مهمًا خاصةً عند التعامل مع واجهات برمجة التطبيقات التي تدعم تنسيقات بيانات مختلفة.

لفهم كيفية عمل مهلة العميل Timeout، يمكننا تعديل الشيفرة لمحاكاة سيناريو يستغرق فيه الطلب وقتًا أطول من المهلة المحددة، وذلك لكي نتمكن من ملاحظة تأثير المهلة.

يمكننا إضافة استدعاء دالة `time.Sleep` ضمن دالة المعالجة لخادم HTTP، إذ تؤدي هذه الدالة إلى تأخير مُصطنع في استجابة الخادم. للتأكد من أن التأخير يتجاوز قيمة المهلة التي حددناها، نجعل `time.Sleep` يستمر لمدة 35 ثانية، وبالتالي سيظهر الخادم وكأنه غير مستجيب، مما يتسبب في انتظار العميل للاستجابة:

```
...
func main() {
  go func() {
        mux := http.NewServeMux()
```

```
        mux.HandleFunc("/", func(w http.ResponseWriter, r
*http.Request) {
                ...
                fmt.Fprintf(w, `{"message": "hello!"}`)
                time.Sleep(35 * time.Second)
        })
        ...
  }()
  ...
}
```

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

عنـد تنفيـذ البـرنامج مع إضـافة `time.Sleep (35 * time.Second)` في دالـة المعـالجة، سيتغير سلوك العميل والخادم، وذلك بسبب التأخير المصطنع الذي حدث في استجابة الخادم. نلاحظ أيضًا أن البرنـامج يستغرق وقتًا أطول للخروج مقارنةً بالسابق، وذلك لأن البرنامج ينتظر انتهاء طلب HTTP قبـل الإنهـاء. الآن مـع الانتظار الجديد الذي أضفناه، لن يكتمل طلب HTTP حتى تصل الاستجابة أو نتجاوز المهلة المحـددة البالغـة 30 ثانية، ونظرًا لأن الخادم متوقف مؤقتًا لمدة 35 ثانية، متجاوزًا مدة المهلة، فسيلغي العميل الطلب بعد 30 ثانية، وستجري عملية معالجة خطأ المهلة الزمنية:

```
server: POST /
server: query id: 1234
server: content-type: application/json
server: headers:
    Content-Type = application/json
    Accept-Encoding = gzip
    User-Agent = Go-http-client/1.1
    Content-Length = 36
server: request body: {"client_message": "hello, server!"}
client: error making http request: Post "http://localhost:3333?
id=1234": context deadline exceeded (Client.Timeout exceeded while
awaiting headers)
exit status 1
```

نلاحظ تلقي الخادم طلب POST بطريقة صحيحة إلى المسار /، والتقط أيضًا معامل الاستعلام id بقيمة 1234. حدد الخادم نوع محتوى الطلب content-type على أنه application/json بناءً على ترويسة نوع المحتوى، كما عرض الخادم الترويسات التي تلقاها. طبع الخادم أيضًا متن الطلب.

نلاحظ حدوث خطأ context deadline exceeded من جانب العميل أثناء طلب HTTP كما هو متوقع. تشير رسالة الخطأ إلى أن الطلب تجاوز المهلة الزمنية البالغة 30 ثانية (الخادم تلقى الطلب وعالجه، لكنه بسبب وجود تعليمة انتظار من خلال استدعاء دالة time.Sleep، سيبدو وكأنه لم ينتهي من معالجته). بالتالي سيفشل استدعاء التابع client.Do، ويخرج البرنامج مع رمز الحالة 1 باستخدام os.Exit (1).

يسلط هذا الضوء على أهمية ضبط قيم المهلة الزمنية بطريقة مناسبة لضمان عدم تعليق الطلبات إلى أجل غير مسمى.

أجرينا خلال هذا القسم تعديلات على البرنامج لتخصيص طلب HTTP من خلال إضافة ترويسة | Content-Type إليه. عدّلنا البرنامج أيضًا من خلال إنشاء http.Client جديد مع مهلة زمنية 30 ثانية، والتي استخدمناها لاحقًا لتنفيذ طلب HTTP. فحصنا أيضًا تأثير المهلة من خلال استخدام التعليمة time.Sleep داخل معالج طلب HTTP. سمح لنا هذا بملاحظة أن استخدام عميل http.Client مع المهلات الزمنية المضبوطة بدقة أمرٌ بالغ الأهمية لمنع الطلبات من الانتظار إلى أجل غير مسمى. بالتالي اكتسبنا رؤى حول أهمية ضبط المهلات المناسبة لضمان معالجة الطلب بكفاءة ومنع المشكلات المحتملة مع الطلبات التي قد تظل خاملة.

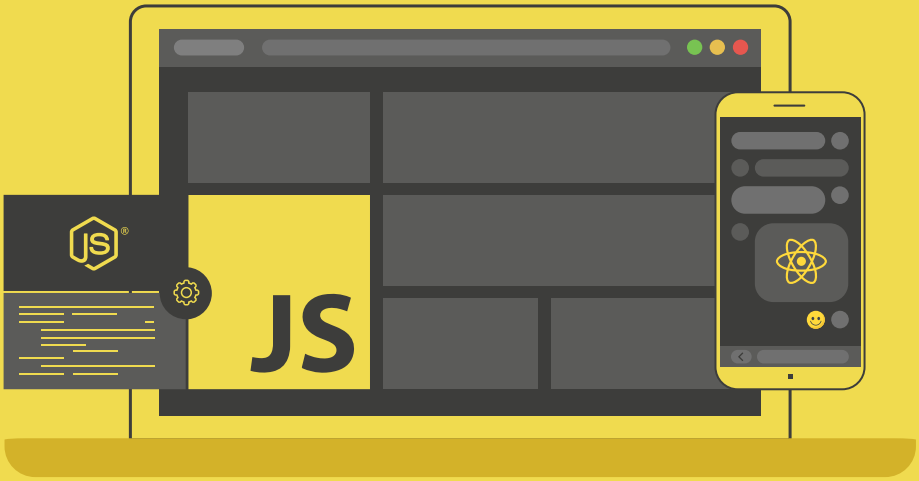
39.5 الخاتمة

اكتسبنا في هذا الفصل فهمًا شاملًا للعمل مع طلبات HTTP في لغة جو باستخدام حزمة net/http؛ إذ بدأنا بتقديم طلب GET إلى خادم HTTP باستخدام كل من دالة http.Get و http.NewRequest مع عميل HTTP الافتراضي؛ ثم وسّعنا بعد ذلك معرفتنا عن طريق إجراء طلب POST مع متن طلب باستخدام bytes.NewReader. كما تعلمنا كيفية تخصيص الطلب عن طريق ضبط ترويسة Content-Type باستخدام التابع Set في حقل ترويسة http.Request. اكتشفنا أهمية ضبط المهلات للتحكم في مدة الطلبات من خلال إنشاء عميل http.Client.

تجدر الإشارة إلى أن حزمة net/http توفر دوال أكثر مما غطيناه في هذا الفصل، على سبيل المثال يمكن استخدام دالة http.Post لتقديم طلبات POST والاستفادة من ميزات مثل إدارة ملفات تعريف الارتباط.

من خلال إتقان هذه المفاهيم، نكون قد زودنا أنفسنا بالمهارات الأساسية للتفاعل مع خوادم HTTP وإنشاء أنواع مختلفة من الطلبات باستخدام لغة جو.

دورة تطوير التطبيقات
باستخدام لغة JavaScript



مميزات الدورة

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

اشترك الآن

40. استخدام الأنواع المعممة Generics

قدمت لغة جو في الإصدار 1.18 ميزةً جديدة تُعرف باسم الأنواع المُعمّمة generic types -أو generics- والتي كانت في قائمة أمنيات مُطوّري هذه اللغة والنوع المُعمّم هو نوع يمكن استخدامه مع أنواع متعددة أخرى.

سابقًا، عندما كنا نرغب في استخدام نوعين مختلفين لنفس المتغير في لغة جو، كنا نحتاج إما لاستخدام واجهة مُعرّفة بأسلوب معين مثل io.Reader، أو استخدام interface{} الذي يسمح باستخدام أي قيمة. المشكلة في أن استخدام `interface{}` يجعل التعامل مع تلك الأنواع صعبًا، لأنه يجب التحويل بين العديد من الأنواع الأخرى المحتملة للتفاعل معها (عليك تحويل القيم بين أنواع مختلفة من البيانات للتعامل معها). على سبيل المثال، إذا كان لدينا قيمة من `interface{}` تمثل عددًا، يجب علينا تحويلها إلى `int` قبل أن تتمكن من إجراء العمليات الحسابية عليها. هذه العملية قد تكون معقدة وتستلزم كتابة الكثير من التعليمات الإضافية للتعامل مع تحويل الأنواع المختلفة، أما الآن وبفضل الأنواع المُعمّمة، يمكننا التفاعل مباشرةً مع الأنواع المختلفة دون الحاجة للتحويل بينها، مما يؤدي إلى شيفرة نظيفة سهلة القراءة.

نُنشئ في هذا الفصل برنامجًا يتفاعل مع مجموعة من البطاقات، إذ سنبدأ بإنشائها بحيث تستخدم `interface{}` للتفاعل مع البطاقات الأخرى، ثم نُحدّثها من أجل استخدام الأنواع المُعمّمة، ثم سنضيف نوعًا ثانيًا من البطاقات باستخدام الأنواع المُعمّمة، ثم سنُحدّثها لتقييد نوعها المُعمّم، بحيث تدعم أنواع البطاقات فقط. نُنشئ أخيرًا دالةً تستخدم البطاقات الخاصة بنا وتدعم الأنواع المُعمّمة.

40.1 المتطلبات الأولية

لمتابعة هذا الفصل التعليمي، سنحتاج إلى:

- إصدار مُثبّت من جو 1.16 أو أعلى، ارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبت لغة جو بحسب نظام تشغيلك.

- فهم قوي لأساسيات لغة جو، مثل المتغيرات والدوال وأنواع البيانـات والشـرائح والحلقـات ...إلخ. وقـد شرحنا كل هذه الأساسيات في فصول سابقة من الكتاب

40.2 التجميعات Collections في لغة جو بدون استخدام الأنواع المعممة

يشير مصطلح التجميعة Collection في هذا السياق إلى حاوية بيانات يمكن أن تحتوي على قيم أو عناصـر متعددة، وهو مفهوم ذو مستوى أعلى يشمل أنواعًـا مختلفـة من هياكـل البيانـات، مثـل المصـفوفات والقـوائم والمجموعات والخرائط maps، مما يسـمح بتخـزين وتنظيم عـدة قيم مرتبطـة معًـا. تـوفر التجميعـات عمليـات وتوابع للوصول إلى العناصر التي تحتويها ومعالجتها وتكرارها. نستخدم مصطلح "التجميعة هنا" لوصف الهيكـل أو الحاوية المستخدمة لتخزين وإدارة أوراق اللعب في البرنامج.

تتمتع لغة جو بميزة قوية تتمثّل في قدرتها على تمثيل العديد من الأنواع بطريقة مرنة باسـتخدام الواجهـات interfaces. يمكن للكثير من الشيفرات المكتوبة بلغة جو أن تعمـل جيـدًا باسـتخدام دوال الواجهـات المتاحـة، وهذا هو أحد الأسباب التي جعلت اللغة موجودةً لفترة طويلة دون دعم الأنواع المُعمّمة.

نُنشـئ في هـذا الفصـل برنامجًـا يحـاكي عمليـة الحصـول على بطاقـة لعب عشـوائية `PlayingCard` من مجموعة بطاقات `Deck`. سنستخدم في هذا القسم `interface{}` للسماح للدستة `Deck` بالتفاعل مع أي نوع من البطاقات. نُعدّل لاحقًا البرنـامج لاسـتخدام الأنـواع المُعمّمـة لنتمكن من فهم الفـروق بينهمـا والتعـرف على الحالات التي تكون فيها الأنواع المُعمّمة خيارًا أفضل من غيرها.

يمكن تصنيف أنظمة النوع type systems في لغات البرمجة عمومًا إلى فئتين مختلفتين: النـوع typing والتحقق من النوع type checking. تُشير الفئة الأولى إلى كيفية التعامل مع الأنواع في اللغـة وكيفيـة تعريـف وتمثيل أنواع البيانات المختلفة.

هناك نوعان رئيسيان من أنظمة "النوع" هنا النوع القوي strong والنوع الضـعيف weak وكـذلك الثـابت static والديناميكي dynamic. تُطبّق قواعد صارمة في النوع القوي تضمن توافق الأنواع وعـدم وجـود تضـارب بينها. على سبيل المثال، لا يمكن تخزين قيمة من نوع بيانات ما في متغير من نوع بيانات آخر (لا يمكن تخـزين قيمة `int` في متغير `string`) إلا إذا كانت هناك توافقية بينهما. أمـا في النـوع الضـعيف، فقـد تسـمح اللغـة بتحويل تلقائي للأنواع من خلال عمليات تحويل ضمنية.

