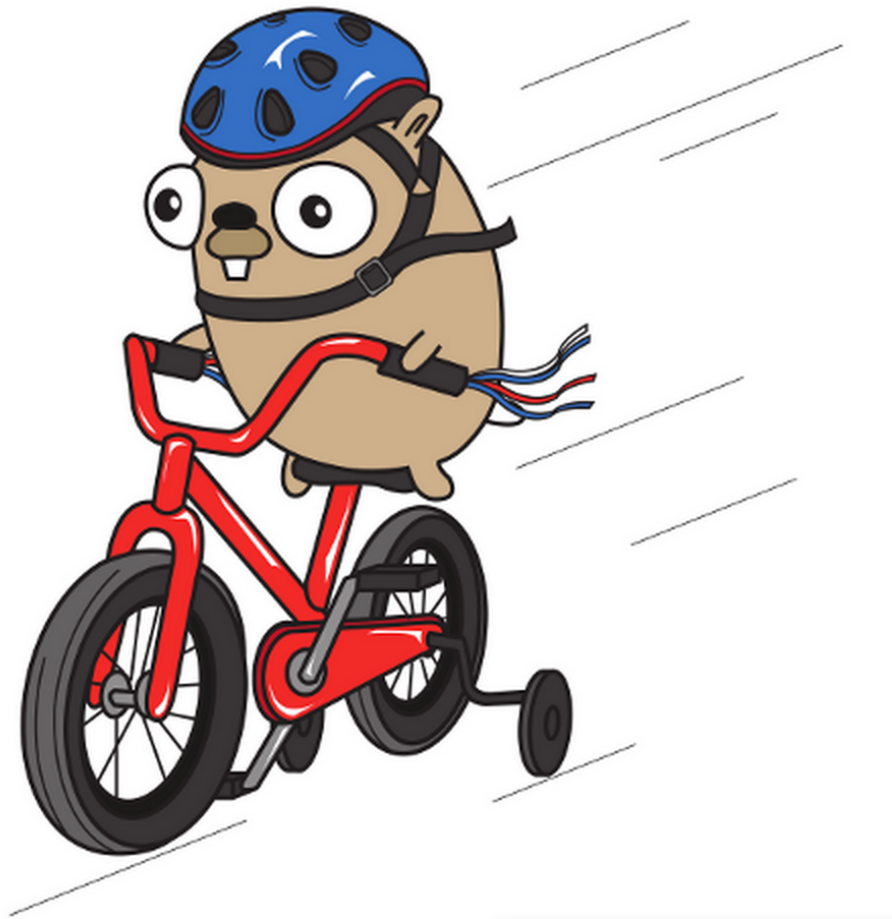


แนะนำการเขียนโปรแกรมด้วยภาษา โก(Golang)



โดย เซเลบ ด็อกซ์

บทที่1: บทนำ

การเขียนโปรแกรมนั้นเป็นทั้งศิลปะ,งานฝีมือและวิทยาศาสตร์ เพื่อสั่งงานให้คอมพิวเตอร์ ให้ทำงานตามที่เราต้องการ หนังสือเล่มนี้จะสอนให้คุณให้รู้จักกับการเขียนโปรแกรมคอมพิวเตอร์ด้วยภาษาในการเขียนโปรแกรมที่ออกแบบโดย Google ที่ชื่อว่าภาษาโก (Go)

โก เป็นภาษาที่ใช้ในการเขียนโปรแกรมเชิงอรรถประโยชน์ (general purpose programming language) ที่มาพร้อมกับฟีเจอร์ที่ก้าวหน้าต่างๆ มากมาย อีกทั้งไวยากรณ์ของภาษาที่ดูสะอาดสะอ้าน และเนื่องจากเป็นภาษาสามารถใช้งานได้หลากหลายแพลตฟอร์ม ซึ่งมีไลบรารีพื้นฐานที่มีความทนทานต่อข้อผิดพลาดและมีเอกสารประกอบที่ครบถ้วนสมบูรณ์ ทั้งยังเป็นภาษาที่คำนึงถึงการออกแบบตามหลักการของวิศวกรรมซอฟต์แวร์ที่ดีอีกด้วย ดังนั้นอาจจะเรียกได้ว่าเป็นภาษาในอุดมคติสำหรับผู้เริ่มเรียนรู้การเขียนโปรแกรมเป็นภาษาแรกเลยทีเดียว

กระบวนการที่เราใช้ในการพัฒนาโปรแกรมด้วยภาษาโก (และในภาษาอื่นๆ) นั้นค่อนข้างจะตรงไปตรงมา โดยประกอบด้วย:

- i. รวบรวมความต้องการ
- ii. ค้นหาแนวทางที่เหมาะสมในการแก้ปัญหา
- iii. ลงมือเขียนซอร์สโค้ด ตามแนวทางในการแก้ปัญหา
- iv. คอมไพล์ตัวซอร์สโค้ดให้อยู่ในรูปแบบที่พร้อมทำงาน(executable)
- v. รันและทดสอบโปรแกรม เพื่อให้แน่ใจว่าโปรแกรมทำงานได้ถูกต้องอย่างที่ต้องการ

โดยกระบวนการนี้จะถูกกระทำเป็นวงรอบ (หมายความว่ามันจะถูกทำซ้ำๆ ได้หลายรอบ) และแต่ละขั้นตอนก็มักจะมีการทับซ้อนกันอยู่ แต่ก่อนที่เราจะเริ่มเขียนโปรแกรมแรกด้วยภาษาโกนั้น มีแนวความคิดอยู่สองสามอย่างที่เราต้องทำความเข้าใจก่อน

1.1 ไฟล์และโฟลเดอร์

ไฟล์ คือกลุ่มของข้อมูลที่เก็บอยู่เป็นหน่วยเดียวกันโดยมีชื่อเรียก ระบบปฏิบัติการสมัยใหม่ (อย่าง วินโดวส์ หรือ Mac OSX) จะบรรจุไปด้วยไฟล์นับล้านไฟล์ ซึ่งจัดเก็บสารสนเทศมากมายหลากหลายประเภท นับตั้งแต่เทกซ์ (text) จนถึงไฟล์ที่พร้อมทำงาน (executable) และไฟล์ประเภทมัลติมีเดียทั้งหลาย

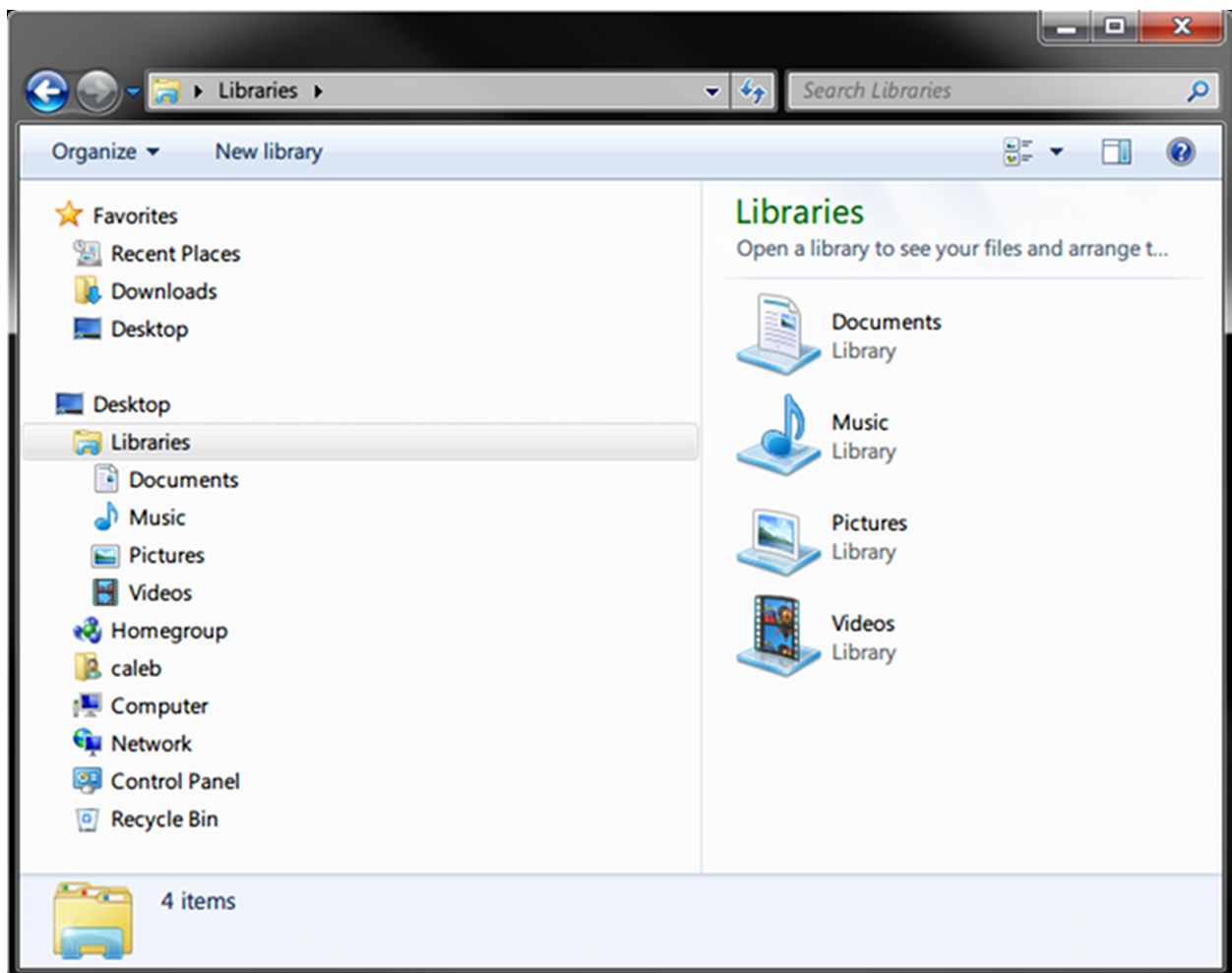
ไฟล์ทุกอันจะถูกจัดเก็บในแบบเดียวกันในคอมพิวเตอร์ โดยแต่ละไฟล์จะมีชื่อ ขนาดที่ชัดเจน (หน่วยเป็นไบต์) และประเภทของไฟล์ โดยทั่วไปประเภทของไฟล์จะถูกบ่งบอกด้วยนามสกุลของไฟล์ (คือส่วนของชื่อไฟล์ที่ตามหลังเครื่องหมาย . ยกตัวอย่างเช่น `hello.txt` จะมีนามสกุลเป็น `txt` ซึ่งจะจัดเก็บข้อมูลประเภทตัวอักษร)

โฟลเดอร์ (หรือเรียกอีกอย่างว่าไดเรกทอรี) จะใช้ในการจัดกลุ่มของไฟล์เข้าไว้ด้วยกัน และสามารถมีโฟลเดอร์อยู่ข้างในด้วยก็ได้ ในระบบวินโดวส์ ไฟล์และโฟลเดอร์พาร (path) (ที่อยู่ของไฟล์) จะถูกแทนด้วยอักษร \ (backslash) ยกตัวอย่างเช่น: C:\Users\john\example.txt โดย example.txt คือชื่อของไฟล์นั่นเอง ซึ่งไฟล์นี้จะอยู่ภายใต้โฟลเดอร์ชื่อ john ซึ่งตัวโฟลเดอร์เองก็อยู่ภายใต้โฟลเดอร์ Users ซึ่งอยู่ในไดรฟ์ C (ซึ่งใช้แทนตัว physical hard drive ในระบบวินโดวส์)อีกทีหนึ่ง

บนระบบปฏิบัติการ OSX (และระบบปฏิบัติการที่เหลืเกือบทั้งหมด) ไฟล์และโฟลเดอร์ path จะถูกแสดงด้วยเครื่องหมาย / (forward slash) ยกตัวอย่างเช่น: /Users/john/example.txt และเช่นเดียวกับระบบวินโดวส์ example.txt ก็คือชื่อไฟล์ ซึ่งอยู่ภายใต้โฟลเดอร์ john ซึ่งอยู่ภายในโฟลเดอร์ Users อีกที แต่ในระบบ OSX นั้นจะไม่มีตัวอักษรบ่งบอกไดรฟ์ (drive letter) เหมือนในระบบวินโดวส์

วินโดวส์

ในระบบวินโดวส์ นั้น เราจะสามารถเปิดดูไฟล์และโฟลเดอร์ได้โดยใช้ Windows Explorer (เรียกใช้งานโดยดับเบิลคลิก ""My Computer หรือกดปุ่ม win+e)



OSX

ส่วนใน OSX นั้น เราสามารถเรียกดูไฟล์และโฟลเดอร์ได้โดยใช้ Finder (เรียกใช้งานได้โดยคลิกที่ไอคอน Finder - ไอคอนรูปใบหน้าที่อยู่ด้านล่างซ้ายมือของบาร์)



1.2 เทอร์มินัล

การใช้งานคอมพิวเตอร์ในปัจจุบันนั้น กระทำผ่านทางส่วนติดต่อผู้ใช้งาน แบบกราฟฟิกที่ซับซ้อน (GUIs) โดยการใช้งานคีย์บอร์ด เมาส์และทัชสกรีนในการติดต่อกับปุ่ม หรือคอนโทรลแบบต่างๆ ที่แสดงบนหน้าจอ แต่ก็ได้ไม่เป็นเช่นนี้เสมอไป ก่อนที่เราจะมี GUI เราใช้งานผ่านทาง เทอร์มินัล - ส่วนติดต่อคอมพิวเตอร์แบบอักษร โดยแทนที่เราจะจัดการปุ่มบนหน้าจอ เราก็สั่งงานด้วยคำสั่งและรอรับผลลัพธ์ตอบกลับ เสมือนเรากำลังคุยกับคอมพิวเตอร์ ถึงแม้ในโลกของคอมพิวเตอร์ปัจจุบันดูเหมือนจะทอดทิ้งเทอร์มินัลให้เป็นเสมือนวัตถุโบราณ แต่ความจริงก็คือว่าเทอร์มินัล ยังเป็นส่วนติดต่อพื้นฐานที่ภาษาโปรแกรมส่วนใหญ่ใช้ในการติดต่อกับกับคอมพิวเตอร์ โดยภาษาโกนั้น ก็ได้แตกต่างแต่อย่างใด ดังนั้นก่อนที่จะเริ่มเขียนโปรแกรมด้วยภาษาโก เราควรมีความเข้าใจพื้นฐานในการทำงานของเทอร์มินัล

วินโดวส์

บนวินโดวส์ นั้นสามารถเรียกใช้งานเทอร์มินัล (หรือเรียกอีกอย่างว่าคอมมานด์ไลน์) โดยกดปุ่ม windows key + r (กดปุ่ม windows key ค้างไว้แล้วกดปุ่ม r) แล้วพิมพ์ cmd.exe แล้วกด enter คุณควรจะได้เห็นหน้าจอสีดำนี้อย่างภาพ:

A screenshot of a Windows Command Prompt window. The title bar at the top shows the path 'C:\Windows\system32\cmd.exe'. The main area of the window is black with white text. It displays the Windows version information: 'Microsoft Windows [Version 6.1.7601] Copyright (c) 2009 Microsoft Corporation. All rights reserved.' followed by the current directory 'C:\Users\caleb>'.