أما بالنسبة "للتحقق من النوع"، فهذا يشير إلى كيفية التحقق من صحة استخدام الأنواع في البرنامج. هناك نوعـان رئيسـيان من أنظمـة التحقـق من النـوع: التحقـق الثـابت statically-checked والتحقـق الـديناميكي dynamically-checked. تُفحص الأنواع في التحقق الثابت خلال عملية التصريف ويُفحص توافقهـا وصـحتها والتأكد من عدم وجود أخطاء في النوع قبل تنفيذ البرنامج. أما في التحقق الديناميكي، فـإن الفحص يجـري أثنـاء تنفيذ البرنامج نفسه، ويمكن للأنواع أن تتغير وتحول ديناميكيًا خلال تنفيذ البرنامج.

تنتمي لغة جو عمومًا إلى فئة اللغات ذات النوع القوي والتحقق الثابت، إذ تستخدم قواعد صارمة للتحقق من توافق الأنواع وتتحقق من صحتها خلال عملية التصريف، مما يساعد في تجنب الأخطاء الناتجة عن تعامل غير صحيح مع الأنواع في البرنامج، لكن يترتب على ذلك بعض القيود على البرامج، لأنه يجب أن تُعرّف الأنواع التي تنوي استخدامها قبل تصريف البرنامج. يمكن التعامل مع هذا من خلال استخدام النوع `interface{}`، إذ يعمل نوع `interface{}` مع أي قيمة.

لبدء إنشاء البرنامج باستخدام `interface{}` لتمثيل البطاقات، نحتاج إلى مجلد للاحتفاظ بمجلد البرنامج فيه، وليكن باسم `projects`. أنشئ المجلد وانتقل إليه:

```
$ mkdir projects
$ cd projects
```

نُنشئ الآن مجلدًا للمشروع، وليكن باسم `generics` ثم ننتقل إليه:

```
$ mkdir generics
$ cd generics
```

من داخل هذا المجلد، نفتح الملف main.go باستخدام محرر نانو nano أو أي محرر آخر تريده:

```
$ nano main.go
```

نضع مقتطف الشيفرة التالي داخل هذا الملف، إذ نُصرّح عن الحزمة `package` لإخبار المُصرّف أن يحوّل البرنامج إلى ملف ثنائي لتتمكّن من تشغيله مباشرةً. نستورد أيضًا الحزم اللازمة عن طريق تعليمة `import`:

```
package main
import (
  "fmt"
  "math/rand"
  "os"
  "time"
)
```

نُعرّف النوع `PlayingCard` والدوال والتوابع المرتبطة به:

```
...
type PlayingCard struct {
  Suit string
  Rank string
}
```

```
func NewPlayingCard(suit string, card string) *PlayingCard {
  return &PlayingCard{Suit: suit, Rank: card}
}
func (pc *PlayingCard) String() string {
  return fmt.Sprintf("%s of %s", pc.Rank, pc.Suit)
}
```

عرّفنا بنية بيانات تُسمى PlayingCard مع الخصائص نوع البطاقة Suit وترتيبها Rank لتمثيل مجموعة ورق اللعب Deck المكونة من 52 بطاقة. يكون Suit أحد القيم التالية: Diamonds أو Hearts أو Clubs أو Spades. أما Rank يكون أما A أو 2 أو 3، وهكذا حتى K.

عرّفنا أيضًا دالة NewPlayingCard لتكون باني constructor لبنية البيانات PlayingCard، وتابع String ليُرجع ترتيب ونوع البطاقة باستخدام fmt.Sprintf.

ننشئ الآن النوع Deck بالدوال AddCard و RandomCard، إضافةً إلى الدالة NewPlayingCardDeck لإنشاء *Deck مملوء بجميع بطاقات اللعبة التي عددها 52.

```
...
type Deck struct {
  cards []interface{}
}
func NewPlayingCardDeck() *Deck {
  suits := []string{"Diamonds", "Hearts", "Clubs", "Spades"}
  ranks := []string{"A", "2", "3", "4", "5", "6", "7", "8", "9", "10",
"J", "Q", "K"}
  deck := &Deck{}
  for _, suit := range suits {
        for _, rank := range ranks {
              deck.AddCard(NewPlayingCard(suit, rank))
        }
  }
  return deck
}
func (d *Deck) AddCard(card interface{}) {
  d.cards = append(d.cards, card)
}
func (d *Deck) RandomCard() interface{} {
```

```
    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    cardIdx := r.Intn(len(d.cards))
    return d.cards[cardIdx]
}
```

في البنية `Deck` المُعرّفة أعلاه، أنشأنا حقلًا يُسمّى `cards` لاحتواء مجموعة ورق اللعب. نظرًا لأننا نرغب في أن تكون مجموعة البطاقات قادرة على استيعاب أنواع متعددة مختلفة من البطاقات، فلا يمكن تعريفها فقط على أنها `*PlayingCard`. لذا عرّفناها على أنها `[]interface{}` حتى نتمكن من استيعاب أي نوع من البطاقات التي قد نُنشئها مستقبلًا. بالإضافة إلى الحقل `[]interface{}` في `Deck`، أنشأنا التابع `AddCard` الذي يقبل قيمة من نفس نوع `{}interface` لإضافة بطاقة إلى حقل البطاقات في `Deck`.

أنشأنا أيضًا التابع `RandomCard` الذي يُرجع بطاقةً عشوائية من `cards` في `Deck`. يستخدم هذا التابع حزمة `math/rand` لإنشاء رقم عشوائي بين `0` وعدد البطاقات في المجموعة. يُنشئ سطر `rand.New` مولد رقم عشوائي جديد باستخدام الوقت الحالي مثل مصدر للعشوائية، وإلا فإن الرقم العشوائي يمكن أن يكون نفسه في كل مرة. يستخدم السطر `r.Intn(len(d.cards))` مولد الأرقام العشوائية لإنشاء قيمة صحيحة بين `0` والعدد المُقدم. نظرًا لأن التابع `Intn` لا يشمل قيمة المعامل ضمن نطاق الأرقام الممكنة، فلا حاجة لطرح `1` من الطول لمعالجة فكرة البدء من `0` (إذا كان لدينا مجموعة ورق لعب تحتوي على `10` عناصر، فالأرقام الممكنة ستكون من `0` إلى `9`). لدينا أيضًا `RandomCard` تُرجع قيمة البطاقة في الفهرس المحدد بالرقم العشوائي.

> كن حذرًا في اختيار مولد الأرقام العشوائية الذي تستخدمه في برامجك.فحزمة math/rand ليست آمنة من الناحية التشفيرية ولا ينبغي استخدامها في البرامج التي تتعلق بالأمان. توفر حزمة crypto/rand مولد أرقام عشوائية يمكن استخدامه لهذه الأغراض.

بالنسبة للدالة `NewPlayingCardDeck` فهي تُرجع القيمة `*Deck` مملوءة بجميع البطاقات الموجودة في دستة بطاقات اللعب. نستخدم شريحتين، واحدة تحتوي على جميع الأشكال المتاحة وأخرى تحتوي على جميع الرتب المتاحة، ثم نكرر فوق كل قيمة لإنشاء `*PlayingCard` جديدة لكل تركيبة قبل إضافتها إلى المجموعة باستخدام `AddCard`. تُرجع القيمة بمجرد إنشاء بطاقات مجموعة ورق اللعب.

بعد إعداد `Deck` و `PlayingCard`، يمكننا إنشاء الدالة `main` لاستخدامها لسحب البطاقات:

```
...
func main() {
    deck := NewPlayingCardDeck()
    fmt.Printf("--- drawing playing card ---\n")
    card := deck.RandomCard()
```

```
    fmt.Printf("drew card: %s\n", card)
    playingCard, ok := card.(*PlayingCard)
    if !ok {
        fmt.Printf("card received wasn't a playing card!")
        os.Exit(1)
    }
    fmt.Printf("card suit: %s\n", playingCard.Suit)
    fmt.Printf("card rank: %s\n", playingCard.Rank)
}
```

أنشأنا بدايةً مجموعةً جديدة من البطاقات باستخدام الدالة `NewPlayingCardDeck` ووضعناها في المتغير `deck`، ثم استخدمنا `fmt.Printf` لطباعة جملة تدل على أننا نسحب بطاقة. نستخدم التابع `RandomCard` من `deck` للحصول على بطاقة عشوائية من المجموعة. نستخدم بعد ذلك `fmt.Printf` مرةً أخرى لطباعة البطاقة التي سحبناها من المجموعة.

نظرًا لأن نوع المتغير `card` هو `interface{}`، سنحتاج إلى استخدام توكيد النوع type assertion للحصول على مرجع إلى البطاقة بنوعها الأصلي `*PlayingCard`. إذا كان النوع في المتغير `card` ليس نوع `*PlayingCard`، الذي يجب أن يكون عليه وفقًا لطريقة كتابة البرنامج الحالية، ستكون قيمة `ok` تساوي `false` وسيطبع البرنامج رسالة خطأ باستخدام `fmt.Printf` ويخرج مع شيفرة خطأ `1` باستخدام `os.Exit`. إذا كان النوع `*PlayingCard` سيطبع البرنامج قيمتي `Suit` و `Rank` من `playingCard` باستخدام `fmt.Printf`.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

يُفترض أن نرى في الخرج بطاقة مُختارة عشوائيًا من المجموعة، إضافةً إلى نوع البطاقة وترتيبها:

```
--- drawing playing card ---
drew card: Q of Diamonds
card suit: Diamonds
card rank: Q
```

قد تكون النتيجة المطبوعة مختلفةً عن النتيجة المعروضة أعلاه، لأن البطاقة مسحوبة عشوائيًا من المجموعة، لكن يجب أن تكون مشابهة. يُطبع السطر الأول قبل سحب البطاقة العشوائية من المجموعة، ثم يُطبع السطر الثاني بمجرد سحب البطاقة. يمكن رؤية خرج البطاقة باستخدام القيمة المُرجعة من التابع

String من PlayingCard. يمكننا أخيرًا رؤية سطري النوع والرتبة المطبوعة بعد تحويل قيمة interface{} المستلمة إلى قيمة *PlayingCard مع الوصول إلى حقول Suit و Rank.

أنشأنا في هذا القسم مجموعة Deck تستخدم قيم interface{} لتخزين والتفاعل مع أي قيمة، وأيضًا نوع PlayingCard ليكون بمثابة البطاقات داخل هذه الدستة Deck، ثم استخدمنا Deck و PlayingCard لاختيار بطاقة عشوائية من المجموعة وطباعة معلومات حول تلك البطاقة.

للوصول إلى معلومات محددة تتعلق بقيمة *PlayingCard التي سُحبت، كُنّا بحاجة لعمل إضافي يتجلى بتحويل النوع interface{} إلى النوع *PlayingCard مع الوصول إلى حقول Suit و Rank. ستعمل الأمور في المجموعة Deck جيدًا بهذه الطريقة، ولكن قد ينتج أخطاء إذا أُضيفت قيمة أخرى غير *PlayingCard إلى المجموعة Deck.

سنُعدّل في القسم التالي المجموعة Deck، للاستفادة من خصائص الأنواع القوية والتحقق الثابت للأنواع في لغة جو، مع الاحتفاظ بالمرونة في قبول قيم interface{}، وذلك من خلال استخدام الأنواع المعممة.

40.3 التجميعات في لغة جو مع استخدام الأنواع المعممة

أنشأنا في القسم السابق تجميعةً باستخدام شريحة من أنواع interface{}، ولكن كان علينا إتمام عمل إضافي لاستخدام تلك القيم يتجلى بتحويل القيم من نوع interface{} إلى النوع الفعلي لتلك القيم. يمكننا باستخدام الأنواع المعممة إنشاء معامل واحد أو أكثر للأنواع، والتي تعمل تقريبًا مثل معاملات الدالة، ولكن يمكن أن تحتوي على أنواع بدلًا من بيانات. توفر الأنواع المعممة بهذا الأسلوب طريقةً لاستبدال نوع مختلف من معاملات الأنواع في كل مرة يجري فيها استخدام النوع المعمم. هنا يأتي اسم الأنواع المعممة؛ فبما أن النوع المعمم يمكن استخدامه مع أنواع متعددة، وليس فقط نوع محدد مثل io.Reader أو interface{}، يكون عامًا بما يكفي ليناسب عدة حالات استخدام.

نُعدّل في هذا القسم مجموعة ورق اللعب Deck لتكون من نوع معمم يمكنه استخدام أي نوع محدد للبطاقة عند إنشاء نسخة من Deck بدلًا من استخدام interface{}.

نفتح ملف main.go ونحذف استيراد حزمة os:

```
package main
import (
  "fmt"
  "math/rand"
  // "os" حذف استيراد
  "time"
)
```

لن نحتاج بعد الآن إلى استخدام دالة `os.Exit`، لذا فمن الآمن حذف هذا الاستيراد. نُعدّل البنية `Deck` الآن لتكون نوعًا معممًا:

```
...
type Deck[C any] struct {
  cards []C
}
```

يقدم هذا التحديث الصيغة الجديدة لتعريف البنية `Deck`، بحيث نجعلها تحقق مفهوم النوع المُعمم، وذلك من خلال استخدام مفهوم "الموضع المؤقت Placeholder" أو "معاملات النوع Type parameters". يمكننا التفكير في هذه المعاملات بطريقة مشابهة للمعاملات التي نُضمّنُها في دالة؛ فعند استدعاء دالة، تُقدم قيمًا لكل معامل في الدالة. وهنا أيضًا، عند إنشاء قيمة من النوع المعمم، نُقدّم أنواعًا لمعاملات الأنواع.

نلاحظ أننا أضفنا عبارة داخل قوسين مربعين `[]` بعد اسم البنية `Deck` لتحديد معامل أو أكثر من هذه المعاملات للبنية ونحتاج في حالتنا للنوع `Deck` إلى معامل نوع واحد فقط، يُطلق عليه اسم `C`، لتمثيل نوع البطاقات في المجموعة `deck`. من خلال كتابتنا `C any` نكون قد صرّحنا عن معامل نوع باسم `C` يمكننا استخدامه في البنية `struct` ليُمثّل أي نوع (`any`). ينوب `any` عن الأنواع المختلفة، ويُعرف نوعه وقت تمرير قيمته، فإذا كانت القيمة المُمررة سلسلة يكون سلسلة، وإذا كانت عددًا صحيحًا يكون صحيحًا.

يُعد النوع `any` في حقيقة الأمر اسمًا بديلًا للنوع `interface{}`، وهذا ما يجعل الأنواع المعممة أكثر سهولة للقراءة، فنحن بغنى عن كتابة `C interface{}` ونكتب `C any` فقط. يحتاج `deck` الخاص بنا إلى نوع معمم واحد فقط لتمثيل البطاقات، ولكن إذا كنا نحتاج إلى أنواع معممة إضافية، فيمكن إضافتها من خلال فصلها بفواصل، مثل `C any, F any`. يمكن أن يكون اسم المعامل المُستخدم لمعاملات النوع أي شيء نرغب فيه إذا لم يكن محجوزًا، ولكنها عادةً قصيرة وتبدأ بحرف كبير.

عدّلنا أيضًا نوع الشريحة `cards` في البنية `Deck` ليكون من النوع `C`. عند استخدام مفهوم الأنواع المعممة، يمكننا استخدام معاملات النوع لتنوب عن أي نوع، أي يمكننا وضعها في نفس الأماكن التي نضع فيها أنواع البيانات عادةً. في حالتنا، نريد أن يمثل المعامل `C` كل بطاقة في الشريحة، لذا نضع اسم الشريحة ثم `[]` ثم المعامل `C` الذي سينوب عن النوع.

نُعدّل التابع `AddCard` لاستخدام النوع المعمّم، وسنتخطى تعديل دالة `NewPlayingCardDeck` حاليًا:

```
...
func (d *Deck[C]) AddCard(card C) {
  d.cards = append(d.cards, card)
}
```

تضمّن التحديث الخاص بالتابع `AddCard` في `Deck`، إضافة المعامل المُعمّم `[C]` إلى المستقبل الخـاص بالتابع. يُخبر هذا لغة جو باسم المعامل المُعمّم الذي سنستخدمه في أماكن أخرى في تصـريح التـابع، ويـتيح لنـا ذلك معرفة النوع المُمرّر مثل قيمة إلى معامل النوع `C` عند استخدام التابع. يتبع هذا النـوع من الكتابـة بـأقواس معقوفة نفس طريقة التصريح عن بنية `struct`، إذ يُعرّف معامل النوع داخل الأقواس المعقوفة بعد اسم الدالة والمستقبل. لا نحتاج في حالتنا إلى تحديد أي قيود لأنها مُحددة فعلًا في تصريح `Deck`. عدّلنا أيضًا معامل الدالة `card` لاستخدام معامل النوع `C` بدلًا من النوع الأصلي `interface{}`، وهذا يتيح للدالة استخدام النوع `C` الذي سيُعرف ما هو لاحقًا.

بعد تعديل التابع `AddCard`، نُعدّل التابع `RandomCard` لاستخدام النوع المعمم `C` أيضًا:

```
...
func (d *Deck[C]) RandomCard() C {
  r := rand.New(rand.NewSource(time.Now().UnixNano()))
  cardIdx := r.Intn(len(d.cards))
  return d.cards[cardIdx]
}
```

بدلًا من استخدام نـوع معمّم `C` مثـل معامـل دالـة، عـدّلنا التـابع `RandomCard()` ليعيـد قيمـةً من النـوع المُعمّم `C` بدلًا من معامل الدالة `interface{}`. هذا يعني أنه عند استدعاء هذا التابع، سنحصل على قيمة من النوع `C` المحدد، بدلًا من الحاجة إلى تحويل القيمة إلى نوع محدد بواسطة "تأكيـدات النـوع type assertion". عدّلنا أيضًا المستقبل الخاص بالدالة ليتضمن `[C]`، وهو ما يخـبر لغـة جـو بـأن الدالـة هي جـزء من `Deck` الـتي تستخدم النوع المعمم `C` مثل معامل. لا نحتاج إلى أي تحديثات أخرى في الدالة. حقل `cards` في `Deck` مُعـدّل فعلًا في تصريح البنية `struct` ليكون من النوع `[]C`، مما يعني أنه عندما يعيد التـابع قيمـة من `cards`، فإنـه يعيد قيمةً من النوع `C` المحدد، وبالتالي لا حاجة لتحويل النوع بعد ذلك.

بعد تعديلنا للنوع `Deck` لاستخدام الأنواع المعممة، نعود للدالة `NewPlayingCardDeck` لجعلها تستخدم النوع المعمم `Deck` مع الأنواع `*PlayingCard`:

```
...
func NewPlayingCardDeck() *Deck[*PlayingCard] {

  suits := []string{"Diamonds", "Hearts", "Clubs", "Spades"}
  ranks := []string{"A", "2", "3", "4", "5", "6", "7", "8", "9", "10",
"J", "Q", "K"}
  deck := &Deck[*PlayingCard]{}
  for _, suit := range suits {
```

```
            for _, rank := range ranks {
                  deck.AddCard(NewPlayingCard(suit, rank))
            }
      }
      return deck
}
...
```

بقيت معظم الشيفرة في `NewPlayingCardDeck` كما هي، باستثناء بعض التغييرات البسيطة، ولكن الآن نستخدم إصدارًا مُعمّمًا من `Deck`، ويجب علينا تحديد النوع الذي نرغب في استخدامه مع `C` عند استخدام `Deck`. نفعل ذلك عن طريق الإشارة إلى نوع `Deck` بالطريقة المعتادة، سواء كان `Deck` نفسه أو مرجعًا مثل `*Deck`، ثم تقديم النوع الذي يجب استبدال `C` به باستخدام نفس الأقواس المعقوفة التي استخدمناها عند التصريح عن معاملات النوع في البداية. بمعنى آخر، عندما نُعرّف متغير من نوع `Deck` المعمّم، يجب علينا تحديد النوع الفعلي للمعامل `C`؛ فإذا كانت `Deck` من النوع `*Deck`، يجب أن نُوفّر النوع الفعلي الذي يجب استبدال `C` به في الأقواس المعقوفة مثل `[*PlayingCard]`. هكذا نحصل على نسخة محددة من `Deck` لنوع البطاقات التي ترغب في استخدامها، والتي في هذه الحالة هي `*PlayingCard`.

يمكن تشبيه هذه العملية لإرسال قيمة لمعامل دالة عند استدعاء الدالة؛ فعند استخدامنا لنوع `Deck` المعمّم، فإننا نمرر النوع الذي نرغب في استخدامه مثل معامل لهذا النوع.

بالنسبة لنوع القيمة المُعادة في `NewPlayingCardDeck`، نستمر في استخدام `*Deck` كما فعلت من قبل، لكن هذه المرة، يجب أن تتضمن الأقواس المعقوفة و `*PlayingCard` أيضًا؛ فمن خلال تقديم `[*PlayingCard]` لمعامل النوع، فأنت تقول أنك ترغب في استخدام النوع `*PlayingCard` في تصريح `Deck` والتوابع الخاصة به ليحل محل قيمة `C`، وهذا يعني أن نوع حقل `cards` في `Deck` يتغير فعليًا من `[]C` إلى `[]*PlayingCard`.

بمعنى آخر، عندما ننشئ نسخةً جديدةً من `Deck` باستخدام `NewPlayingCardDeck`، فإننا نُنشئ `Deck` يمكنه تخزين أي نوع محدد من `*PlayingCard`. هذا يتيح لنا استخدام مجموعة متنوعة من أنواع البطاقات في `Deck` بدلًا من الاعتماد على `interface{}`. بالتالي يمكننا الآن التعامل مباشرةً مع البطاقات بنوعها الفعلي بدلًا من الحاجة إلى استبدال الأنواع والتحويلات التي كنا نستخدمها مع `interface{}` في الإصدار السابق من `Deck`.

بالمثل، عند إنشاء نسخة جديدة من `Deck`، يجب علينا أيضًا توفير النوع الذي يحل محل `C`. نستخدم عادةً `&Deck{}` لإنشاء مرجع جديد إلى `Deck`، ولكن بدلًا من ذلك يجب علينا تضمين النوع داخل الأقواس المعقوفة للحصول على `&[*PlayingCard]{}`.

الآن بعد أن عدّلنا الأنواع الخاصة بنا لاستخدام الأنواع المُعمّمة، نُعدّل الدالة `main` للاستفادة منها:

```
...
func main() {
  deck := NewPlayingCardDeck()
  fmt.Printf("--- drawing playing card ---\n")
  playingCard := deck.RandomCard()
  fmt.Printf("drew card: %s\n", playingCard)
  // Code removed
  fmt.Printf("card suit: %s\n", playingCard.Suit)
  fmt.Printf("card rank: %s\n", playingCard.Rank)
}
```

أزلنا هذه المرة بعض التعليمات، لأنه لم تعد هناك حاجة لتأكيد قيمة `interface{}` على أنها `*PlayingCard`. عندما عدّلنا التابع `RandomCard` في `Deck` لإعادة النوع `C` و `NewPlayingCardDeck` لإعادة `*Deck[*PlayingCard]`، نكون قد عدّلنا التابع `RandomCard` بجعله يعيد `*PlayingCard` بدلًا من `interface{}`. هذا يعني أن نوع `playingCard` هو `*PlayingCard` أيضًا بدلًا من `interface{}` ويمكننا الوصول إلى حقول `Suit` و `Rank` مباشرةً.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

سنشاهد خرجًا مشابهًا لما سبق، ولكن البطاقة المسحوبة قد تكون مختلفة:

```
--- drawing playing card ---
drew card: 8 of Hearts
card suit: Hearts
card rank: 8
```

على الرغم من أن الخرج هو نفسه مثل الإصدار السابق من البرنامج عندما استخدمنا `interface{}`، إلا أن الشيفرة أنظف وتتجنب الأخطاء المحتملة. لم يعد هناك حاجةٌ لتأكيد النوع `*PlayingCard`، مما يُجنّبنا التعامل مع الأخطاء الإضافية، كما أنّه من خلال تحديد أن نسخة `Deck` يمكن أن تحتوي فقط على `*PlayingCard`، لن يكون هناك أي احتمال لوجود قيمة أخرى غير `*PlayingCard` تُضاف إلى مجموعة البطاقات.

عدّلنا في هذا القسم البنية `Deck` لتكون نوعًا مُعمّمًا، مما يوفر مزيدًا من السيطرة على أنواع البطاقات التي يمكن أن تحتويها كل نسخة من مجموعة أوراق اللعب. عدّلنا أيضًا التابعين `AddCard` و `RandomCard` إما لقبول وسيط مُعمم أو لإرجاع قيمة مُعممة. عدّلنا أيضًا `NewPlayingCardDeck` لإرجاع `*Deck` يحتوي على بطاقات `*PlayingCard`. أخيرًا أزلنا التعامل مع الأخطاء في الدالة `main` لأنه لم يعد هناك حاجة لذلك.

الآن بعد تعديل `Deck` ليكون مُعممًا، يمكننا استخدامه لاحتواء أي نوع من البطاقات التي نرغب. سنستفيد في القسم التالي من هذه المرونة من خلال إضافة نوع جديد من البطاقات إلى البرنامج.

40.4 استخدام أنواع مختلفة مع الأنواع المعممة

يمكننا -بإنشاء نوع مُعمّم مثل النوع `Deck`- استخدامه مع أي نوع آخر؛ فعندما نُنشئ نسخةً من `Deck`، ونرغب باستخدامها مع أنواع `*PlayingCard`، فإن الشيء الوحيد الذي نحتاجه هو تحديد هذا النوع عند إنشاء القيمة، وإذا أردنا استخدام نوع آخر، نُبدّل نوع `*PlayingCard` بالنوع الجديد الذي نرغب باستخدامه.

نُنشئ في هذا القسم بنية بيانات جديدة تُسمى `TradingCard` لتمثيل نوع مختلف من البطاقات، ثم نُعدّل البرنامج لإنشاء `Deck` يحتوي على عدة `*TradingCard`.

لإنشاء النوع `TradingCard`، نفتح ملف main.go مرةً أخرى ونضيف التعريف التالي:

```
...
import (
   ...
)
type TradingCard struct {
   CollectableName string
}
func NewTradingCard(collectableName string) *TradingCard {
   return &TradingCard{CollectableName: collectableName}
}
func (tc *TradingCard) String() string {
   return tc.CollectableName
}
```

النوع `TradingCard` مشابه لنوع `PlayingCard`، ولكن بدلًا من وجود حقول `Suit` و `Rank`، يحتوي على حقل `CollectableName` لتتبع اسم بطاقة التداول، كما يتضمن دالة الباني `NewTradingCard` والتابع `String`، أي بطريقة مشابهة للنوع `PlayingCard`.