โดยปรกติคอมมานด์ไลน์จะเริ่มทำงานในไดเรกทอรีโฮม ของคุณ (ในกรณีของผมนั่นคือ C:\Users\caleb) เราสั่งงานโดยพิมพ์คำสั่งลงไปแล้วกดปุ่ม enter ลองพิมพ์คำสั่ง dir ดู ซึ่งเป็นคำสั่งที่ใช้ในการแสดงรายการของสิ่งที่อยู่ภายในไดเรกทอรี ซึ่งเราควรเห็นผลลัพธ์ดังนี้

```
C:\Users\caleb>dir
Volume in drive C has no label.
Volume Serial Number is B2F5-F125
```

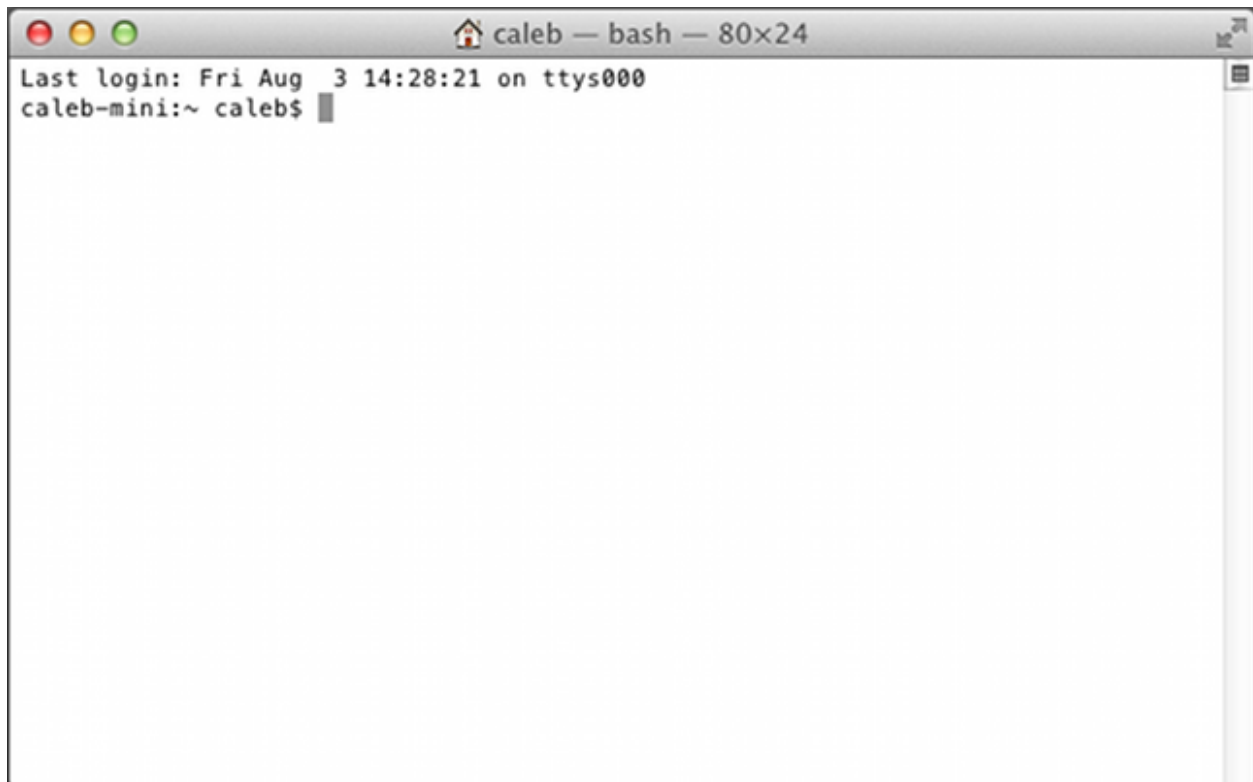
โดยจะตามด้วยรายการของไฟล์และโฟลเดอร์ที่อยู่ในไดเรกทอรีโฮม ของคุณ และเราสามารถที่จะเปลี่ยนไดเรกทอรีโดยใช้คำสั่ง cd ยกตัวอย่างเช่น อาจจะมีโฟลเดอร์ชื่อ Desktop เราสามารถเข้าไปดูข้างในได้ด้วยคำสั่ง cd Desktop ต่อด้วยคำสั่ง dir และหากต้องการกลับไปไดเรกทอรีโฮม สามารถทำได้โดยใช้ชื่อไดเรกทอรี พิเศษชื่อ .. (จุดสองอันติดกัน):

```
cd ..
```

ส่วนจุดเดียวนั้นใช้เป็นเครื่องหมายแทนโฟลเดอร์ปัจจุบันที่เราอยู่ (หรือเรียกอีกชื่อว่า working folder) ดังนั้นการเรียก `cd` . จึงไม่ได้ให้ผลลัพธ์อะไร ยังมีอีกหลายคำสั่งที่เราสามารถใช้งานได้ แต่เพียงเท่านี้ก็เพียงพอสำหรับการเริ่มต้นแล้ว

OSX

บน OSX นั้นเทอร์มินัลสามารถใช้งานได้โดยไปที่ Finder → Applications → Utilities → Terminal ท่านควรจะเห็นหน้าต่างดังนี้:



โดยปกติเทอร์มินัลจะเริ่มทำงานในไดเรกทอรีโฮม ของคุณ (ในกรณีของผมคือ `/Users/caleb`) เราสามารถสั่งงานได้โดยพิมพ์คำสั่งลงไปตามด้วย enter ลองพิมพ์คำสั่ง `ls` ดู จะเป็นการใช้ในการเรียกดูเนื้อหาของไดเรกทอรีใดๆ ซึ่งควรจะได้ผลลัพธ์ดังนี้

```
caleb-mini:~ caleb$ ls
Desktop Downloads Movies Pictures
Documents Library Music Public
```

ระบบจะแสดงรายการของไฟล์และโฟลเดอร์ต่างๆ ที่อยู่ในไดเรกทอรีโฮม ของคุณ(ในกรณีนี้ตัวอย่างจะมีแต่โฟลเดอร์ไม่มีไฟล์) เราสามารถเปลี่ยนไดเรกทอรีได้โดยใช้คำสั่ง `cd` ยกตัวอย่างเช่น ในกรณีที่คุณมีโฟลเดอร์ชื่อ Desktop เราสามารถดูสิ่งที่อยู่ข้างในได้โดยการสั่ง `cd Desktop` แล้วตามด้วย `ls` และหากต้องการกลับไปไดเรกทอรีโฮม คุณสามารถทำได้โดยใช้ชื่อ directory พิเศษชื่อ `..` (จุดสองอันติดกัน):

```
cd ..
```

ส่วนจุดเดียวนั้นใช้เป็นเครื่องหมายแทนโฟลเดอร์ปัจจุบันเราอยู่ (หรือที่รู้จักอีกชื่อว่า working folder) ดังนั้นการสั่ง `cd .` จึงไม่ได้ให้ผลลัพธ์อะไร ยังมีอีกหลายคำสั่งที่เราสามารถใช้งานได้ แต่เพียงเท่านี้ก็เพียงพอสำหรับการเริ่มต้นแล้ว

1.3 Text Editors

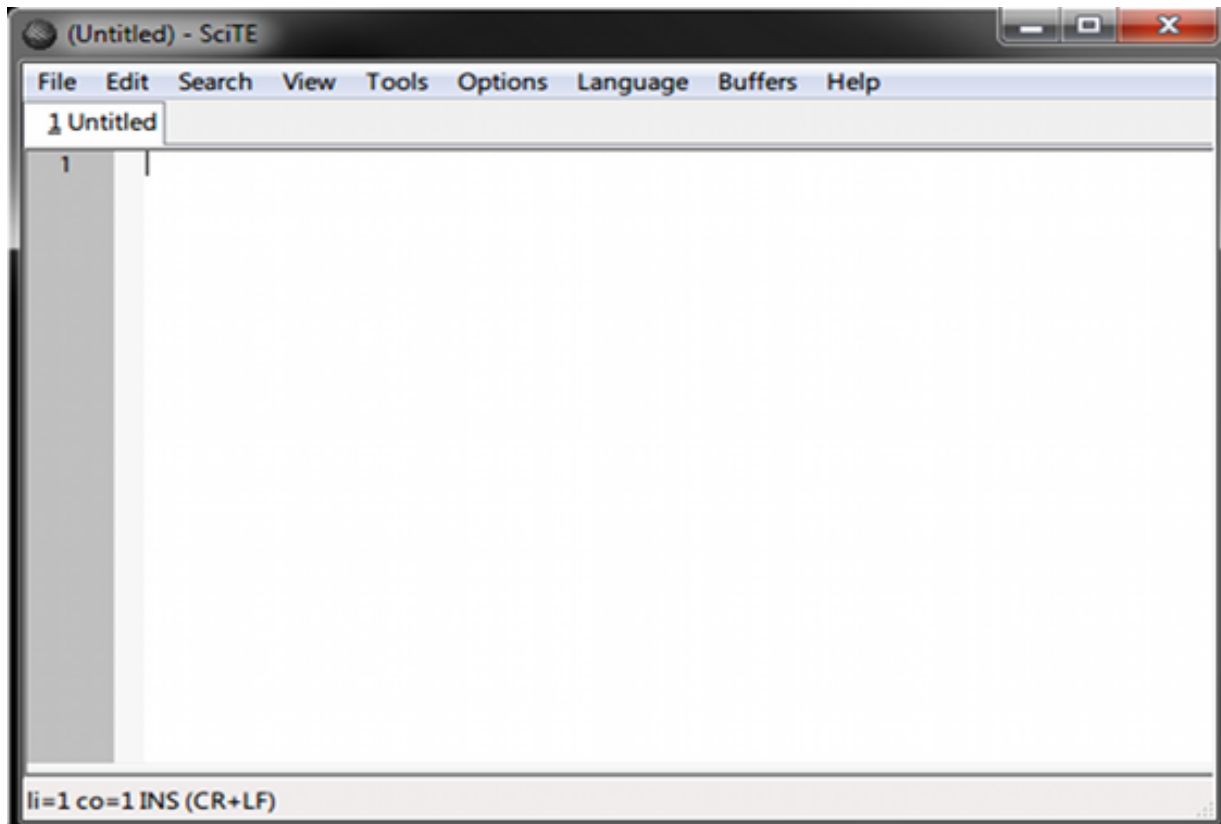
เครื่องมือหลักที่โปรแกรมเมอร์ใช้ในการเขียนโปรแกรมคือ text editor โดยจะทำงานคล้ายๆ กับโปรแกรมประมวลผลคำ (Microsoft Word, Open Office, ...) แต่แตกต่างกันตรงที่จะไม่สามารถกำหนดรูปแบบให้ตัวอักษรได้ (ไม่มีตัวหนา, ตัวเอียง, ...) โดยจะทำงานกับตัวอักษรเท่านั้น ทั้งระบบวินโดวส์ และ OSX จะมี text editor ติดตั้งมาด้วยแล้ว แต่ก็มีข้อจำกัดมากมาย ดังนั้นจึงขอแนะนำให้ติดตั้งตัวที่ดีกว่า

เพื่อเป็นการอำนวยความสะดวกให้การติดตั้งทำได้ง่าย โปรแกรมติดตั้งจะอยู่ที่เว็บไซต์ของหนังสือ:

<http://www.golang-book.com/> โดยโปรแกรมติดตั้งจะทำทั้งติดตั้งชุดเครื่องมือต่างๆ ของภาษาโกและ setup สภาพแวดล้อมในการทำงาน พร้อมทั้งติดตั้ง text editor ให้ด้วย

วินโดวส์

ในระบบวินโดวส์ ตัวติดตั้งจะทำการติดตั้ง text editor ที่ชื่อ Scite โดยเราสามารถเปิดใช้งานโปรแกรมภายหลังติดตั้งเสร็จโดยไปที่ Start → All Programs → Go → Scite โดยโปรแกรมจะเป็นดังภาพ



text editor จะประกอบไปด้วยพื้นที่ว่างที่ให้เราเอาไว้พิมพ์ โดยด้านซ้ายมือจะแสดงหมายเลขบรรทัด และด้านล่างจะของหน้าต่างจะมีแถบแสดงสถานะที่ใช้แสดงข้อมูลของไฟล์และที่อยู่ปัจจุบัน (จากภาพแถบสถานะจะแสดงให้เราเห็นว่า ตอนนี้กำลังอยู่ที่บรรทัดที่ 1 คอลัมน์ที่ 1 โดยข้อความถูก insert ในแบบปรกติ และเรากำลังใช้การขึ้นบรรทัดใหม่ตามแบบของ วินโดวส์)

เราสามารถเปิดไฟล์โดยเลือก File → Open และ browse หาไฟล์ที่เราต้องการ และสามารถบันทึกไฟล์โดยเลือก File → Save หรือ File → Save As

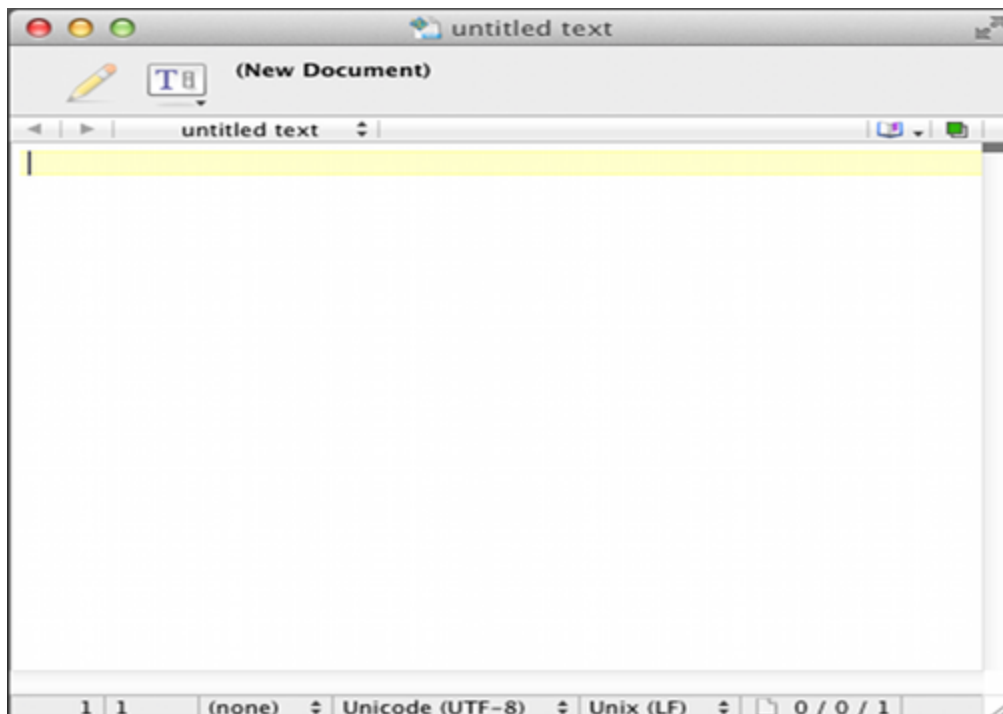
ในขณะที่เราใช้งาน text editor นั้น การเรียนรู้การใช้งาน shortcuts ต่างๆ เอาไว้จะมีประโยชน์มาก โดยทุกๆ เมนูจะแสดง shortcuts อยู่ทางด้านขวามือ ต่อไปนี้คือคำสั่งที่มักจะถูกใช้งานบ่อยๆ:

- Ctrl + S – บันทึกไฟล์
- Ctrl + X – ตัดข้อความที่ถูกเลือกนำไปใส่ไว้ในคลิปบอร์ดทำให้สามารถนำไปวางลง ที่อื่นได้ในภายหลัง
- Ctrl + C – คัดลอกข้อความที่ถูกเลือก
- Ctrl + V – วางข้อความที่เก็บอยู่ในคลิปบอร์ด
- ปุ่มลูกศรใช้ในการเลื่อนเคอร์เซอร์ไปในที่ต่างๆ ปุ่ม Home เพื่อกระโดดไปยังต้นบรรทัด และปุ่ม End เพื่อไปยังท้ายบรรทัด

- กดปุ่ม shift ค้างไว้ พร้อมกับกดปุ่มลูกศร (ปุ่ม Home หรือปุ่ม End) เพื่อเลือกข้อความโดยไม่ต้องใช้เมาส์ลาก
- Ctrl + F – เปิดกล่องค้นหาเพื่อค้นหาข้อความในเนื้อหาของไฟล์นั้น

OSX

สำหรับระบบ OSX โปรแกรมติดตั้งจะติดตั้ง text editor ชื่อ Text Wrangler



และก็คล้ายๆ กับโปรแกรม Scite บน window โปรแกรม Text Wrangler จะมีพื้นที่ให้พิมพ์ข้อความลงไป โดยสามารถเปิดไฟล์ได้โดยเลือก File → Open และบันทึกไฟล์โดยเลือก File → Save หรือ File → Save As และต่อไปนี่คือ shortcuts ที่มีประโยชน์ (Command คือปุ่ม ⌘)

- Command + S – บันทึกไฟล์
- Command + X – ตัดข้อความที่ถูกเลือกนำไปใส่ไว้ในคลิปบอร์ด ทำให้สามารถนำไปวางที่อื่นได้ในภายหลัง
- Command + C – คัดลอกข้อความที่ถูกเลือก
- Command + V – วางข้อความที่เก็บอยู่ในคลิปบอร์ด
- ปุ่มลูกศรใช้ในการเลื่อนเคอร์เซอร์ไปในที่ต่างๆ
- Command + F – เปิดกล่องค้นหาเพื่อค้นหาข้อความในเนื้อหาของไฟล์นั้น

1.4 เครื่องมือต่างๆ ของภาษาโก

ภาษาโก เป็นภาษาที่ต้องทำการคอมไพล์ก่อน (compiled programming language) ซึ่งหมายความว่าซอร์สโค้ด (โปรแกรมที่เราเขียน) จะต้องถูกแปลงไปเป็นภาษาที่คอมไพเลอร์สามารถเข้าใจได้ ดังนั้นก่อนที่จะเราสามารถเขียนโปรแกรมภาษาโกได้ เราต้องมีคอมไพเลอร์ของภาษาโกเสียก่อน

โปรแกรมติดตั้งจะติดตั้งภาษาโกให้เราแบบอัตโนมัติ โดยเราจะใช้เวอร์ชัน 1 (สามารถหาข้อมูลเพิ่มเติมได้ที่ <http://www.golang.org>) เสร็จแล้วให้ตรวจสอบว่าทุกอย่างทำงานได้ถูกต้อง โดยการเปิดเทอร์มินัลแล้วพิมพ์ดังนี้:

```
go version
```

เราควรจะเห็นผลลัพธ์ดังนี้:

```
go version go1.0.2
```

โดยหมายเลขเวอร์ชันนี้อาจจะต่างออกไปเล็กน้อย ถ้าระบบแสดงข้อผิดพลาดว่า ไม่รู้จักคำสั่งแล้วหละก็ให้ลอง restart เครื่องคอมพิวเตอร์ดู

ชุดโปรแกรมของภาษาโก จะประกอบไปด้วยคำสั่ง และคำสั่งย่อยต่างๆ มากมาย หากเราต้องการดูรายการของคำสั่งเหล่านั้น สามารถทำได้โดยการพิมพ์คำสั่ง:

```
go help
```

ซึ่งในบทความต่อไป เราจะได้เห็นว่าคำสั่งเหล่านี้จะถูกใช้งานอย่างไร

บทที่ 2: โปรแกรมแรกของคุณ (Your First Program)

โปรแกรมแรกที่คุณเขียนในภาษาอื่นๆเสมอ ที่เรียกว่าโปรแกรม “Hello World” มันเป็นโปรแกรมแบบง่ายๆที่แสดงผลคำว่า Hello World ที่เทอร์มินัลของคุณ ตอนนี้ เรามาเขียนด้วย โก กันเถอะ

เริ่มจากสร้างโฟลเดอร์ใหม่ในที่ ที่คุณสามารถเก็บโปรแกรมของคุณไว้ได้ ซึ่งตัวติดตั้งที่คุณใช้ในบทที่ 1 ได้สร้างโฟลเดอร์ชื่อว่า โก ไว้ใน โฮมไดเรคทอรี ของคุณ จากนั้นให้สร้างโฟลเดอร์ชื่อว่า ~/Go/src/golang-book/chapter2 (โดยที่ ~ หมายถึง โฮมไดเรคทอรี ของคุณ) คุณสามารถใช้คำสั่งตามนี้ที่เทอร์มินัล:

```
mkdir Go/src/golang-book
mkdir Go/src/golang-book/chapter2
```