نُنشئ الآن الدالة البانية `NewTradingCardDeck` لإنشاء `Deck` مُعبأة بقيم `*TradingCards`:

```
...
func NewPlayingCardDeck() *Deck[*PlayingCard] {
   ...
}
func NewTradingCardDeck() *Deck[*TradingCard] {
   collectables := []string{"Sammy", "Droplets", "Spaces", "App
Platform"}
   deck := &Deck[*TradingCard]{}
   for _, collectable := range collectables {
         deck.AddCard(NewTradingCard(collectable))
   }
   return deck
}
```

عند إنشاء أو إرجاع `*Deck` في هذه الحالة، فإننا نُبدّل قيم `*PlayingCard` بقيم `*TradingCard`، وهذا هو الأمر الوحيد الذي نحتاج إلى تغييره في `Deck`. هناك مجموعة من البطاقات الخاصة، والتي يمكننا المرور عليها لإضافة كل `*TradingCard` إلى `Deck`. يعمل التابع `AddCard` في `Deck` بنفس الطريقة كما في السابق، لكن هذه المرة يقبل قيمة `*TradingCard` من `NewTradingCard`. إذا حاولنا تمرير قيمة من `NewPlayingCard`، سيُعطي المُصرّف خطأ لأنه يتوقع `*TradingCard` وليس `*PlayingCard`.

نُعدّل الدالة `main` لإنشاء `Deck` جديدة من أجل عدة `*TradingCard`، وسحب بطاقة عشوائية باستخدام `RandomCard`، وطباعة معلومات البطاقة:

```
...
func main() {
   playingDeck := NewPlayingCardDeck()
   tradingDeck := NewTradingCardDeck()
   fmt.Printf("--- drawing playing card ---\n")
   playingCard := playingDeck.RandomCard()
   ...
   fmt.Printf("card rank: %s\n", playingCard.Rank)
   fmt.Printf("--- drawing trading card ---\n")
   tradingCard := tradingDeck.RandomCard()
   fmt.Printf("drew card: %s\n", tradingCard)
   fmt.Printf("card collectable name: %s\n",
tradingCard.CollectableName)
}
```

أنشـأنا هنـا مجموعـة جديـدة من بطاقـات التـداول باسـتخدام NewTradingCardDeck وخزّناهـا في tradingDeck. ونظـرًا لأننـا لا نـزال نسـتخدم نفس النـوع Deck كمـا كـان من قبـل، فيمكننـا اسـتخدام RandomCard للحصول على بطاقة تداول عشوائية من Deck وطباعـة البطاقـة. كمـا يمكننـا أيضًـا الإشـارة إلى وطباعة حقل CollectableName مباشرةً من tradingCard لأن Deck المُعمّم الذي نستخدمه قد عـرّف C على أنه *TradingCard.

يُظهر هذا التحديث أيضًا قيمة استخدام الأنواع المُعمّمة. فمن أجل دعم نوع بطاقة جديـد تمامًـا، لم نحتـاج إلى تغيير Deck إطلاقًا، فمن خلال معاملات النوع في Deck تمكّننا من تحديد نوع البطاقـة الـذي نريـده، وبـدءًا من هذه النقطة فصاعدًا، يُستخدم النوع *TradingCard بدلًا من النوع *PlayingCard في أية تفاعلات مع قيم Deck.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر go run:

```
$ go run main.go
```

ليكون الخرج على النحو التالي:

```
--- drawing playing card ---
drew card: Q of Diamonds
card suit: Diamonds
card rank: Q
--- drawing trading card ---
drew card: App Platform
card collectable name: App Platform
```

يجب أن نرى نتيجةً مشابهة للنتيجة الموضحة أعلاه (مع بطاقـات مختلفـة بسـبب العشـوائية) بعـد انتهـاء تشغيل البرنامج. تظهر بطاقتين مسحوبتين: البطاقة الأصلية للّعب وبطاقة التداول الجديدة المُضافة.

أضفنا في هذا القسم النوع TradingCard الجديد لتمثيل نوع بطاقـة مختلـف عن نـوع البطاقـة الأصـلية PlayingCard. بمجرد إضافة نوع البطاقـة TradingCard،أنشـأنا الدالـة NewTradingCardDeck لإنشـاء وملء مجموعة ورق اللعب ببطاقات التداول. أخيرًا عدّلنا الدالة main لاستخدام مجموعة ورق التـداول الجديـدة وطباعة معلومات مُتعلقة بالبطاقة العشوائية المستخرجة.

بصرف النظر عن إنشاء دالة جديدة NewTradingCardDeck لملء الدسـتة ببطاقـات مختلفـة، لم نحتـاج إلى إجراء أي تحـديثات أخـرى على مجموعـة ورق اللعب لـدعم نـوع بطاقـة جديـد تمامًـا. هـذه هي قـوة الأنـواع المُعمّمة؛ إذ يمكننا كتابة الشيفرة مرةً واحدةً فقط واستخدامها متى أردنـا لأنـواع مختلفـة من البيانـات المماثلـة أيضًا. هناك مشكلة واحدة في Deck الحالي، وهي أنه يمكن استخدامه لأي نوع، بسبب التصريح C any الـذي

لدينا. قد يكون هذا هو ما نريده حتى نتمكن من إنشاء مجموعة ورق لعب للقيم `int` بكتابة `&Deck[int]{}`، ولكن إذا كنا نرغب في أن يحتوي `Deck` فقط على بطاقات، سنحتاج إلى طريقة لتقييد أنواع البيانات المسموح بها مع `C`.

40.5 القيود على الأنواع المعممة

غالبًا لا نرغب أو نحتاج إلى أي قيود على الأنواع المستخدمة في الأنواع المُعمّمة لأننا قد لا نهتم بنوع البيانات المُستخدمة، لكن قد نحتاج في أحيان أخرى إلى القدرة على تقييد الأنواع المستخدمة بواسطة الأنواع المُعمّمة. على سبيل المثال، إذا كُنّا نُنشئ نوعًا مُعمّمًا `Sorter`، قد نرغب في تقييد أنواعه المُعمّمة لتلك التي تحتوي على تابع `Compare` (أي تقييد الأنواع المستخدمة معه، بحيث تكون فقط الأنواع القابلة للمقارنة)، حتى يتمكن `Sorter` من مقارنة العناصر التي يحتوي عليها. بدون هذا القيد، فقد لا تحتوي القيم على التابع `Compare`، ولن يعرف `Sorter` كيفية مقارنتها.

نُنشئ في هذا القسم واجهةً جديدة تسمى `Card`، ثم نُعدّل `Deck` للسماح فقط بإضافة أنواع `Card`.

لتطبيق التحديثات، افتح ملف main.go وأضِف الواجهة `Card`:

```
...
import (
...
)
type Card interface {
  fmt.Stringer
  Name() string
}
```

واجهة `Card` مُعرّفة بنفس طريقة تعريف أي واجهة أخرى في لغة جو؛ وليس هناك متطلبات خاصة لاستخدامها مع الأنواع المُعمّمة. نحن نقول في واجهة `Card` هذه، أنه لكي نعد شيئًا ما بطاقة `Card`، يجب أن يُحقق النوع fmt.Stringer (يجب أن يحتوي على تابع `String` الذي تحتويه البطاقات فعلًا)، ويجب أيضًا أن يحتوي على تابع `Name` الذي يعيد قيمة من نوع `string`.

نُعدّل الآن أنواع `TradingCard` و `PlayingCard` لإضافة التابع `Name` الجديد، إضافةً إلى التابع `String` الموجود فعلًا، لكي نستوفي شروط تحقيق الواجهة `Card`:

```
...
type TradingCard struct {
  ...
```

```
}
...
func (tc *TradingCard) Name() string {
  return tc.String()
}
...
type PlayingCard struct {
  ...
}
...
func (pc *PlayingCard) Name() string {
  return pc.String()
}
```

يحتوي كُلًا من `TradingCard` و `PlayingCard` فعليًا على تابع `String` الذي يُحقق الواجهة `fmt.Stringer`. لذا، لتحقيق الواجهة `Card`، نحتاج فقط إلى إضافة التابع الجديد `Name`. بما أن `fmt.Stringer` يمكنها إرجاع أسماء البطاقات، يمكننا ببساطة إرجاع نتيجة التابع `String` في التابع `Name`.

نُعدّل الآن `Deck`، بحيث يُسمح فقط باستخدام أنواع `Card` مع `C`:

```
...
type Deck[C Card] struct {
  cards []C
}
```

كان لدينا `C any` قبل هذا التحديث مثل قيد نوع type constraint والمعروف أيضًا بتقييد النوع type restriction، وهو ليس قيدًا صارمًا. بما أن `any` هو اسم `interface{}` البديل (له نفس المعنى)، فقد سمح باستخدام أي نوع في جو بمثابة قيمة للمعامل `C`. الآن بعدما عوّضنا `any` بالواجهة `Card` الجديدة، سيتحقق المُصرّف في جو أن أي نوع مستخدم مع `C` يُحقق واجهة `Card` عند تصريف البرنامج.

يمكننا الآن بعد وضع هذا القيد، استخدام أي توابع تقدمها الواجهة `Card` داخل توابع النوع `Deck`. إذا أردنا من `RandomCard` طباعة اسم البطاقة المسحوبة، ستتمكن من الوصول إلى التابع `Name` لأنها جزء من الواجهة `Card`. سنرى ذلك في الجزء التالي.

هذه التحديثات القليلة هي التحديثات الوحيدة التي نحتاج إليها لتقييد النوع `Deck`، بحيث يستخدم فقط لقيم `Card`.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج:

```
--- drawing playing card ---
drew card: 5 of Clubs
card suit: Clubs
card rank: 5
--- drawing trading card ---
drew card: Droplets
card collectable name: Droplets
```

نلاحظ أن الناتج لم يتغير من حيث الشكل العام، فنحن لم نُعدّل بالوظيفة الرئيسية للبرنامج؛ وضعنا فقط قيودًا على الأنواع، إذ أضفنا في هذا الجزء الواجهة `Card` الجديدة وعدّلنا كلًا من `TradingCard` و `PlayingCard` لتحقيق هذه الواجهة. عدّلنا أيضًا `Deck` بهدف تقييد الأنواع التي يتعامل معها، بحيث يكون النوع المستخدم يُحقق الواجهة `Card`.

كل ما فعلناه إلى الآن هو إنشاء نوع بيانات جديد من خلال مفهوم البنى `struct` وجعله مُعمّمًا، لكن لغة جو تسمح أيضًا بإنشاء دوال مُعممة، إضافةً إلى إنشاء أنواع مُعممة.

40.6 إنشاء دوال معممة

يتبع إنشاء دوال مُعممة في لغة جو طريقة تصريح مشابه جدًا للأنواع المُعممة في لغة جو. يتطلب إنشاء دوال مُعممة إضافة مجموعة ثانية من المعاملات إلى تلك الدوال كما هو الحال في الأنواع المُعممة التي تحتوي على معاملات نوع.

سوف نُنشئ في هذا الجزء دالة مُعممة جديدة باسم `printCard`، ونستخدم تلك الدالة لطباعة اسم البطاقة المقدمة.

نفتح ملف main.go ونجري التحديثات التالية:

```
...
func printCard[C any](card C) {
  fmt.Println("card name:", card.Name())
}
func main() {
  ...
```

```
    fmt.Printf("card collectable name: %s\n",
tradingCard.CollectableName)
    fmt.Printf("--- printing cards ---\n")
    printCard[*PlayingCard](playingCard)
    printCard(tradingCard)
}
```

نلاحظ أن التصريح عن الدالة `printCard` مألوف من ناحية تعريف معاملات النوع (أقواس معقوفة تحتوي على معاملات النوع المعممة. تحدد هذه المعاملات النوع الذي سيجري استخدامه في الدالة)، تليها المعاملات العادية للدالة داخل قوسين، ثم في دالة `main`، تستخدم الدالة `printCard` لطباعة كل من `*PlayingCard` و `*TradingCard`.

قد نلاحظ أن أحد استدعاءات `printCard` يتضمن معامل النوع `[*PlayingCard]`، بينما الاستدعاء الثاني لا يحتوي على نفس معامل النوع `[*TradingCard]`. يمكن لمُصرّف لغة جو أن يستدل على معامل النوع المقصود من القيمة التي نمررها إلى المعاملات، لذلك في حالات مثل هذه، تكون معاملات النوع اختيارية. يمكننا أيضًا إزالة معامل النوع `[*PlayingCard]`.

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

سيظهر هذه المرة خطأ في التصريف:

```
# وسطاء سطر الأوامر
./main.go:87:33: card.Name undefined (type C has no field or method
Name)
```

عند استخدامنا للشيفرة المذكورة أعلاه وتصريفها تظهر رسالة الخطأ "card.Name undefined"، وتعني أن `Name` غير معرفة في النوع `C`. هذا يحدث لأننا استخدمنا `any` قيدًا للنوع في معامل النوع `C` في دالة `printCard`. تعني `any` أن أي نوع يمكن استخدامه مع `C`، ولكنه لا يعرف عنه أي شيء محدد مثل وجود التابع `Name` في النوع.

لحل هذه المشكلة والسماح بالوصول إلى `Name`، يمكننا استخدام الواجهة `Card` قيدًا للنوع في `C`. تحتوي الواجهة `Card` على التابع `Name` ونُطبقها في النوعين `TradingCard` و `PlayingCard`. وبذلك تضمن لنا لغة جو أن يكون لدى `C` التابع `Name` وبالتالي يتيح لنا استخدامها في الدالة `printCard` دون وجود أخطاء في وقت التصريف.

نُعدّل ملف main.go لآخر مرة لاستبدال القيد `any` بالقيد `Card`:

```
...
func printCard[C Card](card C) {
  fmt.Println("card name:", card.Name())
}
```

لنُشغّل ملف البرنامج main.go بعد حفظه من خلال الأمر `go run`:

```
$ go run main.go
```

سيكون الخرج:

```
--- drawing playing card ---
drew card: 6 of Hearts
card suit: Hearts
card rank: 6
--- drawing trading card ---
drew card: App Platform
card collectable name: App Platform
--- printing cards ---
card name: 6 of Hearts
card name: App Platform
```

نلاحظ سحب البطاقتين كما اعتدنا سابقًا، ولكن تطبع الدالة `printCard` البطاقات وتستخدم التابع `Name` للحصول على الاسم وطباعته.

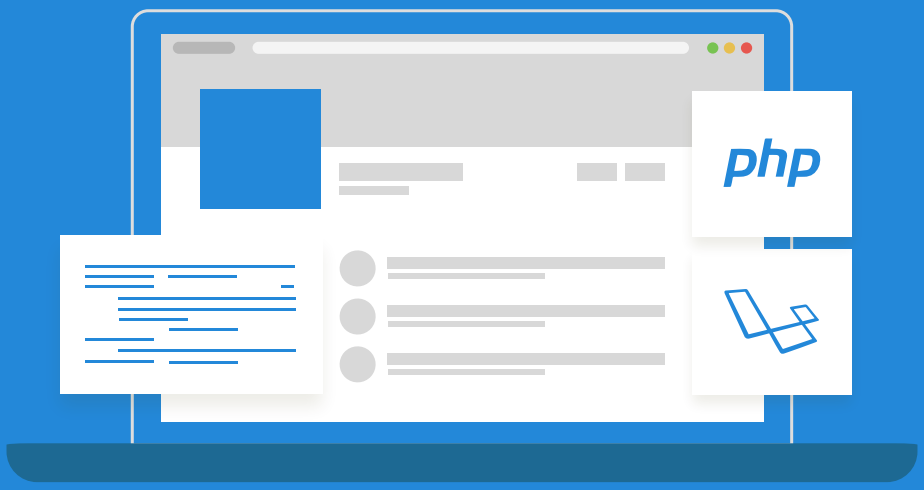
أنشأنا في هذا القسم دالةً جديدة مُعممة `printCard` قادرة على أخذ أي قيمة `Card` وطباعة الاسم. رأينا أيضًا أن استخدام قيد النوع `any` بدلًا من `Card` أو قيمةً أخرى محددة يؤثر على التوابع المتاحة.

40.7 الخاتمة

أنشأنا في هذا الفصل برنامجًا جديدًا يحتوي على بنية اسمها `Deck` تُمثّل نوعًا يمكن أن يُعيد بطاقة عشوائية من مجموعة ورق اللعب بصيغة `interface{}`، وأنشأنا النوع `PlayingCard` لتمثيل بطاقة اللعب في المجموعة، ثم عدّلنا النوع `Deck` لدعم مفهوم النوع المُعمّم وتمكنّا من إزالة بعض عمليات التحقق من الأخطاء لأن النوع المُعمّم يضمن عدم ظهور ذلك النوع من الأخطاء. أنشأنا بعد ذلك نوعًا جديدًا يسمى `TradingCard` لتمثيل نوع مختلف من البطاقات التي يمكن أن تدعمها `Deck`، وكذلك أنشأنا مجموعة ورق لعب لكل نوع من أنواع البطاقات وأرجعنا بطاقةً عشوائيةً من كل مجموعة. أضفنا أيضًا قيدًا للنوع إلى البنية `Deck` لضمان أنه يمكن إضافة أنواع تُحقق الواجهة `Card` فقط إلى مجموعة ورق اللعب. أنشأنا أخيرًا دالة مُعمّمة تسمى `printCard` يمكنها طباعة اسم أي قيمة من نوع `Card` باستخدام التابع `Name`.

يمكن أن يُقلل استخدام الأنواع المُعمّمة في الشيفرة الخاصة بنا إلى حد كبير عدد الأسـطر البرمجيـة اللازمـة لدعم أنواع متعددة لنفس الشيفرة. يجب أن نُحقق التوازن بين الأداء وسهولة قراءة الشيفرة عند استخدام الأنواع المُعمّمة؛ إذ يؤثر استخدام الأنواع المُعمّمة على الأداء، لكنه يُسهّل القراءة. من جهـة أخـرى، اسـتخدام الواجهـات بدلًا من الأنواع المُعمّمة أفضل من ناحية الأداء، لكنه أسوأ في القراءة. بالتالي، إذا كان بإمكاننا استخدام الواجهـة بدلًا من الأنواع المُعمّمة فمن الأفضل استخدام الواجهة.

دورة تطوير تطبيقات الويب
باستخدام لغة PHP



احترف تطوير النظم الخلفية وتطبيقات الويب
من الألف إلى الياء دون الحاجة لخبرة برمجية مسبقة

التحق بالدورة الآن

41. استخدام القوالب Templates

إذا كنا بحاجة إلى عرض البيانات بتنسيقات منظمة مثل التقارير النصية أو صفحات HTML، فإن قوالب لغة جو توفر حلًا فعالًا. تتضمن مكتبة لغة جو القياسية حزمتين تسمحان لأي برنامج مكتوب في هذه اللغة بتقديم البيانات بطريقة منسقة بدقة، وهما text/template و html/template.

يمكننا باستخدام هذه الحزم إنشاء قوالب نصية وتمرير البيانات فيها لتصيير render مستندات مصممة خصيصًا لمتطلباتنا. توفر القوالب المرونة في التكرار على البيانات باستخدام الحلقات وتطبيق المنطق الشرطي لتحديد محتوى ومظهر كل عنصر. نستكشف في هذا الفصل كيفية استخدام كلتا حزم القوالب. نستخدم في البداية حزمة `text/template` لإنشاء تقرير نصي عادي من خلال الاستفادة من الحلقات والعبارات الشرطية والدوال المخصصة. نستخدم بعد ذلك `html/template` لتصيير مستند HTML مع ضمان الحماية ضد الثغرات الأمنية في إدخال التعليمات البرمجية.

41.1 المتطلبات الأولية

لمتابعة هذا الفصل ستحتاج إلى التالي:

- امتلاك مساحة عمل خاصة مُهيئة وجاهزة في لغة جو، فإذا لم يكن لديك واحدة، فارجع للتعليمات الواردة في الفصل الأول من الكتاب، وثبت لغة جو Go وقم بإعداد بيئة تطوير محلية بحسب نظام تشغيلك، ويُفضّل أن تكون قد اطلعت أيضًا على فقرة التعرّف على GOPATH وفقرة استيراد الحزم في لغة جو Go قبل المتابعة في قراءة الفقرات التالية.
- معرفة بطريقة إنشاء البنى Structs وتعريف التوابع في لغة جو.

41.2 الخطوة 1: استيراد حزمة text/template

لنفترض أننا نريد إنشاء تقرير بسيط عن بيانات الكلاب التي لدينا. تنسيق التقرير المطلوب هو كما يلي:

```
\---
Name: Jujube
Sex:  Female (spayed)
Age:  10 months
Breed: German Shepherd/Pitbull
\---
Name: Zephyr
Sex:  Male (intact)
Age:  13 years, 3 months
Breed: German Shepherd/Border Collie
```

نحتاج لإنشاء هذا التقرير باستخدام حزمة `text/template` إلى استيراد الحزمة اللازمة وإعداد المشروع. يتألف التقرير من نص ثابت في القالب (العناصر على اليسار) وبيانات ديناميكية نمررها إلى القالب لتقديمها (على اليمين). يمكن تخزين القوالب مثل متغيرات من النوع `string` ضمن الشيفرة أو ملفات منفصلة خارج الشيفرة. تحتوي القوالب على نص ثابت معياري متداخل مع عبارات شرطية `else` و `if` وعبارات التحكم في التدفق (الحلقات) واستدعاءات الدوال، وكلها محاطة داخل أقواس معقوصة `{{. . .}}`. أخيرًا يمكننا إنشاء المستند النهائي من خلال توفير البيانات للقالب كما في المثال أعلاه.

ننتقل الآن إلى مساحة العمل الخاصة بنا `go env GOPATH` وننشئ مجلدًا جديدًا لهذا المشروع، ثم ننتقل إليه. يمكن إجراء ذلك من خلال التعليمات التالية بالترتيب:

```
$ cd `go env GOPATH`
$ mkdir pets
$ cd pets
```

باستخدام محرر نانو nano أو أي محرر آخر تريده، نفتح ملفًا جديدًا يسمى pets.go:

```
$ nano pets.go
```

ونضع فيه التعليمات التالية:

```
package main
import (
  "os"
```

```
    "text/template"
)
func main() {
}
```

تنتمي الشيفرة السابقة إلى الحزمة main، وتتضمّن الدالة main التي تسمح بتنفيذ الشيفرة باستخدام الأمر go run. تستورد الشيفرة حزمتين، هما: text/template من مكتبة جو القياسية، والتي نستخدمها لكتابة القالب وعرضه، وحزمة os للتفاعل مع نظام التشغيل من خلال الدوال التي توفرها.

بذلك تكون الأمور جاهزة لبدء كتابة المنطق اللازم لإنشاء التقرير المطلوب باستخدام حزمة text/template.

41.3 الخطوة 2: إنشاء بيانات القالب

بدايةً يجب أن يكون لدينا بعض البيانات لتمريرها إلى القالب، لذا سنعرّف بنيةً تسمى Pet تمثل خصائص حيوان أليف. تحتفظ هذه البنية ببيانات كل كلب في التقرير.

```
. . .
type Pet struct {
  Name  string
  Sex  string
  Intact bool
  Age  string
  Breed string
}
```

نُنشئ أيضًا شريحةً من Pet لتخزين معلومات كلبين:

```
func main() {
  dogs := []Pet{
        {
              Name:  "Jujube",
              Sex:  "Female",
              Intact: false,
              Age:  "10 months",
              Breed: "German Shepherd/Pitbull",
        },
        {
```

```
                Name:  "Zephyr",
                Sex:  "Male",
                Intact: true,
                Age:  "13 years, 3 months",
                Breed: "German Shepherd/Border Collie",
        },
    }
} // end main
```

عرّفنا البنية `Pet` بحقول تمثل الخصائص المختلفة للحيوان الأليف. نستخدم هذه البنى للاحتفاظ ببيانات كل كلب في التقرير. تتضمن الحقول: اسم الحيوان الأليف `Name` وجنس الحيوان الأليف `Sex` وقيمة منطقية تشير إلى ما إذا كان الحيوان الأليف سليم `Intact` وعمر الحيوان الأليف `Age` والسلالة `Breed`.

أنشأنا داخل الدالة `main` شريحة `Pet` باسم `dogs` وملأناها بنموذجين من كلاب مختلفة. الكلب الأول يُسمّى `Jujube` والكلب الثاني `Zephyr`. من المهم ملاحظة أنه في سيناريو العالم الحقيقي يمكن جلب بيانات القالب من قاعدة بيانات أو الحصول عليها من واجهة برمجة تطبيقات خارجية أو توفيرها من خلال إدخال المستخدم، لكن هنا أدخلنا البيانات يدويًا.

يمكننا الآن المتابعة إلى الخطوة التالية لكتابة القالب وعرضه.

41.4 الخطوة 3: تنفيذ وعرض بيانات القالب

حان الوقت الآن لاستكشاف كيفية استخدام حزمة `text/template` لتوليد مستند من قالب، ولكي نتأكد من أن الأمور تعمل بنجاح، سننشئ ملف قالب فارغ ثم نمرّر البيانات إلى القالب لتنفيذه. على الرغم من أن النموذج الأولي لن يعرض سوى النص "Nothing here yet"، إلا أنه سيكون بمثابة نقطة بداية لتوضيح دوال الحزمة `text/template`.

ننشئ ملف باسم pets.tmpl بالمحتوى التالي:

```
Nothing here yet.
```

نحفظ القالب ونخرج من المحرر. في حالة المحرر نانو nano، نضغط على المفتاحين Ctrl+X ثم المفتاح Y و ENTER لتأكيد التغييرات. نضيف الآن مقتطف الشفرة التالي داخل `main`:

```
. . .
    var tmplFile = "pets.tmpl"
    tmpl, err := template.New(tmplFile).ParseFiles(tmplFile)
    if err != nil {
```

```
            panic(err)
    }
    err = tmpl.Execute(os.Stdout, dogs)
    if err != nil {
            panic(err)
    }
} // نهاية الدالة main
```

صرّحنا ضمن الدالة `main` عن المتغير `tmplFile` وأسندنا له القيمة `pets.tmpl`، والتي تمثل اسم ملف القالب. استخدمنا بعد ذلك الدالة `template.New` لإنشاء قالب من `template`، مع تمرير `tmplFile` اسمًا للقالب. استدعينا بعد ذلك `ParseFiles` في القالب الذي أنشأناه حديثًا، مع تمرير `tmplFile` مثل ملف لتحليله. تربط هذه الخطوة ملف القالب بالقالب.