จากนั้นใช้โปรแกรม text editor ของคุณ พิมพ์ตามนี้

```
package main

import "fmt"

// this is a comment

func main() {
    fmt.Println("Hello World")
}
```

ตรวจสอบให้แน่ใจว่าไฟล์ของคุณเหมือนกับที่แสดงให้เห็นนี้ และบันทึกเป็นไฟล์ชื่อ main.go ในโฟลเดอร์ที่เพิ่งสร้างขึ้นมา จากนั้นเปิดเทอร์มินัลใหม่แล้วพิมพ์ตามนี้

```
cd Go/src/golang-book/chapter2
go run main.go
```

คุณน่าจะได้เห็น Hello World แสดงที่เทอร์มินัลของคุณ โดยคำสั่ง go run จะนำเอาไฟล์ที่ต่อจากคำสั่ง (ซึ่งคั่นด้วยเว้นวรรค) มาคอมไพล์เป็นตัวที่จะสามารถ execute ได้และบันทึกลงไปในไดเรคทอรีชั่วคราว(temporary directory) และสั่งรันโปรแกรมนั้น ถ้าคุณไม่เห็น Hello World คุณอาจจะทำอะไรผิดพลาดอย่างตอนที่เขียนโปรแกรม โดย โก คอมไพล์เลอร์ จะให้คำแนะนำคุณเกี่ยวกับจุดที่ผิดพลาดเหมือนอย่างที่คอมไพล์เลอร์อื่นๆทำ และ โก คอมไพล์เลอร์นั้น เข้มงวดมากๆและไม่ใจดีกับความผิดพลาดใดๆ

อ่านโปรแกรม โก อย่างไร(How to Read a Go Program)

เรามาดูที่รายละเอียดของโปรแกรมนี้กัน โดยโปรแกรม โก นั้น อ่านจากบนลงล่าง ซ้ายไปขวา(เหมือนหนังสือ) อย่างบรรทัดแรกบอกว่า:

```
package main
```

สิ่งนี้ทำให้รู้ถึง “การประกาศ แพกเกจ”(package declaration) และทุกๆโปรแกรม โก จะต้องเริ่มต้นด้วยการประกาศแพกเกจ

แพกเกจ คือแนวทางของ โก ในการจัดระเบียบการนำโค้ดมาใช้ซ้ำ โดยโปรแกรม โก มีสองแบบ:

คือแบบที่ execute ได้ และแบบ ไลบรารี โดยโปรแกรมที่ execute ได้ คือแบบของโปรแกรมที่เราสามารถรันได้ตรงๆจากเทอร์มินัล (ใน Windows พวกมันจะลงท้ายด้วย .exe) ส่วน ไลบรารี เป็นคอลเลกชันของโค้ดที่เราสามารถนำไปรวมไว้และเรียกใช้ได้ ในโปรแกรมอื่นๆ ซึ่งเราจะพูดถึง ไลบรารี กันในรายละเอียดภายหลัง แต่ในตอนนี้เราแค่ต้องแน่ใจว่า เราได้รวมบรรทัดนี้ไว้ในโปรแกรมใดก็ตามที่เราเขียน

บรรทัดต่อไปเป็นบรรทัดว่าง ซึ่งคอมพิวเตอร์แทนค่า newlines เป็นอักขระพิเศษ(หรือ serveral characters) newlines,space และ tabs ถูกรับรู้ว่าเป็น whitespace (เพราะว่าเราไม่เห็นมัน) โดยส่วนใหญ่ โก ไม่สนใจ whitespace เราใช้มันเพื่อให้โปรแกรมง่ายต่อการอ่าน(คุณสามารถเอาบรรทัดนี้ออกได้และโปรแกรมจะยังคงทำงานตามเดิม) จากนั้นเราเห็นแบบนี้

```
import "fmt"
```

คำหลัก(keyword) `import` คือการบอกว่าเราจะรวมโค้ดจาก แพกเกจ อื่นเข้ามาในโปรแกรมของเราอย่างไร

`fmt`(คำย่อของ `format`) คือแพกเกจ ที่รวมเครื่องมือจัดรูปแบบสำหรับ input และ output และเราเพิ่งเรียนรู้เกี่ยวกับ แพกเกจ มาเมื่อครู่นี้ คุณคิดว่า แพ้มของ แพกเกจ `fmt` น่าจะมีการประกาศแพกเกจอยู่ด้านบนด้วยหรือไม่?

ขอให้รู้ว่า `fmt` ด้านบนนั้นถูกรอับไว้ด้วยเครื่องหมายฟันหนู(double quotes) การใช้เครื่องหมายฟันหนูลักษณะนี้บอกให้รู้ว่า ตัวหนังสือในนั้นเป็นชนิดของ expression

ใน โก นั้น สตริง คือลำดับของอักขระ(ตัวอักษร,ตัวเลข,สัญลักษณ์,...) ที่มีความยาวที่แน่นอน สตริง จะถูกอธิบายในรายละเอียดในบทต่อไป แต่ในตอนนี้สิ่งสำคัญกว่าคือให้จำให้ขึ้นใจว่าการขึ้นต้นด้วยอักขระ " จะต้องถูกปิดท้ายด้วย " เสมอ และอักขระใดๆในระหว่างสองอักขระนี้จะถือว่าเป็น สตริง(ตัวอักขระ " เองไม่ได้เป็นส่วนหนึ่งของสตริงด้วย)

ในบรรทัดที่เริ่มต้นด้วย `//` ขอให้รับทราบว่าคือคอมเม้นท์ และการคอมเม้นท์จะถูกเมินโดย โก คอมไพล์เลอร์ และมันจะเป็นประโยชน์ต่อตัวคุณ(หรือใครก็ตามที่จะนำซอสโค้ดของคุณมาดูต่อ) โก มี 2 รูปแบบการคอม

เม้นท์: `//` จะคอมเม้นท์ตัวหนังสือทั้งหมดตั้งแต่ `//` ไปจนถึงท้ายบรรทัด และ `/* */` จะคอมเม้นท์ทุกอย่างระหว่าง `*` ทั้งสองตัว (และอาจรวมหลายบรรทัดได้)

หลังจากนี้คุณจะเห็นการประกาศฟังก์ชัน

```
func main() {  
    fmt.Println("Hello World")  
}
```

ฟังก์ชันคือการสร้างบล็อกของโปรแกรมใน โก โดยพวกมันมี inputs และ outputs และ มีการเรียงร้อยขั้นตอนการทำงานหรือที่เรียกว่า statements ซึ่งถูก execute ในลักษณะคำสั่ง

ฟังก์ชันทั้งหมดจะเริ่มต้นด้วยคำหลักว่า `func` ตามด้วยชื่อของฟังก์ชัน(ในตัวอย่างนี้คือ `main`) พารามิเตอร์ตั้งแต่ ศูนย์หรือมากกว่านั้นถูกครอบไว้ด้วยเครื่องหมายวงเล็บ การระบุค่าเป็นทางเลือกที่จะมีหรือไม่มีก็ได้ และ ส่วนบอดี(body) หรือเนื้อหา ถูกครอบไว้ด้วยวงเล็บปีกกาฟังก์ชันนี้ไม่มีพารามิเตอร์ ไม่มีการระบุค่าใดๆ และมีแค่ statment เดียว ชื่อ `main` เป็นฟังก์ชันพิเศษ เพราะมันเป็นฟังก์ชันที่ถูกเรียกเมื่อคุณ execute โปรแกรม ส่วนสุดท้ายของโปรแกรมของเราคือบรรทัดนี้

```
fmt.Println("Hello World")
```

statement นี้ถูกสร้างขึ้นจากสามองค์ประกอบ หนึ่ง เราเข้าถึงฟังก์ชันอื่นภายใน แพคเกจของ `fmt` ที่ชื่อ `Println` (นั่นคือ `fmt.Println` โดย `Println` หมายถึงพิมพ์ที่ละบรรทัด) จากนั้นเราสร้าง สตริงใหม่ว่า `Hello World` และ invoke (หรือจะเรียกว่า call หรือ execute ก็ได้) ฟังก์ชันโดยส่งสตริงเป็นอาร์กิวเมนต์ตัวแรกและตัวเดียว

ณ จุดจุดนี้ เราพร้อมแล้วที่จะเห็นระบบคำแบบใหม่ๆอีกมากและคุณอาจจะรู้สึกงงนิดๆบางครั้งการตั้งใจอ่านโปรแกรมของคุณดูๆอาจมีประโยชน์ การอ่านโปรแกรมที่เพิ่งเขียนไปอาจจะเป็นแบบนี้:

สร้างโปรแกรมใหม่ที่ execute ได้ ซึ่งอ้างถึง ไลบรารี ชื่อ `fmt` และประกอบไปด้วยหนึ่งฟังก์ชันชื่อว่า `main` มันไม่มีอาร์กิวเมนต์ ไม่ระบุค่าอะไรเลย และตามมาด้วยการเรียกใช้ฟังก์ชัน `Println` ที่อยู่ใน แพคเกจ `fmt` และ เรียกใช้มันโดยใส่หนึ่งอาร์กิวเมนต์ คือสตริง `Hello World`

ฟังก์ชัน `Println` คือตัวที่ทำงานจริงๆในโปรแกรมนี้ คุณสามารถค้นหาเกี่ยวกับมันได้โดยพิมพ์ตามนี้ที่เทอร์มินอล

```
godoc fmt Println
```

แล้วคุณก็จะได้เห็นตามนี้

Println formats using the default formats for its operands and writes to standard output. Spaces are always added between operands and a new line is appended. It returns the number of bytes written and any write error encountered.

โก เป็นภาษาโปรแกรมมิ่งที่ทำเอกสารดีมาก แต่การทำเอกสารนี้อาจสร้างความสับสนในการเข้าใจ เว้นแต่ว่าคุณจะคุ้นเคยกับภาษาโปรแกรมมิ่งมามากพอ แต่กระนั้นก็ตามคำสั่ง `godoc` ก็ยังคงมีประโยชน์อย่างมากและเป็นที่ที่ดีที่คุณจะเริ่มเมื่อคุณมีปัญหา

กลับไปฟังกัซันของเรากันดีกว่า เอกสารนี้กำลังบอกคุณว่า ฟังก์ชัน `Println` จะส่งอะไรก็ตามที่คุณให้มันไปแสดงที่ standard output - ชื่อของ output ของเทอร์มินัลที่คุณกำลังทำงานอยู่

ฟังก์ชันนี้ทำให้เห็น `Hello World`

ในบทต่อไปเราจะลงไปดูในรายละเอียดว่า โก เก็บและแสดงสิ่งๆที่เหมือนกับ `Hello World` อย่างไรโดยเรียนรู้เกี่ยวกับ ประเภทข้อมูล(Types)

ปัญหาท้าทาย(Problems)

- whitespace คืออะไร?
- comment คืออะไร? และสองวิธีที่จะเขียน comment คืออะไร?
- โปรแกรมของเราเริ่มต้นด้วย `package main` แล้วแฟ้ม `fmt` ควรจะเริ่มด้วยอะไร?
- เราได้ใช้ฟังก์ชัน `Println` ที่ถูกประกาศไว้ใน `package fmt` ถ้าเราต้องการใช้ฟังก์ชัน `Exit` จาก แพกเกจ `os` เราควรจะต้องทำอะไร?
- แก้ไขโปรแกรมที่เราได้เขียนไปแล้ว โดยแทนที่จะพิมพ์ว่า `Hello World` ให้พิมพ์แทนด้วย `Hello, my name is` ตามด้วยชื่อคุณ

บทที่ 3: ไทป์ (Types)

ในบทที่ผ่านมาเราได้ใช้ตัวแปรซึ่งมีประเภทข้อมูล (data type) เป็นสตริง (string) เพื่อเก็บข้อความ “Hello World” มาแล้ว ประเภทข้อมูล ทำหน้าที่จำแนกค่าของตัวแปร (value) ออกเป็นหมวดหมู่ ใช้อธิบายการทำงานที่เราจะสามารถกระทำกับตัวแปรนั้นๆ ได้ รวมถึงระบุวิธีการจัดเก็บค่าของตัวแปรนั้น การทำความเข้าใจหลักการเกี่ยวกับประเภทข้อมูล อาจจะเป็นเรื่องยาก ดังนั้นเราจะลองมองเรื่องนี้ด้วยมุมมองต่างๆ ก่อนที่เราจะเข้าไปเรียนรู้ว่าเราจะใช้ภาษาโกในการจัดการประเภทข้อมูล เหล่านี้อย่างไร

บางครั้งนักปรัชญาจะจำแนกความแตกต่างระหว่าง “ประเภท” กับ “ชื่อเรียก” ออกจากกัน ตัวอย่างเช่น สมมุติว่าคุณมีสุนัขชื่อ แม็กซ์ แม็กซ์เป็นชื่อเรียก (เพื่อบอกให้ชัดว่ากำลังพูดถึงสุนัขตัวไหน) และสุนัขเป็นประเภท (เพื่ออธิบายถึงคุณสมบัติทั่วไปของสิ่งที่กำลังพูดถึง) “สุนัข” หรือ “ความเป็นสุนัข” จะอธิบายถึงคุณสมบัติที่สุนัขทุกตัวพึงจะมีเป็นพื้นฐาน ซึ่งถ้าอธิบายถึงความเป็นเหตุเป็นผลกันของ “ประเภท” กับ “ชื่อเรียก” ก็อย่างเช่น สุนัขทุกตัวจะมีสี่ขา แม็กซ์เป็นสุนัข ดังนั้นแม็กซ์จึงมีสี่ขาด้วย เราสามารถอธิบายเรื่องประเภทข้อมูลที่อยู่ในเรื่องการเขียนโปรแกรมได้ด้วยวิธีเดียวกัน เช่น ตัวแปรที่มีประเภทข้อมูล เป็น สตริง จะมีความยาวของตัวอักษรเสมอ ตัวแปร x เป็นสตริงดังนั้น x จึงมีความยาวของตัวอักษรด้วย

ในทางคณิตศาสตร์ เรามักจะพูดถึงเรื่อง เซต (Set) กันอยู่บ่อยๆ ยกตัวอย่างเช่น \mathbb{R} (ชุดของตัวเลขที่เป็นจำนวนจริง) หรือ \mathbb{N} (ชุดของตัวเลขที่เป็นจำนวนนับ (counting number - เป็นจำนวนเต็มบวกเสมอ)) สมาชิกแต่ละตัวที่อยู่ในเซตเหล่านี้จะมีคุณสมบัติที่เหมือนกันทุกตัวในเซต นั้นๆ ยกตัวอย่างเช่น จำนวนนับทุกตัวล้วนสัมพันธ์เชื่อมโยงกัน เช่น มีจำนวนนับ A, B และ C , $A + (B + C)$ จะเท่ากับ $(A + B) + C$ หรือ $A \times (B \times C)$ จะเท่ากับ $(A \times B) \times C$ เสมอ ประเภทข้อมูลก็เหมือนกัน ค่าของตัวแปรที่มีประเภทข้อมูลเหมือนกัน จะมีคุณสมบัติเหมือนกันเสมอ

โก เป็นภาษาเขียนโปรแกรมประเภท static type (statically typed programming language) นั่นคือตัวแปรทุกตัวจะต้องระบุประเภทข้อมูลให้กับตัวแปรเสมอ และไม่สามารถเปลี่ยนแปลงประเภทข้อมูลของตัวแปรนั้นได้หลังจากระบุไปแล้ว ในช่วงแรก static type อาจจะดูยุ่งยากซับซ้อน และอาจต้องใช้เวลามากกว่าที่คุณจะแก้โปรแกรมจนสามารถคอมไพล์ได้ แต่ด้วยการที่ต้องระบุประเภทข้อมูล นี่เองจะช่วยให้เราสามารถอธิบายได้ว่าโปรแกรมของเราจะทำอะไร และดักจับความผิดพลาดของโปรแกรมได้สะดวกขึ้น

โก ได้จัดเตรียมประเภทข้อมูล บางส่วนพร้อมให้เราใช้งานได้ทันที เรามาดูกันว่า ประเภทข้อมูล ที่ โก เตรียมไว้ให้นั้นมีรายละเอียดอย่างไรบ้าง

ตัวเลข (Numbers)

โก ได้เตรียมประเภทข้อมูล แบบต่างๆ ไว้สำหรับข้อมูลประเภทตัวเลข โดยทั่วไปเราสามารถจำแนกตัวเลขออกเป็นสองชนิดหลักๆ คือ ตัวเลขจำนวนเต็ม (integer) และ ตัวเลขทศนิยม (floating-point)

เลขจำนวนเต็ม (Integers)

ตัวเลขจำนวนเต็มเป็นตัวเลขที่ไม่มีจุดทศนิยม (ตอบแบบกำปั้นทุบดินซะม๊าด) เช่น -3, -2, -1, 0, 1, 2, 3 เป็นต้น ซึ่งในคอมพิวเตอร์จะไม่ได้ใช้ตัวเลขฐาน 10 อย่างที่เราใช้กันในชีวิตประจำวัน แต่จะใช้ตัวเลขฐาน 2 เพื่อแทนค่าตัวเลขในระบบ