تحققنا بعد ذلك من أية أخطاء حدثت أثناء تحليل القالب. نلتقط الخطأ في حالة حدوثه، وتنتج لدينا حالة انهيار panic في البرنامج.

الآن لتنفيذ القالب نستدعي التابع `Execute`، ونمرر له وسيط أول `os.Stdout` ليكون وجهة الخرج ووسيط ثان `dogs` ليمثّل البيانات الممررة إلى القالب. يمثل `os.Stdout` (أو أي شيء آخر يحقق الواجهة `io.Writer`، أي ملف مثلًا) الخرج القياسي الذي سيطبع في هذه الحالة التقرير المُنشأ على الطرفية.

سيؤدي تنفيذ القالب في هذه المرحلة إلى عرض النص المذكور أنفًا، وذلك لأننا لا نستخدم بيانات ديناميكية في القالب.

ستكون الشيفرة كاملة كما يلي:

```
package main
import (
    "os"
    "text/template"
)
type Pet struct {
    Name  string
    Sex  string
    Intact bool
    Age  string
    Breed string
}
func main() {
```

```
    dogs := []Pet{
          {
                Name:  "Jujube",
                Sex:  "Female",
                Intact: false,
                Age:  "10 months",
                Breed: "German Shepherd/Pitbull",
          },
          {
                Name:  "Zephyr",
                Sex:  "Male",
                Intact: true,
                Age:  "13 years, 3 months",
                Breed: "German Shepherd/Border Collie",
          },
    }
    var tmplFile = "pets.tmpl"
    tmpl, err := template.New(tmplFile).ParseFiles(tmplFile)
    if err != nil {
          panic(err)
    }
    err = tmpl.Execute(os.Stdout, dogs)
    if err != nil {
          panic(err)
    }
} // main نهاية الدالة
```

لنُشغّل ملف البرنامج pets.go من خلال الأمر go run:

```
$ go run pets.go
```

سيكون الخرج على النحو التالي:

```
Nothing here yet.
```

لا يطبع البرنامج البيانات حتى الآن، ولكن على الأقل يعمل بطريقة صحيحة. لنكتب الآن قالبًا.

41.5 الخطوة 4: كتابة قالب

القالب هو أكثر من مجرد نص عادي بترميز UTF-8، إذ يحتوي القالب على نص ثابت إضافةً إلى الإجـراءات التي توجّه محرك القالب حول كيفية معالجة البيانات وإنشاء المخرجـات. تُغلّـف الإجـراءات بـأقواس معقوصـة `{{ <action> }}`، وتعمل على البيانات باستخدام تدوين النقطة (`.`).

من الشائع استخدام بُنى البيانات القابلة للتكرار عند تمرير البيانات إلى قالب، مثل الشرائح أو المصـفوفات أو الخرائط maps. سنستكشف في هذه الخطوة كيفية التكرار على شريحة في القالب باستخدام `range`.

41.5.1 التكرار على شريحة

يمكننا استخدام الكلمة المفتاحية `range` في لغة جو داخل حلقة `for` للتكرار على شريحة، وكذلك هو الحال في القوالب؛ إذ يمكننا استخدام الإجراء `range` لتحقيـق نفس النتيجـة، ولكن بصـيغة مختلفـة قليلًا؛ فبـدلًا من استخدام كلمة مفتاحية `for`، يمكن ببساطة استخدام `range` متبوعًا بالبيانـات القابلـة للتكـرار، وتغلـق الحلقـة بالتعليمة `{{ end }}`.

لنعدّل ملف pets.tmpl عن طريق استبدال محتوياته بما يلي:

```
{{ range . }}
\---
(Pet will appear here...)
{{ end }}
```

يأخذ الإجراء `range` النقطة (`.`) وسيطًا له، والذي يمثـل كامـل شـريحة `dogs`، ثم نُغلـق الحلقـة باسـتخدام `{{ end }}`. نضع ضمن الحلقة نصًا ثابتًا سيُعرض لكل حيوان أليف. في هذه المرحلة عمومًا، لن تُعـرض أيـة معلومات عن الكلاب في الخرج.

احفظ الملف pets.tmpl وشغّل ملف البرنامج pets.go من خلال الأمر `go run`:

```
$ go run pets.go
```

سيكون الخرج على النحو التالي:

```
\---
(Pet will appear here...)
\---
(Pet will appear here...)
```

يُطبع النص الثابت مرتين نظرًا لوجود كلبين في الشريحة. دعونا الآن نستبدل هذا ببعض النصوص الثابتة المفيدة، إلى جانب بيانات الكلاب.

41.5.2 عرض حقل

عند استخدام `range` مع النقطة `.` في القالب السابق، إذ تشير النقطة إلى العنصر الحالي في الشريحة أثناء كل تكرار للحلقة وعندما يكون هناك عنصر واحد في الشريحة فهو يشير إلى كامل الشريحة. يتيح ذلك الوصول إلى الحقول المُصدّرة لكل حيوان أليف مباشرةً باستخدام تدوين النقطة دون الحاجة إلى الإشارة إلى فهارس الشريحة. بالتالي لكي نعرض حقل، يمكن ببساطة تغليفه بأقواس معقوصة وإسباقه بنقطة. لنحّدث ملف pets.tmpl بالشيفرة التالية:

```
{{ range . }}
\---
Name: {{ .Name }}
Sex:  {{ .Sex }}
Age:  {{ .Age }}
Breed: {{ .Breed }}
{{ end }}
```

بذلك سيتضمن الخرج أربعة حقول لكل كلب: الاسم والجنس والعمر والسلالة. تمثل النقطة `.` الحيوان الأليف الحالي الذي يجري تكراره في الحلقة. إذًا يمكننا الوصول إلى الحقل المقابل لكل حيوان أليف وعرض قيمته جنبًا إلى جنب مع التسميات المناسبة باستخدام تدوين النقطة.

لنُشغّل ملف البرنامج pets.go من خلال الأمر `go run`

```
$ go run pets.go
```

سيكون الخرج على النحو التالي:

```
\---
Name: Jujube
Sex:  Female
Age:  10 months
Breed: German Shepherd/Pitbull
\---
Name: Zephyr
Sex:  Male
Age:  13 years, 3 months
```

```
Breed: German Shepherd/Border Collie
```

يبدو الأمر جيدًا. الآن دعونا نرى كيفية استخدام المنطق الشرطي لعرض الحقل الخامس.

41.5.3 استخدام الشروط

تجاهلنا الحقل `Intact` في القالب السابق؛ لإبقاء التقرير أكثر سهولة للقراءة، فبدلًا من عرض القيمة المنطقية مباشرةً `true` أو `false`، يمكننا استخدام إجراء `if-else` لتخصيص الخرج بناءً على قيمة الحقل، وتقديم معلومات أوضح وأكثر سهولة للفهم من مجرد وضع `true` أو `false`.

نفتح ملف pets.tmpl مجددًا ونعدّل القالب على النحو التالي:

```
{{ range . }}
\---
Name: {{ .Name }}
Sex:  {{ .Sex }} ({{ if .Intact }}intact{{ else }}fixed{{ end }})
Age:  {{ .Age }}
Breed: {{ .Breed }}
{{ end }}
```

يشتمل القالب الآن على عبارة `if-else` للتحقق من قيمة الحقل `Intact`. إذا كان الحقل `true`، فإنه يطبع `(intact)`، وإلا فإنه يطبع `(fixed)`. يمكننا أيضًا تحسين الخرج أكثر؛ من خلال عرض المصطلحات الخاصة بالجنس لكلب حالته `fixed`، مثل `spayed` أو `neutered`، بدلًا من استخدام المصطلح العام `fixed`. لتحقيق ذلك يمكننا إضافة عبارة `if` متداخلة داخل كتلة `else`:

```
{{ range . }}
\---
Name: {{ .Name }}
Sex:  {{ .Sex }} ({{ if .Intact }}intact{{ else }}{{ if (eq .Sex
"Female") }}spayed{{ else }}neutered{{ end }}{{ end }})
Age:  {{ .Age }}
Breed: {{ .Breed }}
{{ end }}
```

مع هذا التعديل؛ يتحقق القالب أولًا مما إذا كان الحيوان الأليف سليمًا. إذا لم يكن الأمر كذلك، فإنه يتحقق أيضًا مما إذا كان الحيوان الأليف أنثى `Female`. يسمح هذا بمعلومات أكثر دقة في التقرير.

احفظ ملف القالب وشغّل ملف البرنامج pets.go من خلال الأمر `go run`:

```
$ go run pets.go
```

سيكون الخرج:

```
\---
Name: Jujube
Sex:  Female (spayed)
Age:  10 months
Breed: German Shepherd/Pitbull
\---
Name: Zephyr
Sex:  Male (intact)
Age:  13 years, 3 months
Breed: German Shepherd/Border Collie
```

لدينا كلبان وثلاث حالات محتملة لعرض `Intact`. دعونا نضيف كلبًا آخر إلى الشريحة في `pets.go` لتغطية الحالات الثلاث:

```
. . .
func main() {
  dogs := []Pet{
        {
              Name:  "Jujube",
              Sex:  "Female",
              Intact: false,
              Age:  "10 months",
              Breed: "German Shepherd/Pitbull",
        },
        {
              Name:  "Zephyr",
              Sex:  "Male",
              Intact: true,
              Age:  "13 years, 3 months",
              Breed: "German Shepherd/Border Collie",
        },
        {
              Name: "Bruce Wayne",
```

```
                Sex:  "Male",
                Intact:     false,
                Age:  "3 years, 8 months",
                Breed:      "Chihuahua",
        },
    }
. . .
```

لنُشغّل ملف البرنامج pets.go من خلال الأمر `go run`:

```
$ go run pets.go
```

ليكون الخرج:

```
\---
Name: Jujube
Sex:  Female (spayed)
Age:  10 months
Breed: German Shepherd/Pitbull
\---
Name: Zephyr
Sex:  Male (intact)
Age:  13 years, 3 months
Breed: German Shepherd/Border Collie
\---
Name: Bruce Wayne
Sex:  Male (neutered)
Age:  3 years, 8 months
Breed: Chihuahua
```

رائع، يبدو كما هو متوقع.

الآن دعونا نناقش دوال القالب، مثل الدالة `eq` التي استخدمناها للتو.

41.5.4 استخدام دوال القالب

توفر الحزمة `text/template` -إضافةً إلى الدالة `eq` التي استخدمناها سابقًا- العديد من الدوال الأخرى لمقارنة قيم الحقول وإرجاع النتائج المنطقية، مثل `gt` (أكبر من) و `ne` (عدم تساوي) و `le` (أقل من أو يساوي) والمزيد. يمكن استدعاء هذه الدوال بطريقتين مختلفتين:

1. كتابة اسم الدالة متبوعة بمعامل واحد أو أكثر ومفصولة بمسافات. هذه هي الطريقة التي استخدمنا بها الدالة `eq` في هذا الفصل: `eq .Sex "Female"`.
2. كتابة معامل واحد متبوع برمز الأنبوب `|`، ثم اسم الدالة والمعلمات الإضافية إذا لزم الأمر. يسمح هذا بربط استدعاءات عدة دوال معًا، مع جعل خرج كل دالة مدخلًا للتالية. هذا مشابه لكيفية عمل أنابيب الأوامر في سطر أوامر Unix.

مثلًا يمكن كتابة عملية المقارنة السابقة باستخدام الدالة `eq` في القالب بالشكل التالي: `.Sex | eq "Female"`، وهذا يُكافئ التعبير `eq .Sex "Female"`.

دعونا الآن نستخدم الدالة `len` لعرض عدد الكلاب في الجزء العلوي من التقرير. نفتح ملف pets.tmpl ونضيف له الشيفرة التالية في البداية:

```
Number of dogs: {{ . | len -}}
{{ range . }}
. . .
```

يمكنك أيضًا كتابتها بالشكل التالي `{{ len . -}}` حيث تحسب هذه الدالة طول البيانات المُمررة في النقطة `.` وهي في هذه الحالة شريحة الكلاب. بالتالي سنتمكن من عرض عدد الكلاب في أعلى التقرير من خلال تضمين هذه الدالة في القالب.

لاحظ الشَرطة `-` بجانب الأقواس المزدوجة المعقوصة، وتمنع هذه الشرطة طباعة الأسطر الجديدة `\n` بعد الإجراء. يمكن أيضًا استخدامها لمنع طباعة السطر الجديد قبل الإجراء من خلال وضعها قبل الإجراء، أي في البداية `{{ - len . -}}`.

لنُشغّل ملف البرنامج pets.go من خلال الأمر `go run`:

```
$ go run pets.go
```

سيكون الخرج على النحو التالي:

```
Number of dogs: 3
\---
Name: Jujube
Sex:  Female (spayed)
Age:  10 months
Breed: German Shepherd & Pitbull
\---
Name: Zephyr
```

```
Sex:  Male (intact)
Age:  13 years, 3 months
Breed: German Shepherd & Border Collie
\---
Name: Bruce Wayne
Sex:  Male (neutered)
Age:  3 years, 8 months
Breed: Chihuahua
```

باستخدام الشرطة في `{{- | len .}}`، لا توجد أسطر فارغة بين جملة `Number of dogs` وتفاصيل الكلب الأول.

تجدر الإشارة إلى أن الدوال المضمنة في حزمة `text/template` محدودة، ومع ذلك يمكن استخدام أي دالة في لغة جو مع القوالب طالما أنها تُرجع قيمة واحدة أو قيمتين، إذ تكون الثانية قيمةً من نوع خطأ `error`. يسمح هذا بتوسيع دوال القوالب التي يمكن استخدامها من خلال دمج دوال خاصة.

41.5.5 استخدام دوال لغة جو مع القوالب

لنفترض أننا نريد كتابة قالبًا يأخذ شريحة من الكلاب ويعرض فقط الكلب الأخير من بينها. يمكننا في قوالب لغة جو استخراج مجموعة فرعية من شريحة باستخدام الدالة المبنية مسبقًا `slice`، والتي تعمل بطريقة تشبه `mySlice [x:y]` في لغة جو. إذا كنا نريد مثلًا استرداد العنصر الأخير من شريحة مكونة من ثلاثة عناصر، فيمكن استخدام `{{ slice . 2 }}`.

من المهم ملاحظة أن `slice` تُرجع شريحةً أخرى، وليس عنصرًا فرديًا. لذا فإن الكتابة `{{slice. 2}}` تكافئ `slice [2:]`، وليس `slice [2]`. يمكن أيضًا أن تقبل الدالة `slice` عدة فهارس، مثل `{{ slice. 0 2 }}` لاسترداد الشريحة `slice [0: 2]`، لكننا لن نستخدم ذلك في هذا السيناريو.

يظهر التحدي عندما نريد الإشارة إلى الفهرس الأخير للشريحة داخل القالب الخاص بنا. على الرغم من أن الدالة `len` متاحة، إلا أن العنصر الأخير في الشريحة موجود في الفهرس `len - 1`، وللأسف، لا تدعم القوالب العمليات الحسابية. هنا يمكننا إنشاء دالة مخصصة للتغلب على هذا القيد، وذلك من خلال كتابة دالة تُنقص قيمة عدد صحيح مُمرر لها.

بدايةً ننشئ ملف قالب جديد. نفتح ملفًا جديدًا يسمى lastPet.tmpl ونضع المحتوى التالي:

```
{{- range (len . | dec | slice . ) }}
\---
Name: {{ .Name }}
```

```
Sex:  {{ .Sex }} ({{ if .Intact }}intact{{ else }}{{ if ("Female" | eq
.Sex) }}spayed{{ else }}neutered{{ end }}{{ end }})
Age:  {{ .Age }}
Breed: {{ .Breed }}
{{ end -}}
```

يستخدم هذا القالب إجراء `range` مع الشريحة المُعدّلة للتكرار على آخر كلب في الشريحة المحـددة. نُطبّـق الدالة المخصصة `dec` الموجودة في السطر الأول لتقليل طول الشريحة، مما يسـمح لنـا بالوصـول إلى الفهـرس الأخير. يعرض القالب بعد ذلك المعلومـات ذات الصـلة بـالكلب الأخـير، بمـا في ذلـك الاسـم والجنس والعمـر والسلالة.

لتعريف الدالة `dec` المخصصة وتمريرها إلى القالب، نُجري التغييرات التالية داخل الدالة `main` في الملــف pets.go -بعد شريحة الكلاب وقبل استدعاء `tmpl.Execute()`- كما هو موضح أدناه:

```
. . .
  funcMap := template.FuncMap{
        "dec": func(i int) int { return i - 1 },
  }
  var tmplFile = "lastPet.tmpl"
  tmpl, err :=
template.New(tmplFile).Funcs(funcMap).ParseFiles(tmplFile)
  if err != nil {
        panic(err)
  }
  . . .
```

صـرّحنا عن `FuncMap` على أنهـا خريطـة `map` للـدوال، إذ تمثِّـل أزواج (المفتـاح، القيمـة) أسـماء الـدوال والتطبيق المقابلة لها. نُعرّف في هذه الحالة الدالة `dec` على أنها دالة مجهولـة تطـرح `1` من عـدد صـحيح وتعيـد النتيجة.

نُغيّر بعد ذلك اسم ملف القالب إلى lastPet.tmpl. أخيرًا نستدعي التابع `Funcs` من القالب، قبل استدعاء `ParseFiles`، ونمرر لـه `funcMap` لإتاحـة الدالـة `dec` داخـل القـالب. من المهم ملاحظـة أنـه يجب اسـتدعاء `Funcs` قبل `ParseFiles` لتسجيل الدالة المخصصة بطريقة صحيحة مع القالب.

دعونا نفهم بدايةً ما يحدث في الإجراء `range`:

```
{{- range (len . | dec | slice . ) }}
```

يجري في هذا السطر الحصول على طول شريحة الكلاب باستخدام `len .`، ثم تمرير النتيجة إلى الدالة `dec` المخصصة لطرح قيمة `1` من المتغير المُمرر لها `len . | dec`، ثم تمرير النتيجة مثل معاملٍ ثانٍ إلى الدالة `slice`. لذلك، بعبارات أبسط، بالنسبة لشريحة مكونة من ثلاثة كلاب، فإن `range` تعادل ما يلي:

```
{{- range (slice . 2) }}
```

لنُشغّل ملف البرنامج pets.go بعد حفظه من خلال الأمر `go run`

```
$ go run pets.go
```

سيكون الخرج على النحو التالي:

```
\---
Name: Bruce Wayne
Sex:  Male (neutered)
Age:  3 years, 8 months
Breed: Chihuahua
```

ماذا لو أردنا إظهار آخر كلبين بدلًا من آخر كلب فقط؟ نُحرّر الملف lastPet.tmpl ونستدعي `dec` مجددًا:

```
{{- range (len . | dec | dec | slice . ) }}
. . .
```

لنُشغّل ملف البرنامج pets.go بعد حفظه من خلال الأمر `go run`:

```
$ go run pets.go
```

ليكون الخرج على النحو التالي:

```
\---
Name: Zephyr
Sex:  Male (intact)
Age:  13 years, 3 months
Breed: German Shepherd/Border Collie
\---
Name: Bruce Wayne
Sex:  Male (neutered)
Age:  3 years, 8 months
Breed: Chihuahua
```

يمكننا العمل على تحسين الدالة `dec` من خلال جعلها تأخذ معاملًا واحدًا ونغيّر اسمها بحيث نكتب `minus 2` بدلًا من كتابة `dec | dec`.

لنفرض أننا أردنا عرض الكلاب الهجينة مثل "Zephyr" بطريقة مختلفة، وذلك باستبدال الشرطة المائلة بعلامة العطف &. لحسن الحظ لن نضطر لكتابة دالة خاصة لذلك، إذ يمكننا الاستفادة من دالة موجودة في الحزمة `strings`، لكن نحتاج إلى إجراء بعض التغييرات على ملف pets.go قبل ذلك. نستورد أولًا الحزمة `strings` مع الحزم الأخرى في أعلى الملف. نُحدِّث بعد ذلك المتغير `funcMap` داخل الدالة `main` لتضمين دالة `ReplaceAll` من حزمة `strings`:

```
package main
import (
  "os"
  "strings"
  "text/template"
)
. . .
func main() {
  . . .
  funcMap := template.FuncMap{
        "dec":   func(i int) int { return i - 1 },
        "replace": strings.ReplaceAll,
  }
  . . .
} // نهاية الدالة main
```

من خلال إضافة `strings.ReplaceAll` إلى `funcMap`، نكون قد جعلناها متاحةً في القالب تحت الاسم `replace`. نفتح ملف lastPet.tmpl ونعدّله لاستخدام `replace`:

```
{{- range (len . | dec | dec | slice . ) }}
\---
Name: {{ .Name }}
Sex:  {{ .Sex }} ({{ if .Intact }}intact{{ else }}{{ if ("Female" | eq
.Sex) }}spayed{{ else }}neutered{{ end }}{{ end }})
Age:  {{ .Age }}
Breed: {{ replace .Breed "/" " & " }}
{{ end -}}
```

لنُشغّل ملف البرنامج pets.go بعد حفظه من خلال الأمر `go run`:

```
$ go run pets.go
```

سيكون الخرج على النحو التالي:

```
\---
Name: Zephyr
Sex:  Male (intact)
Age:  13 years, 3 months
Breed: German Shepherd & Border Collie
\---
Name: Bruce Wayne
Sex:  Male (neutered)
Age:  3 years, 8 months
Breed: Chihuahua
```

تحتوي سلالة Zephyr الآن على علامة عطف بدلًا من شرطة مائلة. أجرينا هذا التعديل على حقل `Breed` داخل القالب بدلًا من تعديل البيانات في `pets.go`. يتبع هذا المبدأ القائل بأن عرض البيانات هو مسؤولية القوالب وليس الشيفرة.

تجدر الإشارة إلى أن بعض بيانات الكلاب، مثل حقل `Breed`، تحتوي على بعض المظاهر التي قد لا تكون طريقة عرض المعلومات فيها مثالية في هندسة البرمجيات عمومًا، إذ يمكن أن يؤدي التنسيق الحالي لتخزين سلالات متعددة في سلسلة واحدة مفصولة بشرطة مائلة `/` إلى اختلافات في عملية إدخال البيانات، مما يؤدي إلى تنسيقات غير متسقة في قاعدة البيانات (على سبيل المثال: `Labrador/Poodle` و `Labrador & Poodle` و `Labrador, Poodle` و `Labrador-Poodle mix`).

لمعالجة هذه المشكلة وتحسين المرونة في البحث حسب السلالة وتقديم البيانات، قد يكون من الأفضل تخزين حقل `Breed` مثل شريحة من السلاسل (`[]string`) بدلًا من سلسلة واحدة. سيؤدي هذا التغيير إلى إزالة الغموض في التنسيق ويسمح بمعالجة أسهل في القوالب. يمكن بعد ذلك استخدام دالة `strings.Join` ضمن القالب لربط جميع السلالات، جنبًا إلى جنب مع ملاحظة إضافية من خلال الحقل `.Breed`. بحيث تشير إلى ما إذا كان الكلب سلالة أصيلة `(purebred)` أو سلالة هجينة `(mixed breed)`.

دعونا في الختام نعرض نفس البيانات في مستند HTML ونرى لماذا يجب علينا دائمًا استخدام حزمة `html/template` عندما يكون ناتج القالب الخاصة بنا بتنسيق HTML.

41.6 الخطوة 5: كتابة قالب HTML

في حين أن الحزمة `text/template` مناسبة لطباعة الخرج بدقة (سواءً من سطر الأوامر أو مكان آخر) وإنشاء ملفات منظّمة من البرامج الدفعية Batch program (برامج تعالج سلسلة من المهام أو الأوامر دفعة واحدة أو بطريقة غير تفاعلية)، إلا أنه من الشائع استخدام قوالب لغة جو لتصيير صفحات HTML في تطبيقات الويب. على سبيل المثال، يعتمد مُنشئ الموقع الثابت (أداة تساعد في إنشاء ملفات HTML ثابتة بناءً على القوالب والمحتوى. يبسط عملية إنشاء مواقع الويب وإدارتها عن طريق تحويل القوالب والمحتوى والموارد الأخرى إلى موقع ويب ثابت جاهز للنشر) هوغو Hugo على كل من `text/template` و `html/template` مثل أساس لنظام القوالب الخاص به. تتيح هذه الحزم للمستخدمين تحديد القوالب ذات معاملات النوع وإدراج البيانات ديناميكيًا فيها، مما يتيح إنشاء صفحات HTML لمواقع الويب.

تقدم لغة HTML ميزات فريدة لا نراها مع النص العادي، إذ تستخدم أقواس الزاوية لتغليف العناصر (`<td>`) وعلامات العطف لتمييز الكيانات (` `) وعلامات الاقتباس لتغليف قيم أو سمات الوسوم (`<a href="https://www.digitalocean.com/">`). عند إدخال البيانات التي تحتوي على هذه الأحرف باستخدام حزمة `text/template`، يمكن أن ينتج عن ذلك HTML تالف أو حتى حقن شيفرة Code injection (وهي ثغرة أمنية يتمكن منها المهاجم من إدخال التعليمات البرمجية الضارة وتنفيذها داخل تطبيق أو نظام).

تعالج حزمة `html/template` هذه التحديات، بحيث تهرب تلقائيًا من المحارف التي قد تخلق إشكالية، وتستبدلها بكيانات HTML الآمنة. تُصبح علامة العطف في البيانات (`&amp;`) وقوس الزاوية اليسرى (`&It;`) وهكذا.