ถ้าเราจะเขียนตัวเลขจำนวนสิบตัว เราจะต้องปวดหัวกับการไล่ลำดับของตัวเลขจำนวนหนึ่ง สมมุติว่าเริ่มตั้งแต่ 2 ตัวถัดไปจะเป็น 3, 4, 5, ... ไปเรื่อยๆ ตัวเลขที่ต่อจาก 9 ก็คือ 10 ไล่ไปจนถึง 99 และต่อจากตัวเลข 99 ก็จะเป็น 100 เป็นแบบนี้ไปเรื่อยๆ ในระบบคอมพิวเตอร์ทำแบบเดียวกัน แต่คอมพิวเตอร์จะใช้ตัวเลข 0 และ 1 เท่านั้น ตัวอย่างเช่น 0, 1, 10, 11, 100, 101, 110, 111 เป็นแบบนี้ไปเรื่อยๆ ความแตกต่างระหว่างระบบตัวเลขที่มนุษย์เราใช้กับระบบตัวเลขที่คอมพิวเตอร์ใช้ ก็คือ ในระบบคอมพิวเตอร์ เลขจำนวนเต็มจะมีการกำหนดขนาดไว้อย่างชัดเจน (ขึ้นอยู่กับประเภทข้อมูล ที่ระบุ) ซึ่งคือจำนวนหลัก (digit) ที่จะเก็บตัวเลขได้ เช่น integer มีขนาดเท่ากับ 4 บิต ตัวเลขที่จะเป็นไปได้ เช่น 0000, 0001, 0010, 0011, 0100 เป็นต้น จนกระทั่งเราใช้ไปจนเต็มความจุที่ integer นี้จะเก็บได้แล้ว ระบบคอมพิวเตอร์ก็จะย้อนกลับไปใช้ตำแหน่งเริ่มต้นอีกครั้ง (แล้วเราก็จะเจอพฤติกรรมประหลาดๆ จากปรากฏการณ์นี้จนกลายเป็นบั๊กที่หาไม่เจอ T_T)

ประเภทข้อมูล Integer ที่ โก เตรียมไว้ให้ ได้แก่ uint8, uint16, uint32, uint64, int8, int16, int32, และ int64 ตัวเลข 8, 16, 32, 64 ที่เราเห็นนั้น เป็นตัวบอกว่าประเภทข้อมูลแต่ละตัวสามารถเก็บข้อมูลได้กี่บิต ในขณะที่ uint จะหมายถึง เลขจำนวนเต็มบวก (unsigned integer ซึ่งรวมถึงเลขศูนย์ด้วย) ส่วน int จะหมายถึง เลขจำนวนเต็ม (signed integer ซึ่งคือเลขลบ เลขศูนย์ และเลขบวก) ยังมีประเภทข้อมูลอีกสองชนิดที่เพิ่มขึ้นมาคือ byte ซึ่งจะเหมือนกับ uint8 และ rune ซึ่งจะเหมือนกับ int32 ไบต์ (bytes) เป็นหน่วยวัดที่ใช้กันทั่วไปในระบบคอมพิวเตอร์ (เช่น 1 ไบต์ = 8 บิต, 1025 ไบต์ = 1 กิโลไบต์, 1024 กิโลไบต์ = 1 เมกะไบต์,... เป็นต้น) ซึ่ง byte ที่มากับ โก จะถือเป็นประเภทข้อมูลอีกชนิดหนึ่ง นอกจากนี้ยังมีประเภทข้อมูลที่มีขนาดขึ้นกับสถาปัตยกรรมคอมพิวเตอร์โดยตรง ได้แก่ uint, int, และ uintptr

โดยทั่วไปแล้ว ถ้าต้องใช้ประเภทข้อมูลที่เป็นเลขจำนวนเต็ม เราควรใช้ประเภทข้อมูล `int` ก็พอ

เลขทศนิยม (Floating Point Numbers)

เลขทศนิยม เป็นตัวเลขที่มีจุดทศนิยม (เลขจำนวนจริง) เช่น 1.234, 123.4, 0.00001234, รวมถึง 12340000 การแสดงค่าเลขจำนวนจริงบนระบบคอมพิวเตอร์ค่อนข้างซับซ้อน และยังไม่ใช้เรื่องจำเป็นที่เราจะต้องเข้าใจเกี่ยวกับมัน ตอนนี้ขอให้อ่านไว้ว่า

- เลขทศนิยมเป็นตัวเลขที่ไม่เที่ยงตรง ตัวอย่างเช่น $1.01 - 0.99$ แล้วได้ผลลัพธ์เท่ากับ 0.0200000000000000018 ผลลัพธ์นี้ใกล้เคียงกับที่เราคิดไว้ (เท่ากับ 0.02) แต่มันไม่ใช่ค่าเดียวกัน (อธิบายก็คือ ถ้าเราปัดเศษผลลัพธ์ตัวแรกเหลือทศนิยมสองตำแหน่ง ก็จะได้เท่ากับ 0.02 เหมือนกัน)
- คล้ายกับเลขจำนวนเต็ม (integer) เลขทศนิยมก็จะมีขนาดที่แน่นอนเหมือนกัน (32 บิต หรือ 64 บิต) การใช้เลขทศนิยมที่มีขนาดใหญ่ขึ้น จะเพิ่มความเที่ยงตรงของตัวเลขให้สูงขึ้น
- ค่าของเลขทศนิยมสามารถแสดงในรูปแบบ “ไม่ใช่ตัวเลข (NaN – Not a Number)” ซึ่งอาจจะเกิดจาก 0 หาร 0 (0/0) อีกรูปแบบหนึ่งคือ เลขอนันต์เชิงบวก เลขอนันต์เชิงลบ (infinity number)

โก เตรียมประเภทข้อมูลสำหรับเลขทศนิยมไว้ให้สองแบบคือ `float32` และ `float64` (`float64` จะเที่ยงตรงเป็นสองเท่าของ `float32`) นอกจากนั้น โก ยังเตรียมประเภทข้อมูลสำหรับเลขเชิงซ้อน (complex number ซึ่งเป็นระบบตัวเลขที่มีเลขจินตภาพ (imaginary number) เป็นส่วนประกอบ) ไว้ด้วย คือ `complex64` และ `complex128` สำหรับการใช้งานทั่วไปเราควรใช้ `float64` เพื่อทำงานกับตัวเลขทศนิยม

ตัวอย่าง

มาลองเขียนโปรแกรมเพื่อลองใช้งานระบบตัวเลขใน โก กัน เริ่มจากสร้างไฟล์เดอร์ชื่อ `chapter3` และสร้างไฟล์ชื่อ `main.go` ซึ่งมีโค้ดตามด้านล่างนี้

```
package main

import "fmt"

func main() {
    fmt.Println("1 + 1 =", 1 + 1)
}
```

ถ้าลองสั่งให้โปรแกรมทำงาน คุณควรเห็นผลลัพธ์ดังนี้

```
$ go run main.go

1 + 1 = 2
```

สังเกตว่าโปรแกรมนี้น่าจะคล้ายกับโปรแกรมที่เราเคยเขียนในบทที่ 2 ทั้งชื่อแพ็คเกจและไลบรารี ที่อิมพอร์ตเข้ามา การประกาศฟังก์ชันและยังใช้ฟังก์ชัน `Println` อีกด้วย แต่ในโปรแกรมนี้นั้นแทนที่จะแสดงคำว่า `Hello World` ออกหน้าจอ เราสั่งให้โปรแกรมแสดงผลเป็นข้อความ `1 + 1 =` แล้วตามด้วยผลการคำนวณผลบวกระหว่าง `1 + 1` ในคำสั่งนี้มีส่วนประกอบอยู่ 3 ส่วน คือเลขจำนวนเต็ม `1`, ตัวดำเนินการ `+` (การบวก), และเลขจำนวนเต็ม `1` อีกตัวหนึ่ง ที่นี้ลองทำแบบเดียวกันแต่เปลี่ยนไปใช้เลขทศนิยมแทนดังนี้

```
fmt.Println("1 + 1 =", 1.0 + 1.0)
```

สังเกตว่าเราใช้ `.0` เพื่อบอก โก ว่าเป็นเลขทศนิยม เมื่อสั่งให้โปรแกรมทำงาน ก็จะได้ผลลัพธ์เหมือนเดิม

นอกจากตัวดำเนินการ `+` (การบวก) โก ยังเตรียมตัวดำเนินการอื่นๆ ไว้ให้ดังนี้

+	การบวก
-	การลบ
*	การคูณ
/	การหาร

%	การหาเศษ
---	----------

สตริง (Strings)

อย่างที่เราได้เห็นในบทที่ 2 สตริง คือการเรียงกันของตัวอักษรด้วยความยาวที่แน่นอน เพื่อใช้เป็นตัวแทน “ข้อความ” สตริงใน Go นั้นเป็นการประกอบกันขึ้นมาจากไบต์ ซึ่งหนึ่งไบต์คือหนึ่งตัวอักษร (สำหรับตัวอักษรในภาษาอื่นๆ อย่างเช่น ภาษาจีนนั้น อาจจะต้องใช้มากกว่าหนึ่งไบต์) สตริงที่เป็นสัญลักษณ์สามารถสร้างได้ด้วยการใส่สัญลักษณ์ใน “ (double quote) เช่น “Hello World” หรือใน ` (back ticks) เช่น `Hello World` สิ่งที่แตกต่างกันคือ เราสามารถใส่อักขระขึ้นบรรทัดใหม่ (newline character) หรืออักขระพิเศษอื่นๆ อยู่ใน “ ได้ (ถ้าอ่านภาษาอังกฤษ ใช้คำว่า cannot ซึ่งผมว่าน่าจะผิด เพื่อความแน่ใจเลยลองเขียนโปรแกรมทดสอบแล้ว ยืนยันว่าภาษาอังกฤษน่าจะพิมพ์ผิดครับ) ตัวอย่างเช่น ใช้ \n สำหรับการขึ้นบรรทัดใหม่ และ \t สำหรับการตั้งระยะข้อความ (tab character)

เราสามารถหาความยาวของข้อความที่อยู่ในสตริงด้วยเมธอด len เช่น len(“Hello World”) หรือเข้าถึงตัวอักษรใดๆ ในข้อความที่อยู่ในสตริงก็สามารถทำได้โดยใช้ “Hello World”[1] หรือเอาข้อความสองข้อความมาประกอบกัน ทำได้โดย “Hello “ + “World” เอาละมาลองแก้โปรแกรมที่เคยเขียนก่อนหน้านี้เพื่อทดลองสิ่งที่เราได้เรียนรู้เกี่ยวกับสตริงกัน

```
package main

import "fmt"

func main() {
    fmt.Println(len("Hello World"))
    fmt.Println("Hello World"[1])
    fmt.Println("Hello " + "World")
}
```

มีข้อสังเกตบางอย่างที่เราควรรู้

- ช่องว่างถือเป็นตัวอักษรด้วย ดังนั้นในผลการหาความยาวของ “Hello World” จึงเท่ากับ 11 ไม่ใช่ 10 และในคำสั่งที่สาม เราจึงใช้ “Hello ” แทนที่จะใช้ “Hello”
- การระบุตัวอักษรในสตริง จะเริ่มจาก 0 ไม่ใช่ 1 ซึ่งในคำสั่งที่เราใช้ [1] จะได้ผลลัพธ์เป็นตัวอักษรซึ่งอยู่ในตำแหน่งที่สองนั่นเอง ไม่ใช่ตำแหน่งที่หนึ่ง นอกจากนี้ลองสังเกตว่าเมื่อสั่งงานโปรแกรม “Hello World”[1] จะให้ผลลัพธ์เป็น 101 แทนที่จะเป็น e นั่นเป็นเพราะตัวอักษรนั้นจะถูกแทนด้วยไบต์ (และอย่าลืมว่าไบต์คือ integer)

- การนำข้อความมาประกอบกันเราใช้เครื่องหมาย `+` เหมือนการบวกตัวเลข คอมไพเลอร์ของ โก จะประมวลผลโดยขึ้นอยู่กับประเภทข้อมูลที่เรียกใช้ เมื่อ นิพจน์ทั้งซ้ายและขวาของเครื่องหมาย `+` เป็นสตริง คอมไพเลอร์จะรู้ว่าคุณหมายถึงการนำข้อความสองข้อความมาประกอบกัน ไม่ใช้การบวกตัวเลข

บูลีน (Booleans)

ค่าบูลีน (ตั้งชื่อตาม George Boole) คือบิตที่ใช้แทนค่า `true` และ `false` (หรือ `on` และ `off`) ตรรกะที่สามารถใช้กับค่าบูลีนมีอยู่สามแบบคือ

<code>&&</code>	<code>and</code>
<code> </code>	<code>or</code>
<code>!</code>	<code>not</code>

มาดูตัวอย่างการใช้งานกัน

```
package main

import "fmt"

func main() {
    fmt.Println(true && true)
    fmt.Println(true && false)
    fmt.Println(true || true)
    fmt.Println(true || false)
    fmt.Println(!true)
}
```

ผลลัพธ์ที่ได้หลังจากสั่งโปรแกรมทำงานเป็นดังนี้

```
$ go run main.go

true
false
true
true
false
```

ปกติเราจะใช้ตารางความจริง (truth table) เพื่ออธิบายว่าแต่ละคำสั่งทำงานมีการทำงานอย่างไร

นิพจน์ (expression)	ผลลัพธ์
true && true	true
true && false	false
false && true	false
false && false	false

นิพจน์ (expression)	ผลลัพธ์
true true	true
true false	true
false true	true
false false	false

นิพจน์ (expression)	ผลลัพธ์
!true	false
!false	true

ที่พูดมาทั้งหมดเป็นประเภทข้อมูลอย่างง่ายที่สุด ซึ่งจะถูกนำไปใช้เป็นรากฐานของประเภทข้อมูลอื่นๆ ที่จะตามมาในภายหลัง

ปัญหาท้ายบท

- ในระบบคอมพิวเตอร์มีการจัดเก็บค่าของเลขจำนวนเต็มอย่างไร

- เราได้เรียนรู้ว่าในระบบเลขฐาน 10 ตัวเลขที่มีค่ามากที่สุดสำหรับจำนวนเต็มหนึ่งตำแหน่งคือ 9 และ 99 สำหรับจำนวนเต็มสองตำแหน่ง ถ้าเป็นระบบเลขฐาน 2 สำหรับค่ามากที่สุดที่มีสองตำแหน่งก็จะเป็น 11 (หรือเท่ากับ 3 ในระบบเลขฐาน 10) 111 (หรือเท่ากับ 7 ในระบบเลขฐาน 10) สำหรับสามตำแหน่ง และ 1111 (หรือเท่ากับ 15 ในระบบเลขฐาน 10) สำหรับสี่ตำแหน่ง ให้หาว่าเลขมากที่สุดในระบบเลขฐาน 2 ที่มีจำนวนแปดตำแหน่งคืออะไร (คำใบ้: $10^1 - 1 = 9$, $10^2 - 1 = 99$)
- ให้ลองเขียนโปรแกรมเพื่อคำนวณผลคูณของ 32132 x 42452 แล้วแสดงผลออกมาทางหน้าจอ (ให้ใช้เครื่องหมาย * แทน x สำหรับการคูณในตอนที่เราเขียนโปรแกรม)
- สตริงคืออะไร คุณจะหาความยาวของสตริงได้อย่างไร
- ให้หาผลลัพธ์ของ (true && false) || (false && true) || !(false && false)

บทที่ 4: ตัวแปร (Variable)

มาถึงตอนนี้เราได้เขียน Code กันมาแบบใช้ค่าตัวเลขบ้าง ตัวอักษรบ้าง ใน Code ซึ่งมันยังดูไม่ค่อยจะหล่อเท่าไรนัก คราวนี้เราลองมาปรับให้ Code มันดูหล่อขึ้นโดยใช้ 2 แนวทาง คือ Variables และ Control Flow Statement ซึ่งในบทนี้เราจะว่ากันด้วยเรื่อง ตัวแปร (Variable) ของ Go

ตัวแปร (Variable) เป็น ตัวเก็บค่า โดยประกอบไปด้วย 2 ส่วน คือ กำหนดชนิดของตัวแปร และ ชื่อตัวแปรที่จะให้เก็บค่า ลองมาปรับแก้ Code จากบทที่ 2 Hello World ให้ดูหล่อขึ้นจากเดิมกันด้วย ตัวแปร

```
package main

import "fmt"

func main() {
    var x string = "Hello World"
    fmt.Println(x)
}
```

จาก Code ด้านบน จะเห็นว่า ชุดตัวอักษร Hello World จาก Code บทที่ 2 ยังคงอยู่ แต่แทนที่เราจะ ส่งชุดตัวอักษรนั้นไปแสดงผลผ่าน function Println ตรงๆ เราทำการส่งชุดตัวอักษรผ่านตัวแปรไปแทน การสร้างตัวแปรใน Go เราเริ่มต้นด้วยการประกาศด้วย var แล้วตามด้วย ชื่อตัวแปร (x) ต่อด้วย ชนิดของตัวแปร (string) และสุดท้ายก็คือการส่งค่าไปเก็บไว้ในตัวแปร (Hello World) ถ้าจะให้หล่อกว่า Code ด้านบน เราสามารถเขียนออกมาได้ในแบบนี้

```
package main

import "fmt"

func main() {
    var x string
    x = "Hello World"
    fmt.Println(x)
}
```

ตัวแปรใน Go จะคล้ายๆ กับตัวแปรทางคณิตศาสตร์ แต่จะมีส่วนที่แตกต่างกันในรายละเอียดดังนี้

จาก Code เมื่อเราเห็นเครื่องหมาย เท่ากับ “=” เราก็จะอ่านว่า “x มีค่าเท่ากับ Hello World” ซึ่งก็ไม่ได้ผิด อะไรในการอ่านเช่นนั้น แต่จะดูดีกว่าถ้าเราอ่านแบบนี้ “x รับค่าชุดตัวอักษร Hello World” หรือ “x

ถูกใส่ค่าชุดตัวอักษร Hello World” ซึ่งความต่างในการอ่านนี้เป็นเรื่องที่สำคัญเพราะว่าตัวแปรสามารถจะถูกเปลี่ยนแปลงค่าของมันไปได้ตลอดเวลา ยกตัวอย่าง เช่น

```
package main

import "fmt"

func main() {
    var x string
    x = "first"
    fmt.Println(x)
    x = "second"
    fmt.Println(x)
}
```

ซึ่งเราสามารถเขียนแบบนี้ก็ได้

```
var x string
x = "first "
fmt.Println(x)
x = x + "second"
fmt.Println(x)
```

ถ้าเราอ่าน Code แบบคณิตศาสตร์มันจะฟังดูแปลกๆ แต่ถ้าเราลองอ่านให้ถูกต้องตามชุดของคำสั่งดังนี้ เมื่อเราเห็น `x = x + "second"` เราจะต้องอ่านว่า “นำค่าเดิมของ x ต่อด้วยชุดตัวอักษร second แล้วเก็บค่าไว้ใน x” ซึ่งจะต้องกระทำการฟังขวาของเครื่องหมาย “=” ให้เรียบร้อยก่อน แล้วจึงเก็บค่าที่ได้ไปยังฝั่งซ้ายของเครื่องหมาย “=”

รูปแบบ `x = x + y` เป็นรูปแบบทั่วไปของการเขียน Code สำหรับ Go เราสามารถใช้ “+=” แทนได้ ดังนั้นจาก `x = x + "second"` เราสามารถเขียนใหม่ได้เป็น `x += "second"` ซึ่งก็ได้ผลลัพธ์ออกมาแบบเดียวกัน และสามารถใช้กับ Operators อื่นๆ ได้เช่นกัน

อีกหนึ่งความแตกต่างระหว่าง Go และคณิตศาสตร์ คือ Go ใช้เครื่องหมาย “==” เพื่อเทียบว่า เท่ากัน หรือไม่ โดยจะให้ ค่าผลลัพธ์ออกมาเป็น Boolean ยกตัวอย่างเช่น

```
var x string = "hello"
var y string = "world"
fmt.Println(x == y)
```

ค่าผลลัพธ์ที่จะได้ออกมา คือ false เพราะ คำว่า hello ไม่ใช่ค่าเดียว หรือเหมือนกับคำว่า world ลองมาดูอีกตัวอย่าง

```
var x string = "hello"  
var y string = "hello"  
fmt.Println(x == y)
```

ในกรณีนี้ค่าที่ได้ออกมาจะเป็น true เพราะ ชุดตัวอักษรทั้งสองเหมือนกัน

ใน Go เราสามารถสร้างตัวแบบ string ได้อีกวิธีดังนี้

```
x := "Hello World"
```

จะเห็นว่าเราใส่เครื่องหมาย : เข้าไปข้างหน้า = โดยไม่ต้องประกาศชนิดของตัวแปรก็ได้เช่นกัน เพราะ Compiler ของ Go สามารถที่จะระบุชนิดของตัวแปรได้จากค่าที่รับเข้ามาเก็บไว้ ซึ่ง Compiler จะมองว่ามีค่าเทียบเท่ากับ

```
var x = "Hello World"
```

เราสามารถใช้วิธีการนี้กับตัวแปรชนิดอื่นๆ ได้เช่นกัน ยกตัวอย่างเช่น

```
x := 5  
fmt.Println(x)
```

แนะนำว่าควรใช้รูปแบบนี้เมื่อมีโอกาส

วิธีการตั้งชื่อตัวแปร

เรื่องการตั้งชื่อตัวแปรเป็นเรื่องที่สำคัญเรื่องหนึ่งของการพัฒนาซอฟต์แวร์ ชื่อตัวแปรต้องเริ่มต้นด้วย ตัวอักษร และประกอบไปด้วยตัวอักษร ตัวเลข และเครื่องหมาย _ (underscore) สำหรับ Compiler ของ Go ไม่สนใจว่าเราจะตั้งชื่อตัวแปรมาแบบไหน การตั้งชื่อตัวแปรนั้นเป็นเรื่องสำคัญที่ใช้ในการสื่อสาร และอธิบาย Code ดังนั้นจะต้องใส่ใจ และให้ความสำคัญกับการตั้งชื่อตัวแปรให้เข้าใจได้ง่าย ไม่ต้องตีความหรือคาดเดา ยกตัวอย่างเช่น

```
x := "Max"
fmt.Println("My dog's name is", x)
```

จะเห็นว่า การตั้งชื่อตัวแปรว่า `x` ไม่สื่อความหมายของตัวแปรทำอะไร ลองเปลี่ยนเป็นแบบนี้

```
name := "Max"
fmt.Println("My dog's name is", name)
```

หรือ

```
dogsName := "Max"
fmt.Println("My dog's name is", dogsName)
```

ในกรณีนี้เราสามารถเลือกใช้วิธีการตั้งชื่อตัวแปรเป็นคำๆ โดยใช้รูปแบบที่เรียกว่า Camel Case โดยเริ่มต้นคำแรกด้วยชุดอักษรตัวพิมพ์เล็กทั้งหมด แล้วคำต่อๆ ไปตัวแรกของแต่ละคำเป็นตัวพิมพ์ใหญ่ ลักษณะจะเหมือนหลังอุฐ

ขอบเขตของตัวแปร (Scope)

ย้อนกลับไปดู Code ที่เราพูดคุยกันตอนเริ่มต้นบทนี้อีกครั้ง

```
package main

import "fmt"

func main() {
    var x string = "Hello World"
    fmt.Println(x)
}
```

เราสามารถเขียนได้อีกแบบ

```
package main

import "fmt"

var x string = "Hello World"

func main() {
    fmt.Println(x)
}
```

เมื่อเราย้ายตัวแปร ออกมาไว้ข้างนอก function main แล้วนั้น function อื่นๆ ก็สามารถที่จะเรียกใช้งานตัวแปรนั้นได้เช่นกัน ยกตัวอย่างเช่น

```
var x string = "Hello World"

func main() {
    fmt.Println(x)
}

func f() {
    fmt.Println(x)
}
```

function f สามารถเรียกใช้งานตัวแปร x ได้ แต่ถ้าเราเขียนแบบนี้

```
func main() {
    var x string = "Hello World"
    fmt.Println(x)
}

func f() {
    fmt.Println(x)
}
```

เมื่อเรา Run จะเจอ Error ดังนี้

```
.\main.go:11: undefined: x
```

Compiler จะบอกว่าไม่พบการประกาศตัวแปร x ใน function f ซึ่งตัวแปร x ถูกประกาศไว้ในเฉพาะ function main เท่านั้น ใน Go ใช้เครื่องหมาย { } (ปีกกา) เป็นตัวกำหนดขอบเขตของตัวแปร ที่ function ต่างๆ จะสามารถอ้างอิง หรือเรียกใช้งานได้

ค่าคงที่ (Constant)

ใน Go เราสามารถกำหนดค่าคงที่ขึ้นมาได้ โดยวิธีการกำหนดใช้แบบเดียวกับการกำหนดค่าตัวแปร แต่เปลี่ยนจากการประกาศ `var` เป็น `const` ซึ่งเมื่อเราประกาศตัวแปรใดๆ เป็นค่าคงที่แล้ว ตัวแปรตัวนั้นจะไม่สามารถ ถูกเปลี่ยนแปลงค่าได้ ยกตัวอย่างเช่น

```
package main

import "fmt"

func main() {
    const x string = "Hello World"
    fmt.Println(x)
}
```

เมื่อเราลอง

```
const x string = "Hello World"
x = "Some other string"
```

ผลลัพธ์ที่ได้จะเป็น Error ดังนี้

```
.\main.go:7: cannot assign to x
```

ค่าคงที่ เหมาะสำหรับการค่าที่จะถูกอ้างอิง หรือเรียกใช้งานบ่อยๆ เช่น ค่า Pi ใน math package ถูกกำหนดเป็นค่าคงที่ เป็นต้น

กำหนดชุดตัวแปร (Defining Multiple Variables)

เราสามารถกำหนดค่าของตัวแปร `var` หรือ ค่าคงที่ `const` แบบเป็นชุดได้ด้วยวิธีการแบบนี้

```
var (  
    a = 5  
    b = 10  
    c = 15  
)
```

ตัวอย่าง

ตัวอย่าง Code ที่รับค่าตัวเลขเข้ามา แล้วทำการคูณ 2

```
package main  
  
import "fmt"  
  
func main() {  
    fmt.Print("Enter a number: ")  
    var input float64  
    fmt.Scanf("%f", &input)  
  
    output := input * 2  
  
    fmt.Println(output)  
}
```

จาก Code ตัวอย่าง เราเรียกใช้ function Scanf จาก fmt package เพื่อรับค่า input เข้ามา ซึ่งจะขอ
ยก การอธิบาย Scanf ไว้ในบทต่อไป สำหรับตอนนี้เรารู้ก่อนเพียงว่าเราสามารถกำหนดค่าตัวแปร input
โดย Scanf จากการกรอกเข้ามาของผู้ใช้ได้

ปัญหาท้าทาย

- เราสามารถสร้างตัวแปรขึ้นมาใหม่ได้ 2 วิธี ได้แก่?

- ค่าของ x หลังจาก that run มีค่าเท่ากับเท่าไร?: $x := 5$; $x += 1$
- ขอบเขตคืออะไร และเราสามารถจะกำหนดขอบเขตของตัวแปรได้อย่างไร?
- var และ const แตกต่างกันอย่างไ?
- จงเขียนโปรแกรมที่แปลงค่าอุณหภูมิจาก Fahrenheit เป็น Celsius โดยเริ่มต้นจาก $(C = (F - 32) * 5/9)$
- จงเขียนโปรแกรมที่แปลงค่าความยาวจาก ฟุต เป็น เมตร เมื่อกำหนด 1 ฟุต = 0.3048 เมตร

บทที่ 5 :โครงสร้างการควบคุม (Control Structures)

ถึงตอนนี้เราก็ได้รู้วิธีการใช้งานตัวแปร ซึ่งก็น่าจะได้เวลาที่จะลองเขียนอะไรที่เป็นขั้นกว่าขึ้นไปอีก เริ่มจากสร้างโปรแกรมที่พิมพ์ 1 ถึง 10 ถ้าใช้สิ่งที่เราเรียนมาก็จะได้หน้าตาออกมาประมาณนี้

```
package main

import "fmt"

func main() {
    fmt.Println(1)
    fmt.Println(2)
    fmt.Println(3)
    fmt.Println(4)
    fmt.Println(5)
    fmt.Println(6)
    fmt.Println(7)
    fmt.Println(8)
    fmt.Println(9)
    fmt.Println(10)
}
```

หรือว่าแบบนี้

```
package main

import "fmt"

func main() {
    fmt.Println(`1
2
3
4
5
6
7
8
9
10`)
}
```


แต่กระนั้นตัวโปรแกรมข้างบนทั้งคู่ก็ยังดูเย็นเยื่อ เราต้องหาทางชักทางมาจัดการงานที่มันดูซ้ำๆ กันแล้วละ

5.1 For

ซึ่ง for จะช่วยให้เราจัดการกับชุดคำสั่งแบบเดิมๆ ที่เกิดขึ้นซ้ำๆ แล้วเราก็จับโปรแกรมก่อนหน้านี้มาปรับแก้ใหม่โดยใช้ for ก็จะได้หน้าตาแบบนี้

```
package main

import "fmt"

func main() {
    i := 1
    for i <= 10 {
        fmt.Println(i)
        i = i + 1
    }
}
```

เป็นไงหล่อขึ้นมัย จากตัวอย่างข้างบน เริ่มจากสร้างตัวแปรชื่อ i มาเก็บตัวเลขที่ต้องการจะแสดง ถัดมาก็ใช้ for มาตรวจสอบเงื่อนไขว่าเป็น true หรือ false สุดท้ายก็จะเป็นส่วนภายใต้ปีกกา ที่จะถูกเรียกให้ทำงาน ถ้าเงื่อนไขนั้นถูกต้อง โดยเจ้าตัว for loop นั้นก็จะทำงานแบบนี้:

- i. เริ่มจากประเมินตัว `i <= 10` (i มีค่าน้อยกว่าหรือเท่ากับ 10) ซึ่งถ้าเงื่อนไขเป็นจริงก็จะไปทำงานในส่วนที่อยู่ภายในปีกกา ถ้าไม่ใช่ก็จะกระโดดข้ามไปทำงานต่อในส่วนที่อยู่หลังปีกกา (ในที่นี้ก็จะจบการทำงานไปเลยเพราะไม่มีอะไรต่อจาก for loop แล้ว อือ)
- ii. หลังจากโปรแกรมทำการส่งประมวลผล คำสั่งที่อยู่ภายในปีกกาเสร็จแล้ว ก็จะวนกลับไปจุดเริ่มต้นของ for แล้วเริ่มต้นทำข้อ 1 ใหม่

สำหรับตัว `i <= 10` มีความสำคัญโคตรๆ เพราะถ้าไม่มีแล้วตัว for loop ก็จะได้ต่างว่าค่าจากการประเมินเป็นจริง ทำให้ตัวโปรแกรมทำงานไม่จบไม่สิ้น (เรียกว่า infinite loop)

จากแบบฝึกหัดนี้ถ้าเราไล่ตัวโปรแกรม โดยคิดซะว่าตัวเองเป็น คอมพิวเตอร์ ก็น่าจะเห็นเหตุการณ์ที่เกิดขึ้นแบบนี้:

- สร้างตัวแปร i โดยมีค่าเป็น 1
- ตรวจสอบค่า $i \leq 10$ โอ้โอ ยังเป็นจริงอยู่
- พิมพ์ i
- กำหนดค่า i เท่ากับ $i + 1$ (ตอนนี้ i ก็เท่ากับ 2)
- ตรวจสอบค่า $i \leq 10$ โอ้โอ ยังเป็นจริงอยู่
- พิมพ์ i
- กำหนดค่า i เท่ากับ $i + 1$ (ตอนนี้ i ก็เท่ากับ 3)
- ...
- กำหนดค่า i เท่ากับ $i + 1$ (ตอนนี้ i ก็เท่ากับ 11)
- ตรวจสอบค่า $i \leq 10$ โอ้โอ ไม่เป็นจริงซะแล้ว
- ไม่เหลืออะไรให้ทำซะแล้ว จบ

ตัวภาษาอื่นๆก็จะมีประเภทของ loops ให้ใช้แบบว่าหลากหลายมากมาย (while, do, until, foreach, ...) แต่สำหรับ Go มีให้ใช้อย่างเดียวแต่สามารถใช้ได้หลายท่า ตัวอย่างก่อนหน้านี้ ก็จะเขียนได้อีกแบบ ตัวอย่างเช่น:

```
func main() {  
    for i := 1; i <= 10; i++ {  
        fmt.Println(i)  
    }  
}
```

ตอนนี้เราก็มีสมาชิกใหม่เพิ่มเข้ามาพร้อมกับ เครื่องหมาย ; ตัวแรกก็จะเป็นการกำหนดค่าตั้งต้นตัวแปร ถัดมาก็จะเป็นการตรวจสอบเงื่อนไข สุดท้ายก็เป็นการเพิ่มค่าให้กับตัวแปร (การเพิ่มค่าให้ทีละ 1 ก็เห็นได้บ่อยๆ แต่ก็มีทำพิเศษให้ใช้คือ ++ หรือว่าจะลดค่าลงทีละ 1 ก็มีเหมือนกันใช้ --)

เดี๋ยวเราก็จะว่า for loop จะมีทำอะไรให้ใช้อีกบ้างในบทถัดๆไป ตอนนี้ยาวไป!!!!!!

5.2 If

มาๆ มาแต่งองค์ทรงเครื่องให้โปรแกรมเราอีก แทนที่จะให้พิมพ์ค่า 1 - 10 ออกมาแค่นั้นก็ให้มันบอกด้วยว่าเป็นเลขคู่(even) หรือคี่(odd) โดยให้พิมพ์ ต่อท้ายออกมาด้วย อย่างเช่น:

```
1 odd
2 even
3 odd
4 even
5 odd
6 even
7 odd
8 even
9 odd
10 even
```

เอาไงดี?

อย่างแรกสุด ก็หาวิธีที่จะแยกให้ออกว่าตัวเลขที่ได้เป็นจำนวนคี่หรือคู่ ทางออกที่ง่ายที่สุดก็จับหารด้วย 2 ซะ ถ้าหารลงตัวก็เป็นจำนวนคู่ ถ้าเหลือเศษก็จำนวนคี่ แล้วใน Go ละ! เราจะหาเศษที่เหลือจากการหารยังไง? ใช้ % ครับ ตัวอย่าง `1 % 2` เท่ากับ 1, `2 % 2` เท่ากับ 0, `3 % 2` เท่ากับ 1

ถัดมาก็ต้องหาวิธีที่จะแยกการทำงานให้ออกจากกัน โดยให้ขึ้นกับเงื่อนไข ตรงนี้เองที่เราจะใช้ **if**:

```
if i % 2 == 0 {
    // even
} else {
    // odd
}
```

ซึ่ง **if** ก็มีความคล้ายคลึง **for** ที่ว่า จะประกอบไปด้วยส่วนที่เป็นเงื่อนไขแล้วตามด้วย บล็อกภายใต้เครื่องหมายปีกกา แต่จะมี **else** เป็น option เพิ่มเข้ามา ตัวอย่าง ถ้าเงื่อนไขหลัง **if** เป็น **true** บล็อกที่ตามหลังก็就会被สั่งทำงาน ถ้าไม่ใช่ก็จะข้ามไป แต่ถ้ามี **else** อยู่ด้วยก็จะไปสั่งบล็อกของ **else** ทำงาน

ยังๆ ยังไม่หมดแค่นั้น **if** ยังมี **else if** ให้ใช้อีก ตัวอย่าง:

```
if i % 2 == 0 {  
    // divisible by 2  
} else if i % 3 == 0 {  
    // divisible by 3  
} else if i % 4 == 0 {  
    // divisible by 4  
}
```

เงื่อนไขทั้งหมดจะถูกตรวจสอบในลักษณะจากบนลงล่าง ถ้ามีเงื่อนไขไหนเป็นจริง บล็อกที่ตามหลังก็就会被สั่งทำงาน ตัวอื่นๆที่เหลือก็จะไม่ถูกเรียก (ตัวอย่าง ให้ค่าตัวแปร *i* เท่ากับ 8 ซึ่งหารด้วย 4 และ 2 ลงตัว แต่บล็อก **// divisible by 4** จะไม่ถูกสั่งทำงาน เพราะ บล็อก **// divisible by 2** ถูกสั่งทำงานไปก่อนหน้าแล้ว)

ลองเอาทั้งหมดทั้งมวลมารวมกัน:

```
func main() {  
    for i := 1; i <= 10; i++ {  
        if i % 2 == 0 {  
            fmt.Println(i, "even")  
        } else {  
            fmt.Println(i, "odd")  
        }  
    }  
}
```

มาลองไล่โปรแกรมดู:

- สร้างตัวแปรชื่อ **i** เป็นชนิด **int** แล้วให้ค่าเป็น **1**
- ถ้า **i** น้อยกว่า หรือเท่ากับ **10** เป็นจริงก็ให้กระโดดเข้าไปในบล็อก
- ถ้า เศษที่เหลือจาก **i ÷ 2** เท่ากับ **0** เป็น เท็จ ให้กระโดดไปที่บล็อก **else**
- พิมพ์ **i** แล้วตามด้วย **odd**
- เพิ่มค่าให้กับ **i** (ประโยคที่ตามหลังเงื่อนไข **i <= 10**)
- ถ้า **i** น้อยกว่า หรือเท่ากับ **10** เป็นจริงก็ให้กระโดดเข้าไปในบล็อก

- ถ้า เศษที่เหลือจาก $i \div 2$ เท่ากับ 0 เป็น จริง ให้กระโดดไปที่บล็อก if
- พิมพ์ i แล้วตามด้วย even
- ...