دعونا نواصل استخدام نفس بيانات الكلاب، لإثبات خطورة استخدام `text/template` مع HTML. نفتح pets.go ونعدّل حقل `Name` على النحو التالي:

```
. . .
  dogs := []Pet{
        {
              Name:  "<script>alert(\"Gotcha!\");</script>Jujube",
              Sex:  "Female",
              Intact: false,
              Age:  "10 months",
              Breed: "German Shepherd/Pit Bull",
        },
        {
              Name:  "Zephyr",
```

```
                Sex:  "Male",
                Intact: true,
                Age:  "13 years, 3 months",
                Breed: "German Shepherd/Border Collie",
        },
        {
                Name:  "Bruce Wayne",
                Sex:  "Male",
                Intact: false,
                Age:  "3 years, 8 months",
                Breed: "Chihuahua",
        },
  }
    . . .
```

نُنشئ الآن قالب HTML في ملف جديد يسمى petsHtml.tmpl:

```
<p><strong>Pets:</strong> {{ . | len }}</p>
{{ range . }}
<hr />
<dl>
  <dt>Name</dt>
  <dd>{{ .Name }}</dd>
  <dt>Sex</dt>
  <dd>{{ .Sex }} ({{ if .Intact }}intact{{ else }}{{ if (eq .Sex
"Female") }}spayed{{ else }}neutered{{ end }}{{ end }})</dd>
  <dt>Age</dt>
  <dd>{{ .Age }}</dd>
  <dt>Breed</dt>
  <dd>{{ replace .Breed "/" " & " }}</dd>
</dl>
{{ end }}
```

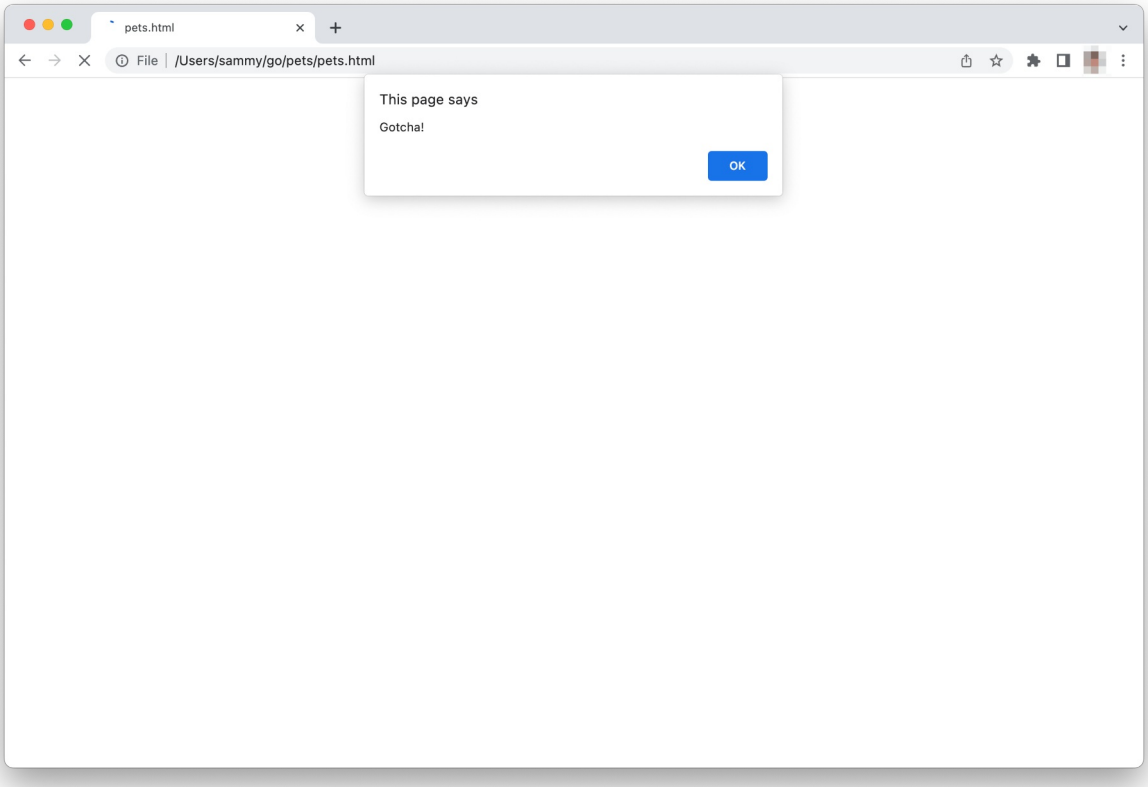
نحفظ قالب HTML. نحتاج الآن إلى تعديل المتغير `tmpFile` قبل تشغيل pets.go، ولكن دعونا أيضًا نُعدّل البرنامج لإخراج القالب إلى ملف بدلًا من الطرفية. نفتح الملف pets.go ونضيف الشيفرة التالية داخل الدالة `main()`:

```
. . .
  funcMap := template.FuncMap{
        "dec":   func(i int) int { return i - 1 },
        "replace": strings.ReplaceAll,
  }
  var tmplFile = "petsHtml.tmpl"

  tmpl, err :=
template.New(tmplFile).Funcs(funcMap).ParseFiles(tmplFile)
  if err != nil {
        panic(err)
  }
  var f *os.File
  f, err = os.Create("pets.html")
  if err != nil {
        panic(err)
  }
  err = tmpl.Execute(f, dogs)
  if err != nil {
        panic(err)
  }
  err = f.Close()
  if err != nil {
        panic(err)
  }
} // end main
```

نفتح ملـف جديـد يسـمى pets.html ونمـرّره (بـدلًا من `os.Stdout`) إلى `tmpl.Execute`، ثم نغلـق الملف عند الانتهاء.

لنُشغّل ملف البرنامج pets.go من خلال الأمر `go run` لإنشاء ملف HTML. نفتح بعد ذلك صفحة الويب المحلية هذه في المتصفح:

```
$ go run pets.go
```

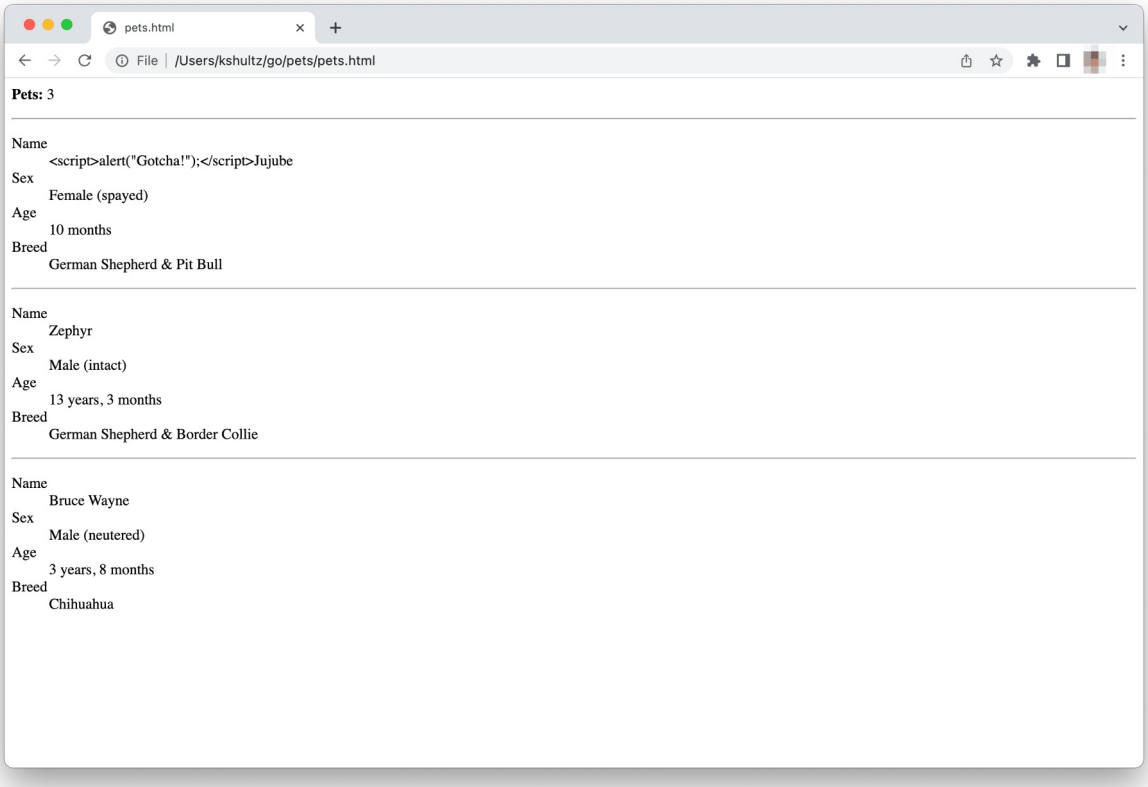


شـغّل المتصـفح البرنـامج النصـي المحقـون، وهـذا هـو السـبب في أنـه لا يجب أبـدًا اسـتخدام حزمـة `text/template` لإنشاء HTML، خاصةً عندما لا يمكن الوثوق تمامًا بمصدر بيانات القالب.

بصرف النظر عن محارف الهروب في HTML، سوف تعمل الحزمة `html /template` تمامًا مثل الحزمة `text/template` ولها نفس الاسم الأساسي template، مما يعني أن كل ما علينا فعله لجعل القالب آمنًا هو استبدال استيراد `text/template` مع `html /template`. لنعدّل ملف pets.go وفقًا لذلك الآن:

```
package main
import (
  "os"

  "strings"
  "html/template"
)
. . .
```

نحفظ الملف لتعديل بيانات pets.html ونشغّله مرةً أخيرة. ثم نعيد تحميل ملف HTML في المتصفح:



صـيّرت حزمـة `html/template` النص المُـدخل على أنـه نص فقـط في صـفحة الـويب. نفتح الملـف pets.html في محرر النصوص (أو نعرض مصدر الصفحة في المتصفح) وننظر إلى أول كلب Jujube:

```
. . .
<dl>
    <dt>Name</dt>

<dd>&lt;script&gt;alert(&#34;Gotcha!&#34;);&lt;/script&gt;Jujube</dd>
    <dt>Sex</dt>
    <dd>Female (spayed)</dd>
    <dt>Age</dt>
    <dd>10 months</dd>
    <dt>Breed</dt>
    <dd>German Shepherd &amp; Pit Bull</dd>
</dl>
. . .
```

اسـتبدلت حزمـة `html` أقـواس الزاويـة ومحـارف الاقتبـاس في اسـم Jujube، وكـذلك علامـة العطـف في السلالة.

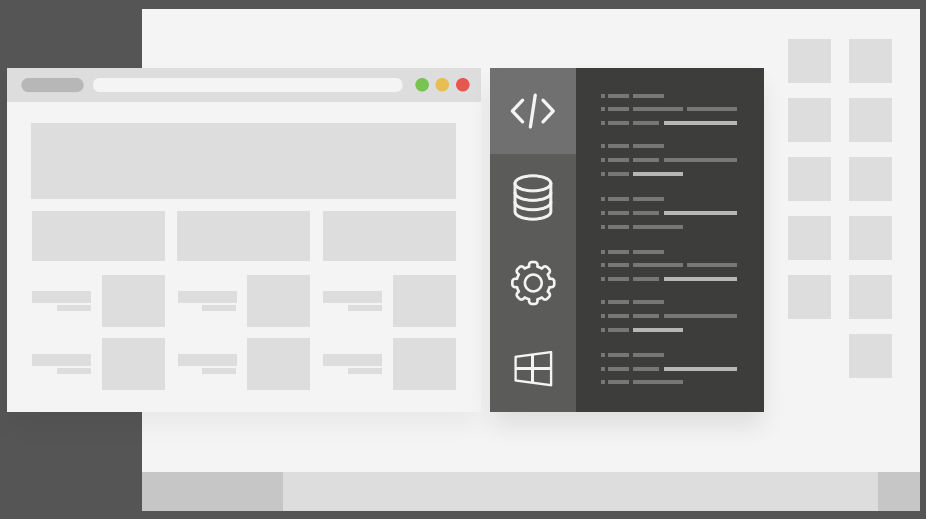
41.7 الخاتمة

تعلمت في هذا الفصل طريقة التعامل مع القوالب في لغة جو والتي توفر حلًا متعدد القدرات لدمج البيانات في تنسيقات نصية متنوعة. إذ يمكن استخدام القوالب في أدوات سطر الأوامر لتنسيق الخرج، وكذلك في تطبيقات الويب لإنشاء صفحات HTML. كما تعلمت كيفية الاستفادة من حزم القوالب المضمنة في لغة جو Go لإنتاج نص جيد التنظيم وعرض HTML باستخدام نفس البيانات.

وبإتمامك لهذا الفصل تكون قد وصلت لنهاية هذا الكتاب الشامل الذي يشرح لغة Go بالتفصيل واكتسبت أهم أساسيات البرمجة بهذه اللغة بداية من كتابة برامج بسيطة إلى التعامل مع مختلف أنواع والتعامل مع الأخطاء البرمجية والتعامل مع الوقت والتاريخ وصولًا إلى بناء تطبيقات الويب وأنت جاهز الآن للانتقال لمرحلة جديدة والبدء بتطوير تطبيقات معقدة تعزز معرض أعمالك وتحل مشكلات حقيقة.

نأمل أن يكون هذا الكتاب قد حقق الهدف المرجو وأكسبك كافة المهارات التي تحتاجها في رحلتك في تعلم البرمجة.

دورة علوم الحاسوب



دورة تدريبية متكاملة تضعك على بوابة الاحتراف
في تعلم أساسيات البرمجة وعلوم الحاسوب

التحق بالدورة الآن

أحدث إصدارات أكاديمية حسوب