เครื่องหมาย % (remainder operator) เหมือนกับว่าได้ตัดขาดจากเราหลังจากจบ ชั้นประถม (จริงหรือเค้าสอนกันด้วย? อ้อ) กลายเป็นว่าสำคัญสุดๆเลยเวลาที่เขียนโปรแกรม จะพบเห็นได้บ่อยๆเลย ตั้งแต่ตารางข้อมูลที่แบ่งแถบสีตามแนวนอน(zebra striping tables) หรือใช้ในการแบ่งกลุ่มของชุดข้อมูล

5.3 Switch

สมมุติว่าเราอยากเขียนโปรแกรมที่พิมพ์ชื่อตัวเลข โดยอาศัยวิชากันหีบที่เรียนผ่านๆมา ก็น่าจะได้หน้าตาประมาณนี้:

```
if i == 0 {
    fmt.Println("Zero")
} else if i == 1 {
    fmt.Println("One")
} else if i == 2 {
    fmt.Println("Two")
} else if i == 3 {
    fmt.Println("Three")
} else if i == 4 {
    fmt.Println("Four")
} else if i == 5 {
    fmt.Println("Five")
}
```

ถ้าเราเขียนออกมาในทำนองนี้มันก็ออกจะน่าเบื่อไปหน่อย Go ก็มีตัวอื่นมาให้ใช้เรียกว่า **switch** (statement) ซึ่งก็เขียนออกมาอีกทีได้ในรูปแบบนี้:

```

switch i {
case 0: fmt.Println("Zero")
case 1: fmt.Println("One")
case 2: fmt.Println("Two")
case 3: fmt.Println("Three")
case 4: fmt.Println("Four")
case 5: fmt.Println("Five")
default: fmt.Println("Unknown Number")
}

```

ตัว switch(statement) ก็จะตั้งต้นด้วยคำว่า switch นี่แหละ แล้วก็ตามด้วย เอ็กซ์เพรสชัน (ในที่นี้คือ i) ลำดับสุดท้ายก็เป็นเคสตามแบบต่างๆ โดยค่าของ เอ็กซ์เพรสชัน ก็จะถูกเปรียบเทียบกับ เอ็กซ์เพรสชัน ของแต่ละเคส ซึ่งถ้าเปรียบเทียบแล้วเท่ากัน ชุดคำสั่งที่อยู่หลังเครื่องหมาย : ก็จะถูกเรียกใช้งาน

ซึ่งก็เหมือนกับ if โดยที่แต่ละกรณี จะถูกเช็คจากบนลงล่าง และเคสตัวแรกที่เปรียบเทียบแล้วเท่ากันก็จะถูกเรียก ยังไม่หมด switch ยังสามารถใช้ ดีฟอลต์ เคส(default case) ในกรณีที่ไม่มีเคสไหนตรงกับค่าที่นำมาเปรียบเทียบเลย(ซึ่งก็คล้ายคล้ายคลึงกับ else ใน if)

พวก control flow statements หลักๆก็จะมีประมาณเท่านี้ ส่วนตัวอื่นๆก็จะมีเสริมเข้ามาในบทถัดไป

ปัญหาท้าทาย

- i. โปรแกรมข้างล่างแสดงผลอะไรออกมาจะ

```

i := 10
if i > 10 {
    fmt.Println("Big")
} else {
    fmt.Println("Small")
}

```

- ii. เขียนโปรแกรมที่พิมพ์ตัวเลขที่หารด้วย 3 ลงตัว เริ่มตั้งแต่ 1 ถึง 100 (3, 6, 9, ฯลฯ)

- iii. เขียนโปรแกรมที่พิมพ์ตัวเลขเริ่มตั้งแต่ 1 ถึง 100 แต่จำนวนที่หารด้วย 3 ลงตัว ให้พิมพ์ "Fizz"
และจำนวนที่หารด้วย 5 ลงตัว ให้พิมพ์ "Buzz" สำหรับ จำนวนที่หารด้วย 3 และ 5 ลงตัว ให้พิมพ์
"FizzBuzz"

บทที่ 6: Arrays, Slices และ Map

ในบทที่ 3 เราเคยเรียนเกี่ยวกับชนิดข้อมูลพื้นฐานของภาษาโก บทนี้เราจะศึกษาชนิดข้อมูลอีก 3 ประเภท ได้แก่ อาร์เรย์ สไลซ์ และแมป

อาร์เรย์

อาร์เรย์คือลำดับเชิงตัวเลขของสมาชิกซึ่งมีชนิดเดียวกันด้วยความยาวคงที่ ในภาษาโกพวกมันมีลักษณะดังนี้:

```
var x [5]int
```

x เป็นตัวอย่างของอาร์เรย์ซึ่งประกอบด้วยจำนวนเต็ม 5 จำนวน เราทดลองรันโปรแกรมข้างล่าง:

```
package main

import "fmt"

func main() {
    var x [5]int
    x[4] = 100
    fmt.Println(x)
}
```

ผลที่ได้ควรเป็น:

```
[0 0 0 0 100]
```

x[4] = 100 ควรอ่านว่า “กำหนดให้สมาชิกลำดับที่ 5 ของอาร์เรย์ x เป็น 100” มันอาจดูแปลกที่ x[4] เป็นสมาชิกลำดับที่ 5 แทนที่จะเป็นลำดับที่ 4 แต่อาร์เรย์มีการสร้างดัชนีเริ่มจาก 0 และถูกเข้าถึงด้วยวิธีเดียวกับสตริงเราสามารถเปลี่ยน fmt.Println(x) เป็น fmt.Println(x[4]) และจะได้ผลเป็น 100

ตัวอย่างของโปรแกรมซึ่งใช้อาร์เรย์:

```
func main() {
    var x [5]float64
    x[0] = 98
    x[1] = 93
    x[2] = 77
    x[3] = 82
    x[4] = 83

    var total float64 = 0
    for i := 0; i < 5; i++ {
```


โปรแกรมคำนวณคะแนนเฉลี่ย ถ้ารันมันผลที่ได้ควรจะเป็น 86.6

ลองไล่โปรแกรม:

- สร้างอาร์เรย์ความยาว 5 หน่วยเพื่อเก็บคะแนน แล้วกำหนดค่าของสมาชิกแต่ละตัว
- เขียน for loop เพื่อคำนวณคะแนนรวม
- หาคะแนนรวมด้วยจำนวนของสมาชิกเพื่อหาค่าเฉลี่ย

ภาษาโกมีฟีเจอร์ที่ทำให้โค้ดชุดนี้ดูดีขึ้น เราลองพิจารณา `i < 5` และ `total / 5` ถ้าเปลี่ยนจำนวนของสมาชิกจาก 5 เป็น 6 เราจำเป็นต้องเปลี่ยนโค้ดสองส่วนนี้ด้วย ดังนั้นจึงควรใช้ความยาวของอาร์เรย์แทน:

```
var total float64 = 0
for i := 0; i < len(x); i++ {
    total += x[i]
}
fmt.Println(total / len(x))
```

แก้แล้วรันโปรแกรม ผลที่ได้ควรเป็น:

```
$ go run tmp.go
# command-line-arguments
.\tmp.go:19: invalid operation: total / 5 (mismatched types float64 and int)
```

ปัญหานี้มีสาเหตุมาจาก `len(x)` และ `total` มีชนิดข้อมูลแตกต่างกัน `total` เป็น `float64` ขณะที่ `len(x)` เป็น `int` ดังนั้นเราจำเป็นต้องแปลง `len(x)` ไปเป็น `float64`:

```
fmt.Println(total / float64(len(x)))
```

นี่เป็นตัวอย่างของการแปลงชนิดข้อมูล โดยปกติเราจะใช้ชื่อของชนิดข้อมูลเหมือนกับฟังก์ชันเพื่อแปลงชนิดข้อมูล

คุณสามารถใช้ for loop อีกรูปแบบหนึ่ง:

```
var total float64 = 0
for i, value := range x {
    total += value
}
fmt.Println(total / float64(len(x)))
```

for loop นี้ตัวแปร `i` แทนตำแหน่งปัจจุบันในอาร์เรย์และ `value` เปรียบเสมือน `x[i]` เราใช้คีย์เวิร์ด `range` ตามด้วยชื่อของตัวแปรที่ต้องการ loop

รันโปรแกรมนี้จะเกิดความผิดพลาดอีกอย่างหนึ่ง:

```
$ go run tmp.go
# command-line-arguments
.\tmp.go:16: i declared and not used
```

คอมไพเลอร์ภาษาโกไม่อนุญาตให้เราสร้างตัวแปรซึ่งไม่ถูกใช้ และ `i` ไม่เคยถูกใช้ใน loop ดังนั้นเราจำเป็นต้องเปลี่ยนมันเป็นดังนี้:

```
var total float64 = 0
for _, value := range x {
    total += value
}
fmt.Println(total / float64(len(x)))
```

— ถูกใช้เพื่อชื่อตัวแปรเพื่อบอกคอมไพเลอร์ว่าเราไม่ต้องการใช้ตัวแปรนั้น

ภาษาโกยังมีไวยากรณ์ที่สั้นกว่าเดิมเพื่อสร้างอาร์เรย์ด้วย:

```
x := [5]float64{ 98, 93, 77, 82, 83 }
```

เราไม่จำเป็นต้องระบุชนิดข้อมูลอีกต่อไปเพราะภาษาโกสามารถรับรู้ได้ด้วยตัวมันเอง บางครั้งอาร์เรย์อาจยาวเกินกว่าจะเขียนอยู่ใน 1 บรรทัด ดังนั้นภาษาโกจึงอนุญาตให้คุณสามารถแบ่งมันให้เป็นหลายบรรทัดดังนี้:

```
x := [5]float64{
    98,
    93,
    77,
    82,
    83,
}
```

สังเกตว่ามี `,` เกินมาหลัง `83`. สิ่งนี้จำเป็นในภาษาโกและมันทำให้เราสามารถลบสมาชิกออกจากอาร์เรย์ได้ง่ายโดยการคอมเมนต์บรรทัดนั้นออกไป:

```
x := [4]float64{
    98,
    93,
    77,
    82,
    // 83,
}
```

สไลซ์

สไลซ์คือส่วนตัดของอาร์เรย์ซึ่งมีการทำดัชนีแต่ความยาวสามารถเปลี่ยนแปลงได้ ดังตัวอย่างด้านล่าง:

```
var x []float64
```

ความแตกต่างระหว่างสไลซ์กับอาร์เรย์คือ ไม่มีการระบุความยาวในวงเล็บ และในกรณีนี้ x มีความยาวเป็น 0

อย่างไรก็ตามเราควรใช้ฟังก์ชัน make ในการสร้าง slice

```
x := make([]float64, 5)
```

ตัวอย่างด้านบนเป็นการสร้าง slice ซึ่งมีความยาวขนาด 5 หน่วย ด้วย array ของ float64. slice จะเชื่อมโยงถึงอาร์เรย์เสมอและไม่สามารถมีความยาวมากกว่าอาร์เรย์นั้น make ฟังก์ชันสามารถส่งอาร์กิวเมนต์ลำดับที่สามได้:

```
x := make([]float64, 5, 10)
```

10 คือความยาวของอาร์เรย์ที่สไลซ์อ้างถึง:



อีกวิธีหนึ่งที่จะสร้างสไลซ์คือใช้นิพจน์ [low : high]:

```
arr := [5]float64{1,2,3,4,5}
x := arr[0:5]
```

`low` คือดัชนีเริ่มต้นและ `high` ดัชนีสุดท้ายของสไลซ์ (แต่ไม่รวมดัชนีนั้น) ยกตัวอย่างเช่น `arr[0:5]` จะได้ `[1,2,3,4,5]` และ `arr[1:4]` จะได้ `[2,3,4]`

เราสามารถละเว้น `low` หรือ `high` ได้ เช่น `arr[0:]` เหมือนกับ `arr[0:len(arr)]`, `arr[:5]` เหมือนกับ `arr[0:5]` และ `arr[:]` เหมือนกับ `arr[0:len(arr)]`

ฟังก์ชันของสไลซ์

ภาษาโกมี 2 ฟังก์ชันเพื่อจัดการกับสไลซ์: `append` และ `copy` ข้างล่างเป็นตัวอย่างการใช้ `append`:

```
func main() {
    slice1 := []int{1,2,3}
    slice2 := append(slice1, 4, 5)
    fmt.Println(slice1, slice2)
}
```

เมื่อรันโปรแกรม `slice1` จะเป็น `[1,2,3]` และ `slice2` จะได้ `[1,2,3,4,5]`. `append` สร้างสไลซ์ตัวใหม่โดย `slice` ที่มีอยู่แล้ว (อาทิวเมนต์ตัวแรก) และเชื่อมต่ออาทิวเมนต์ที่เหลือกับสไลซ์นั้น

ตัวอย่างการใช้ `copy`:

```
func main() {
    slice1 := []int{1,2,3}
    slice2 := make([]int, 2)
    copy(slice2, slice1)
    fmt.Println(slice1, slice2)
}
```

หลังจากรันโปรแกรมด้านบน `slice1` จะมี `[1,2,3]` และ `slice2` จะเป็น `[1,2]` สมาชิกของ `slice1` ถูกคัดลอกไปยัง `slice2` แต่เพราะว่า `slice2` มีขนาด 2 หน่วย สมาชิกของ `slice1` จึงถูกคัดลอกไปแค่ 2 ตัว

แมป

แมปคือชุดข้อมูลแบบไม่เรียงลำดับของคู่อันดับ หรือที่เรียกว่า อาร์เรย์เชื่อมโยง ตารางแฮช หรือ พจนานุกรม `maps` ถูกใช้เพื่อค้นหาค่าโดยใช้คีย์เชื่อมโยงของมันนี่คือตัวอย่างของแมปในภาษาโก:

```
var x map[string]int
```

ชนิดข้อมูลแมปถูกแทนด้วยคีย์เวิร์ด `map` ตามด้วยชนิดของคีย์ในวงเล็บและสุดท้ายคือชนิดของค่า อ่านว่า “`x` คือแมปของสตริงของจำนวนเต็ม”

แมปสามารถถูกเข้าถึงได้โดยใช้วงเล็บเช่นเดียวกับอาร์เรย์และสไลซ์ทดลองรันโปรแกรมข้างล่าง:

```
var x map[string]int
x["key"] = 10
fmt.Println(x)
```

ผลที่ได้ควรเป็น:

```
panic: runtime error: assignment to entry in nil map

goroutine 1 [running]:
main.main()
    main.go:7 +0x4d

goroutine 2 [syscall]:
created by runtime.main
    C:/Users/ADMINI~1/AppData/Local/Temp/2/bindi
t269497170/go/src/pkg/runtime/proc.c:221
exit status 2
```

ก่อนหน้านี้เราเห็นเพียงแค่ข้อผิดพลาดจากการคอมไพล์ นี่เป็นตัวอย่างของข้อผิดพลาดจากกันรัน มันเกิดขึ้นระหว่างที่รันโปรแกรม ขณะที่ข้อผิดพลาดจากการคอมไพล์เกิดขึ้นขณะที่ทดลองคอมไพล์โปรแกรม

สาเหตุของปัญหานี้เกิดจากแมปต้องมีการกำหนดค่าเริ่มต้นก่อนที่จะพวกมันจะถูกใช้งาน เราควรเขียนในลักษณะนี้:

```
x := make(map[string]int)
x["key"] = 10
fmt.Println(x["key"])
```

ถ้าคุณรันโปรแกรมนี้ คุณจะเห็นผลลัพธ์เป็น 10 คำสั่ง `x["key"] = 10` เหมือนกับสิ่งที่เราเคยเห็นเช่นเดียวกับอาร์เรย์ยกเว้นคีย์ที่เป็นสตริงแทนที่จะเป็นจำนวนเต็ม นั่นเป็นเพราะชนิดข้อมูลของคีย์คือสตริง เราสามารถสร้าง map ด้วยคีย์ที่มีชนิดข้อมูลเป็นจำนวนได้ด้วย:

```
x := make(map[int]int)
x[1] = 10
fmt.Println(x[1])
```

สิ่งนี้ต่างกับอาร์เรย์เพียงเล็กน้อย อย่างแรกคือความยาวของแมป (เรียกฟังก์ชัน `len(x)`) สามารถ

เปลี่ยนแปลงโดยการเพิ่มสมาชิกตัวใหม่เข้าไป โดยเริ่มต้น x จะมีความยาวเป็น 0 หลังจากคำสั่ง `x[1] = 10` มันจะมีความยาวเป็น 1 อย่างที่สองคือของแมปไม่มีการเรียงลำดับ ยกตัวอย่างเช่น `x[1]` ถ้าเป็นอาร์เรย์จะหมายถึงสมาชิกตัวที่สอง แต่ในแมปไม่จำเป็นต้องเป็นแบบนั้น

เราสามารถลบสมาชิกออกจากแมปได้โดยใช้ฟังก์ชัน `delete`:

```
delete(x, 1)
```

ตัวอย่างโปรแกรมที่ใช้แมป:

```
package main

import "fmt"

func main() {
    elements := make(map[string]string)
    elements["H"] = "Hydrogen"
    elements["He"] = "Helium"
    elements["Li"] = "Lithium"
    elements["Be"] = "Beryllium"
    elements["B"] = "Boron"
    elements["C"] = "Carbon"
    elements["N"] = "Nitrogen"
    elements["O"] = "Oxygen"
    elements["F"] = "Fluorine"
    elements["Ne"] = "Neon"

    fmt.Println(elements["Li"])
}
```

`elements` คือแมปซึ่งแทนด้วยธาตุทางเคมี 10 ชนิดแรกทำดัชนีโดยสัญลักษณ์ของพวกมัน นี่เป็นวิธีการใช้แมปโดยทั่วไปลักษณะเหมือนกับพจนานุกรม สมมุติว่าเราทดลองค้นหาธาตุซึ่งไม่ปรากฏอยู่ในแมปนี้:

```
fmt.Println(elements["Un"])
```

ถ้าคุณรันโปรแกรมนี้คุณควรจะไม่เห็นผลลัพธ์ใด ๆ เลย ในทางเทคนิคแมปคืนค่า 0 สำหรับชนิดนั้น ๆ (ถ้าข้อมูลเป็นสตริงจะคืนค่าสตริงว่าง) ถึงแม้ว่าเราสามารถตรวจสอบค่า 0 ในเงื่อนไข (`elements["Un"] == ""`) โกมีวิธีที่ดีกว่านั้นโดย:

```
name, ok := elements["Un"]
fmt.Println(name, ok)
```

การเข้าถึงสมาชิกของแมปสามารถคืนค่าสองค่าแทนที่จะเป็นหนึ่ง ค่าแรกคือผลลัพธ์ของการค้นหา ส่วนค่าที่สองบ่งบอกว่าการค้นหาสำเร็จหรือไม่ ในภาษาโก บ่อยครั้งที่เราจะเห็นโค้ดลักษณะนี้:

```
if name, ok := elements["Un"]; ok {  
    fmt.Println(name, ok)  
}
```

อย่างแรกเราพยายามรับเอาค่าจากแมปตามคีย์ที่กำหนด ถ้าค่าที่เชื่อมโยงกับคีย์นั้นมีอยู่จริงจะรันโค้ดที่อยู่ภายในบล็อก

เช่นเดียวกับอาร์เรย์ ภาษาโกมีวิธีการเขียนแมปที่สั้นลง:

```
elements := map[string]string{  
    "H": "Hydrogen",  
    "He": "Helium",  
    "Li": "Lithium",  
    "Be": "Beryllium",  
    "B": "Boron",  
    "C": "Carbon",  
    "N": "Nitrogen",  
    "O": "Oxygen",  
    "F": "Fluorine",  
    "Ne": "Neon",  
}
```

บ่อยครั้งที่แมปถูกใช้เพื่อจัดกับข้อมูลทั่วไป ทดลองเปลี่ยนแปลงโปรแกรมเดิมจากการจัดเก็บแค่ชื่อธาตุเป็นเก็บสถานะพื้นฐานของมันด้วย (สถานะที่อุณหภูมิห้อง):

```
func main() {  
    elements := map[string]map[string]string{  
        "H": map[string]string{  
            "name": "Hydrogen",  
            "state": "gas",  
        },  
        "He": map[string]string{  
            "name": "Helium",  
            "state": "gas",  
        },  
        "Li": map[string]string{  
            "name": "Lithium",  
            "state": "solid",  
        },  
        "Be": map[string]string{  
            "name": "Beryllium",  
        },  
    }
```

สังเกตว่าเราเปลี่ยนจาก `map[string]string` เป็น `map[string]map[string]string` ตอนนี้เรามีแมปของสตริงไปยังแมปของสตริงไปยังสตริง แมปที่อยู่ด้านนอกถูกใช้เป็นตารางค้นหาโดยใช้สัญลักษณ์ของธาตุ ขณะที่แมปที่อยู่ด้านในถูกใช้เพื่อเก็บข้อมูลทั่วไปของธาตุนั้น ถึงแม้ว่าแมปจะถูกใช้ในลักษณะนี้อยู่บ่อย ๆ ในบทที่ 9 เราจะเห็นว่าวิธีที่ดีกว่านี้เพื่อใช้จัดเก็บข้อมูลเชิงโครงสร้าง

ปัญหา

- คุณสามารถเข้าถึงสมาชิกลำดับที่ 4 ของ array หรือ slice อย่างไร
- ความยาวของ slice ซึ่งถูกสร้างด้วย `make([]int, 3, 9)` คืออะไร
- กำหนดให้ array:

```
x := [6]string{"a","b","c","d","e","f"}
```

`x[2:5]` จะให้ผลลัพธ์อะไร

- จงเขียนโปรแกรมหาจำนวนที่น้อยที่สุดในลิสต์นี้

```
x := []int{
    48,96,86,68,
    57,82,63,70,
    37,34,83,27,
    19,97, 9,17,
}
```


ฟังก์ชัน(function)

ฟังก์ชัน(function) คือส่วนของโค้ดที่เราเขียนแยกออกมา โดยอาจจะไม่มีการรับค่าพารามิเตอร์ หรือ รับหลายๆตัวก็ได้ แล้วเมื่อฟังก์ชันทำงานเสร็จ อาจจะไม่มีการส่งผลลัพธ์ออกมา หรือ ถ้ามี โท นั้นก็สามารถมีการส่งค่ากลับออกมาได้หลายค่า เราอาจจะมองฟังก์ชันเป็น กล่องดำ ก็ได้คือ



เราไม่จำเป็นต้องรู้ว่าภายในตัวฟังก์ชันที่เราจะเรียกใช้ทำงานยังไง แค่ว่าจำเป็นต้องใส่อินพุตพารามิเตอร์อะไรเข้าไป ถึงจะได้ผลลัพธ์ตามที่ต้องการออกมา เราได้ลองสร้างฟังก์ชันกันมาก่อนแล้ว คือฟังก์ชัน main ที่เป็นส่วนของโค้ดหลักที่จะถูกเรียกใช้งานเมื่อโปรแกรมทำงาน

```
func main() {}
```

ต่อไปมาดูกันว่าเราจะเขียนฟังก์ชันขึ้นมาใช้เองอีกได้ยังไงบ้าง

สร้างฟังก์ชันใหม่ขึ้นใช้เอง

จากโค้ดนี้ในบทที่ 6

```
func main() {  
    xs := []float64{98,93,77,82,83}  
    total := 0.0  
    for _, v := range xs {  
        total += v  
    }  
    fmt.Println(total / float64(len(xs)))  
}
```

โปรแกรมนี้ทำการคำนวณค่าเฉลี่ยของตัวเลขที่อยู่ในตัวแปร xs การหาค่าเฉลี่ยแบบนี้ก็อยู่ในรูปแบบที่ถูกเอาไปใช้แก้ปัญหาอื่นๆด้วย ดังนั้นจึงควรแยกออกไปเป็นฟังก์ชันใหม่ดีกว่า เราจะสร้างฟังก์ชัน average โดยจะรับค่าข้อมูลแบบ สไลซ์ ของ float64 เข้ามา และ ให้ผลลัพธ์กลับไป 1 ค่าที่เป็นประเภทข้อมูลแบบ float64 ให้เราเพิ่มโค้ดต่อไปนี้ก่อนฟังก์ชัน main

```
func average(xs []float64) float64 {  
    panic("Not Implemented")  
}
```

จะเห็นว่าเราสร้างฟังก์ชันได้โดยใช้คีย์เวิร์ด func, ตามด้วยชื่อฟังก์ชัน ส่วนพารามิเตอร์อยู่ในวงเล็บหลังจากชื่อฟังก์ชัน ในรูปแบบของ ชื่อ ตามด้วยประเภทข้อมูล

```
name type, name type, ...
```

ซึ่งจะไม่มี หรือ มีหลายๆตัวได้ ฟังก์ชัน average นี้รับแค่ 1 พารามิเตอร์คือลิสต์ของตัวเลขที่ต้องการหาค่าเฉลี่ย โดยเราตั้งชื่อให้ว่า xs หลังจากวงเล็บของรายการพารามิเตอร์ เราจะใส่ประเภทข้อมูลที่ฟังก์ชันนี้ต้องการส่งกลับ ส่วนประกอบของฟังก์ชันที่ว่ามีได้แก่ ชื่อ , พารามิเตอร์ , ประเภทข้อมูลส่งกลับ จะเป็นตัวบ่งบอกว่าฟังก์ชันไหน หรือเรียกว่า function signature

สุดท้ายหลังจากกำหนด signature ของฟังก์ชันแล้วก็จะตามด้วยวงเล็บปีกกาที่จะเป็นกลุ่มช่องโค้ดการทำงานของฟังก์ชันนี้หรือเรียกว่า body ของฟังก์ชัน จากโค้ดที่เห็นในบอดีของฟังก์ชันนี้เรามีการเรียกฟังก์ชันชื่อ panic อยู่ ซึ่งตอนนี้จะทำให้เกิด run time error ขึ้น (เดี๋ยวเราดูเรื่อง panic กันอีกทีท้ายๆ บทนี้) การเขียนการทำงานของฟังก์ชันที่ยากๆ มักจะไม่เอามารวมไว้ในทีเดียว แต่เราจะแตกย่อยเป็นฟังก์ชันเล็กๆ แล้วเอากลุ่มฟังก์ชันย่อยที่สร้าง มาสร้างฟังก์ชันที่ซับซ้อนขึ้นมาอีกที

ทีนี้เราเอาโค้ดที่เคยเขียนใน main ย้ายมาอยู่ที่ฟังก์ชัน average ได้แบบนี้

```
func average(xs []float64) float64 {  
    total := 0.0  
    for _, v := range xs {  
        total += v  
    }  
    return total / float64(len(xs))  
}
```

เราเปลี่ยนโค้ดตรง `fmt.Println` เป็น `return` แทน เพราะเราไม่ได้ต้องการให้ฟังก์ชันแสดงอะไรออกไปที่หน้าจอ แต่เราจะใช้ `return` เพื่อส่งค่ากลับไปให้กับโค้ดจุดที่เรียกใช้ฟังก์ชันแทน การใช้ `return` จะทำให้การทำงานของฟังก์ชันหยุดลงทันที และส่งค่าที่กำหนดหลัง `return` ออกไป โค้ดใน `main` เราก็จะแก้ให้เป็นดังนี้

```
func main() {  
    xs := []float64{98,93,77,82,83}  
    fmt.Println(average(xs))  
}
```

เมื่อรัน ก็ควรจะได้ผลลัพธ์แบบเดิมกับโค้ดก่อนหน้านี้ มีสิ่งที่ควรจำไว้ก็คือ ชื่อของตัวแปรที่ส่งให้กับฟังก์ชัน ไม่จำเป็นต้องเป็นชื่อเหมือนกับพารามิเตอร์ตอนที่เราสร้างฟังก์ชัน ตัวอย่างเช่นเราสามารถเขียนแบบนี้ได้

```
func main() {  
    someOtherName := []float64{98,93,77,82,83}  
    fmt.Println(average(someOtherName))  
}
```

แล้วโปรแกรมเราก็ควรจะสามารถทำงานได้เหมือนเดิม โค้ดในตัวฟังก์ชันเอง ไม่สามารถเข้าใช้งานตัวแปรที่ถูกสร้างในฟังก์ชันต้นทางที่เรียกใช้ฟังก์ชันได้ เช่นแบบนี้ทำไม่ได้

```
func f() {  
    fmt.Println(x)  
}  
func main() {  
    x := 5  
    f()  
}
```

เราควรจะให้ฟังก์ชันส่งข้อมูลที่จำเป็นไปแทน ผ่านพารามิเตอร์ แบบนี้

```
func f(x int) {
    fmt.Println(x)
}
func main() {
    x := 5
    f(x)
}
```

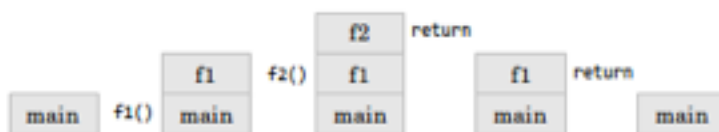
หรือเลือกที่จะประกาศตัวแปรไว้ภายนอกฟังก์ชันทั้งคู่แบบนี้

```
var x int = 5
func f() {
    fmt.Println(x)
}
func main() {
    f()
}
```

กลไกการทำงานของฟังก์ชัน เวลาฟังก์ชันหนึ่งเรียกใช้อีกฟังก์ชันหนึ่ง จะทำงานกันแบบ stack เช่นเรามีโค้ดแบบนี้

```
func main() {
    fmt.Println(f1())
}
func f1() int {
    return f2()
}
func f2() int {
    return 1
}
```

เวลาโปรแกรมทำงานจะเริ่มจาก main และมี stack ของการเรียกฟังก์ชันแสดงออกมาเป็นรูปได้แบบนี้



ทุกครั้งที่เรียกฟังก์ชันอื่นจะเกิดการเก็บข้อมูลในขอบเขตของฟังก์ชันลง stack ช้อนทับลงไปฟังก์ชันที่เป็นคนเรียก และ ก็ช้อนทับไปอีกชั้นเมื่อมีการเรียกต่อไปอีกฟังก์ชัน เมื่อมีการ return จะจบการทำงานของฟังก์ชัน ก็จะเอาข้อมูลของฟังก์ชันที่ทำงานจบแล้วออกไปจาก stack Go นั้นสามารถกำหนดชื่อให้ตัวแปรให้กับค่าที่ต้องการส่งกลับได้ เช่น

```
func f2() (r int) {  
    r = 1  
    return  
}
```

โดยเราสามารถกำหนดค่าให้ตัวแปรนี้แทนการ return ค่าตรงๆได้

การส่งค่ากลับหลายค่า

Go สามารถส่งค่ากลับได้หลายค่า ตัวอย่างเช่น

```
func f() (int, int) {  
    return 5, 6  
}  
  
func main() {  
    x, y := f()  
}
```

จากโค้ดนี้จะเห็นว่าถ้าต้องการทำให้ส่งกลับได้หลายค่า ตรงส่วนของประเภทข้อมูลของค่าที่ส่งกลับ เราจะมีวงเล็บครอบก่อนด้วย โดยในนั้นก็คือลิสต์ของประเภทข้อมูลที่จะส่งกลับ คั่นด้วย , ในส่วนของการเรียกใช้แล้วเราตัวแปรมารับค่าที่ส่งกลับแบบหลายค่า ก็ต้องกำหนดตัวแปรหลายตัวคั่นด้วย , ทางด้านซ้ายของเครื่องหมาย := หรือ = เช่นกัน

การส่งกลับหลายๆค่าใน Go มักจะถูกเอาไปใช้กับการส่งข้อมูลความผิดพลาดของการทำงานของฟังก์ชันออกมาพร้อมกับค่าผลลัพธ์ มักจะอยู่ในรูปแบบ x, err := f() หรือใช้ข้อมูลแบบ boolean ช่วยเช่น x, ok := f()

ฟังก์ชันแบบรับพารามิเตอร์ได้ โดยไม่จำกัดจำนวน (Variadic Functions)

การรับค่าแบบนี้เป็นรูปแบบพิเศษ ตัวอย่างที่เราได้ใช้กันไปแล้วก็อย่างเช่น `fmt.Println` ที่เราสามารถส่งไปกี่ค่าก็ได้ ทั้งนี้ถ้าเราจะสร้างขึ้นมาเองให้รับแบบหลายค่าแบบนี้ได้บ้าง ทำได้ตามตัวอย่างนี้

```
func add(args ...int) int {
    total := 0
    for _, v := range args {
        total += v
    }
    return total
}

func main() {
    fmt.Println(add(1,2,3))
}
```

โดยประเภทของพารามิเตอร์ เราจะใส่ ... เข้าไปหน้าประเภทข้อมูล ข้อบังคับอย่างหนึ่งคือพารามิเตอร์แบบนี้ ต้องเป็นลำดับสุดท้ายของฟังก์ชันเท่านั้น เอาไปอยู่ก่อนหน้าพารามิเตอร์อื่นไม่ได้ ตอนเรียกใช้งาน เราสามารถส่งค่าให้พารามิเตอร์นี้หลายๆค่า หรือไม่ส่งเลยก็ได้

นี่คือตัวอย่างของ signature ของฟังก์ชัน `Println` ที่รับค่าหลายค่า

```
func Println(a ...interface{}) (n int, err error)
```

โดยค่าที่รับเป็นประเภทข้อมูลแบบ `interface{}` เป็นประเภทพิเศษที่เราจะดูกันอีกทีในบทที่ 9

นอกจากการส่งค่าพารามิเตอร์ทีละค่าแล้ว เรายังสามารถใช้ข้อมูลแบบ slices ไปได้โดยเติม ... ด้านหลังตัวแปร slice ตอนเรียกใช้ฟังก์ชัน

```
func main() {
    xs := []int{1,2,3}
    fmt.Println(add(xs...))
}
```

Closure

closure คือฟังก์ชันที่สร้างขึ้นโดยไม่จำเป็นต้องมีชื่อ และสามารถกำหนดให้กับตัวแปรได้ หรือ จะส่งเป็นข้อมูลเข้าออกจากฟังก์ชันอื่นได้ ตัวอย่างเราสร้าง function ขึ้นมาภายในตัวฟังก์ชันได้ เช่น

```
func main() {
    add := func(x, y int) int {
        return x + y
    }
    fmt.Println(add(1,1))
}
```

จากโค้ด add เป็นตัวแปรโลคอล ที่เรากำหนด closure ให้ จะเห็นว่าตรงนี้เราประกาศฟังก์ชันโดยใช้ แค่ func ตามด้วยลิสต์ของพารามิเตอร์ และ ประเภทข้อมูลส่งกลับ แล้วก็ตามด้วย body ของฟังก์ชันเลย ซึ่งตัวแปร add จะมีประเภทข้อมูลเป็น func(int, int) int จะเห็นว่าเวลาเราพูดถึงประเภทข้อมูลของตัวแปรที่เก็บฟังก์ชัน จะบอกแค่ ประเภทข้อมูลของพารามิเตอร์และประเภทข้อมูลที่ส่งกลับ ไม่ได้สนใจชื่อของมัน เช่น add บอกว่าเป็นฟังก์ชันที่รับ int สองค่า และส่งกลับค่า int

จุดสำคัญของ closure หรือการประกาศฟังก์ชันภายในฟังก์ชันนี้ก็คือในขอบเขตของตัวฟังก์ชัน closure จะเรียกใช้ตัวแปรโลคอลภายในฟังก์ชันที่สร้างมันขึ้นมาได้ด้วย เช่น

```
func main() {
    x := 0
    increment := func() int {
        x++
        return x
    }
    fmt.Println(increment())
    fmt.Println(increment())
}
```

จะได้ผลลัพธ์ออกมาเป็น

1

2

ฟังก์ชัน increment ถูกสร้างขึ้นมาภายในฟังก์ชัน main โดยทำการเพิ่มค่าให้กับตัวแปร x ที่ถูกประกาศไว้ภายในฟังก์ชัน main เช่นกัน

อีกวิธีหนึ่งในการใช้ closure คือเราจะเขียนฟังก์ชัน ที่ส่งค่าออกมาได้เป็นฟังก์ชันอื่น ซึ่งเมื่อเอามาเรียกใช้ จะยังคงรักษาค่าของตัวแปรโลคอลของฟังก์ชันที่สร้างมันมา ทำให้เราเอามาใช้สร้างการ generate ลำดับของตัวเลขได้ เช่นตัวอย่างนี้เราจะสร้างฟังก์ชัน ที่ generate ตัวเลขคู่ออกมา

```
func makeEvenGenerator() func() uint {
    i := uint(0)
    return func() (ret uint) {
        ret = i
        i += 2
        return
    }
}
func main() {
    nextEven := makeEvenGenerator()
    fmt.Println(nextEven()) // 0
    fmt.Println(nextEven()) // 2
    fmt.Println(nextEven()) // 4
}
```

ในฟังก์ชัน makeEvenGenerator เองนั้นกำหนดตัวแปร i และค่าเริ่มต้นเอาไว้เป็น 0 และทำการ return closure ฟังก์ชันออกมา โดยที่ตัว closure ฟังก์ชันจะเอาตัวแปร i มาบวกเพิ่มไปที่ละ 2 แล้วส่งกลับ เมื่อเราเรียก makeEvenGenerator() แล้วกำหนดให้ตัวแปร nextEven แล้วเอา nextEven ไปเรียกใช้ ก็จะได้ค่าเลขคู่ออกมา ที่เกิดจากการจำสถานะของค่า i ของฟังก์ชัน makeEvenGenerator นั้นเอง

ฟังก์ชันเรียกตัวเอง

รูปแบบสุดท้ายในการเรียกฟังก์ชันของบนี้ก็คือ ฟังก์ชันเรียกตัวเอง ตัวอย่างของการใช้งานฟังก์ชันเรียกตัวเองเช่น ฟังก์ชันการคำนวณค่า factorial

```
func factorial(x uint) uint {  
    if x == 0 {  
        return 1  
    }  
    return x * factorial(x-1)  
}
```

จะเห็นว่าภายในฟังก์ชัน factorial จะมีการเรียก factorial ใช้งานตรงจุดที่มีการ return ซึ่งเป็นการเรียกตัวเองของฟังก์ชัน factorial เพื่อทำความเข้าใจในการทำงานของฟังก์ชันเรียกตัวเอง ลองไล่ดูการทำงานเมื่อเราเรียกใช้ factorial(2):

- ตรวจสอบว่า $x == 0$ หรือไม่? ไม่ (เพราะตอนนี้ x คือ 2)
- หาค่าของ factorial ของ $x - 1$
 - ตรวจสอบว่า $x == 0$ หรือไม่? ไม่ (เพราะตอนนี้ x คือ 1)
 - หาค่าของ factorial ของ $x - 1$
 - ตรวจสอบว่า $x == 0$ หรือไม่? ใช่ ส่งค่า 1 กลับไป
 - ส่งค่า $1 * 1$
- ส่งค่า $2 * 1$

ทั้ง Closure และฟังก์ชันเรียกตัวเองเป็นเทคนิคที่ทรงพลังมากเพราะเป็นรูปแบบที่ทำให้เราเขียนโปรแกรมในลักษณะเชิงฟังก์ชันได้ (Functional Programming) แม้ว่าคนส่วนใหญ่จะยังคงคิดว่าการเขียนโปรแกรมในลักษณะนี้จะยากในการทำความเข้าใจว่าการใช้การวนซ้ำด้วย for, การเช็คเงื่อนไขด้วย if, ตัวแปร และ การเรียกฟังก์ชันธรรมดาก็ตาม

Defer, Panic & Recover

โก มีคำสั่งพิเศษที่ชื่อว่า defer ซึ่งจะใช้กำหนดฟังก์ชันที่จะถูกเรียกใช้งาน เมื่อฟังก์ชันหลักที่ครอบ defer อยู่ทำงานเสร็จ ลองดูตัวอย่างต่อไปนี้

```

package main

import "fmt"

func first() {
    fmt.Println("1st")
}
func second() {
    fmt.Println("2nd")
}
func main() {
    defer second()
    first()
}

```

โปรแกรมนี้จะพิมพ์ 1st ตามด้วย 2nd เพราะว่าเราสั่ง defer แล้วตามด้วยการเรียก second() ทำให้ second() ถูกเรียกเมื่อฟังก์ชัน main() ทำงานเสร็จแล้วนั่นคือทำ first() เสร็จก่อน เมื่อเรียกลำดับจะเห็นว่า second() จะทำงานลำดับสุดท้ายของ main() เสมอแบบนี้

```

func main() {
    first()
    second()
}

```

defer มักจะถูกเอาไปใช้กับการคือค่าของทรัพยากรระบบเมื่อใช้งานเสร็จ ตัวอย่างเช่นเราทำการเปิดไฟล์ข้อมูลมาใช้งาน ต้องแน่ใจว่าสุดท้ายเมื่อเลิกใช้ไฟล์นั้นแล้วต้องปิดไฟล์นั้น เราจะใช้ defer มาช่วยได้ดังนี้

```

f, _ := os.Open(filename)
defer f.Close()

```

การใช้ defer ช่วยจัดการมีประโยชน์หลัก 3 อย่างดังนี้ (1) โค้ดที่สั่งปิดเราจะเห็นใกล้ๆกับตอนเปิด ทำให้ง่ายในการทำความเข้าใจ (2) ถึงแม้ว่าฟังก์ชันเราจะมีเงื่อนไขในการส่งค่ากลับหลายแบบ แต่การปิดไฟล์จะถูกเรียกเสมอ และ (3) ฟังก์ชันที่กำหนดให้ defer จะถูกเรียกเสมอแม้ว่าเกิดกรณี panic ขึ้นมาระหว่างรันโปรแกรม

Panic & Recover

บทที่แล้วเราได้เห็นกรณีเกิดข้อผิดพลาดขนาดรัน ซึ่งระบบจะเรียกฟังก์ชัน panic เพื่อส่งข้อความบอกข้อผิดพลาดออกมา เราสามารถจัดการกับ panic ที่เกิดขึ้นระหว่างโปรแกรมทำงานได้ โดยใช้ฟังก์ชัน recover ซึ่ง recover จะหยุดการ panic แล้วส่งค่ากลับไปให้กับจุดที่เกิด panic เราจะลองแก้งทำให้เกิด panic ขึ้นเพื่อดูตัวอย่างการใช้ recover ดังนี้

```
package main

import "fmt"

func main() {
    panic("PANIC")
    str := recover()
    fmt.Println(str)
}
```

แต่การเขียน recover เอาไว้หลัง panic แบบนี้จะไม่เกิดอะไรขึ้นเพราะเมื่อเรียก panic จะทำให้โปรแกรมหยุดการทำงานลง ดังนั้นเราจะเรียก recover ผ่าน defer แทนดังนี้

```
package main

import "fmt"

func main() {
    defer func() {
        str := recover()
        fmt.Println(str)
    }()
    panic("PANIC")
}
```

panic นั้นโดยทั่วไปเราจะใช้ระบุข้อผิดพลาดที่เกิดขึ้นจากโปรแกรมเมอร์ (เช่นการพยายามเข้าถึงข้อมูลของ
งาเรย์นอกขอบเขตที่กำหนด, การลืมหาค่าเริ่มต้นให้กับข้อมูลแบบ แมพ, ฯลฯ) หรือ เงื่อนไขพิเศษ
อื่นๆ ที่ยากในการที่จะ recover (เพราะแบบนี้เลยเรียกข้อผิดพลาดว่า “panic”)

ปัญหาท้าทาย

- sum คือฟังก์ชันที่รับค่าข้อมูลแบบ สไลซ์ ของตัวเลข และหาผลรวมทั้งหมดของตัวเลขที่อยู่ใน
สไลซ์ จงเขียนฟังก์ชัน signature ของ sum ที่รองรับการทำงานแบบนี้
- จงเขียนฟังก์ชันที่รับเลขจำนวนเต็ม และ ส่งค่ากลับสองค่า คือ ค่าหารสองของจำนวนนั้น และ
true ถ้าจำนวนนั้นเป็นเลขคู่ หรือ false ถ้าจำนวนนั้นเป็นเลขคี่ เช่น half(1) ให้ส่งค่า (0, false)
และ half(2) ควรจะส่งค่า (1, true) กลับไป
- จงเขียนฟังก์ชันที่รับค่าพารามิเตอร์ 1 ค่าแต่เป็นแบบไม่จำกัด (variadic parameter) เพื่อหา
ค่าที่มากที่สุดของลิสต์ตัวเลขที่ส่งมา
- จากฟังก์ชัน makeEvenGenerator ตามตัวอย่าง จงเขียนฟังก์ชัน makeOddGenerator เพื่อ
ให้สร้างลำดับของเลขคี่ออกมา
- ลำดับของฟีโบนัชชีนั้นถูกนิยามไว้ดังนี้ $fib(0) = 0$, $fib(1) = 1$, $fib(n) = fib(n-1) + fib(n-2)$ จง
เขียนฟังก์ชันแบบเรียกซ้ำเพื่อคำนวณค่าของ $fib(n)$
- อะไรคือ defer, panic และ recover และ เราจะแก้ไขข้อผิดพลาด panic ที่เกิดขึ้นใน
โปรแกรมได้อย่างไร

8 พอยน์เตอร์ (Pointers)

ทุกครั้งที่เราเรียกฟังก์ชันที่รับอาร์กิวเมนต์ อาร์กิวเมนต์เหล่านั้นจะถูกส่งเข้าไปในรูปแบบของการคัดลอกตัวตัวอย่างเช่น

```
func zero(x int) {  
    x=0  
}  
func main() {  
    x := 5  
    zero(x)  
    fmt.Println(x) // x is still 5  
}
```

โปรแกรมด้านบนเราจะเห็นว่าฟังก์ชัน zero จะไม่ทำการแก้ไขค่าเริ่มต้นของตัวแปร x จากฟังก์ชัน main แต่จะทำไมถ้าเราต้องการเปลี่ยนค่า หนึ่งในทางออกสำหรับความต้องการนี้คือการใช้ประเภทของข้อมูลชนิดพิเศษที่เรียกว่าพอยน์เตอร์(pointer)

```
func zero(xPtr *int) {  
    *xPtr = 0  
}  
func main() {  
    x := 5  
    zero(&x)  
    fmt.Println(x) // x is 0  
}
```

พอยน์เตอร์จะอ้างถึงตำแหน่งในหน่วยความจำที่ค่าถูกจัดเก็บไว้แทนที่จะเป็นการอ้างถึงค่าที่ถูกจัดเก็บไว้ (ซึ่งไปอย่างอื่น) ดังนั้นการใช้พอยน์เตอร์ (*int) จะทำให้ฟังก์ชัน zero สามารถเปลี่ยนค่าเริ่มต้นของตัวแปรได้

8.1 เครื่องหมาย * และ &

พอยน์เตอร์ในภาษาโกจะใช้เครื่องหมาย * (ดอกจัน) และตามด้วยประเภทของตัวแปรที่ถูกจัดเก็บ ดังนั้นในฟังก์ชัน zero เราเขียน *xPtr เป็นการบอกว่า xPtr ชี้ไปที่ int นอกจากนี้การใช้ * ยังใช้สำหรับ “dereference” ตัวแปรของพอยน์เตอร์ด้วย การทำ dereference หมายถึงการเข้าถึงค่า(value)ที่พอยน์เตอร์นั้นชี้(point)อยู่ยกตัวอย่างเช่นถ้าเราเขียน *xPtr = 0 เรากำลังทำ “เก็บค่า 0 ไว้ที่หน่วยความ

จำที่ xPtr อ้างอิงอยู่” แต่จะเกิดอะไรขึ้นถ้าเราเขียน xPtr = 0 สิ่งที่เราจะได้คือข้อผิดพลาดจากการคอมไพล์เพราะ xPtr ไม่ใช่ int แต่เป็น *int ดังนั้นสิ่งที่เราทำได้คือการส่งค่า *int (พอยน์เตอร์ไปที่ int) ให้มันเท่านั้นซึ่งสิ่งที่เราทำคือการส่งตำแหน่งของ x เข้าไปแทน นั่นเป็นสิ่งที่ทำให้เราสามารถแก้ไขค่าของตัวแปร &x ในฟังก์ชัน main นั่นก็เพราะ xPtr ในฟังก์ชัน zero อ้างอิงที่ตำแหน่งของหน่วยความจำตำแหน่งเดียวกัน

8.2 new

อีกทางที่เราสามารถใช้ pointer ได้คือการใช้ฟังก์ชัน new ที่มีมาให้แล้วใน Go:

```
func one(xPtr *int) {
    *xPtr = 1
}
func main() {
    xPtr := new(int)
    one(xPtr)
    fmt.Println(*xPtr) // x is 1
}
```

เราเห็นว่า new รับไทป์(type) เป็นอาร์กิวเมนต์และสิ่งที่มันทำการจอง(allocate)หน่วยความจำให้พอดีกับค่าของข้อมูลประเภทนั้นๆและจากนั้นก็ทำการส่งพอยน์เตอร์กลับออกมา มีเรื่องน่าสนใจเกี่ยวกับการใช้ new และ & ในภาษาอื่นๆเพราะในภาษาเหล่านั้นการใช้ new และ & มีความแตกต่างกันอย่างมากการใช้งานต้องทำด้วยความระมัดระวังไม่เช่นนั้นแล้วสิ่งที่เราสร้างไว้ด้วย new จะต้องถูกลบ แต่สำหรับโก สิ่งนี้จะไม่เกิดขึ้นเพราะโกเป็นภาษาที่ใช้ตัวจัดการขยะ(Gabage Collector) นั่นหมายความว่าหน่วยความจำจะถูกเก็บกวาดแบบอัตโนมัติเมื่อไม่มีการใช้งานมันอีกต่อไป แต่สำหรับGo การใช้งานพอยน์เตอร์กับ built-in type เป็นเรื่องที่เกิดขึ้นไม่บ่อยเท่าไรนักแต่อย่างไรก็ตามในบทต่อไปเราจะได้เห็นพอยน์เตอร์จะมีประโยชน์สูงมากเมื่อใช้งานคู่กับสตรัคต์

9 Structs และ Interfaces

ถ้าเราจะเขียนโปรแกรมด้วยโกโดยการใช้แค่ประเภทข้อมูลที่บิลท์อินมา(built-in)อย่างเดียวก็ไม่มีใครว่าอะไร มันเป็นไปได้ แต่ในบางครั้งการทำแบบนั้นมันก็ดูเป็นเรื่องที่ทรมานตัวเองใช้อยู่ ยกตัวอย่างเช่นถ้าเราต้องเขียนโปรแกรมที่ต้องทำงานคำนวณหาพื้นที่ของรูปร่างต่างๆ เราจะเขียนออกมาได้ประมาณนี้

```
package main
import ("fmt"; "math")
func distance(x1, y1, x2, y2 float64) float64 {
    a := x2 - x1
    b := y2 - y1
    return math.Sqrt(a*a + b*b)
}
func rectangleArea(x1, y1, x2, y2 float64) float64 {
    l := distance(x1, y1, x1, y2)
    w := distance(x1, y1, x2, y1)
    return l * w
}
func circleArea(x, y, r float64) float64 {
    return math.Pi * r*r
}
func main() {
    var rx1, ry1 float64 = 0, 0
    var rx2, ry2 float64 = 10, 10
    var cx, cy, cr float64 = 0, 0, 5
    fmt.Println(rectangleArea(rx1, ry1, rx2, ry2))
    fmt.Println(circleArea(cx, cy, cr))
}
```

การไล่ตามเก็บค่า พิกัด หลายๆตัวพร้อมๆกันเป็นเรื่องปวดกระบาลและทำให้โปรแกรมเข้าใจยากมากมาย และสิ่งนี้อาจส่งผลให้เราทำอะไรที่ผิดพลาดได้

9.1 Structs

ถ้าเราอยากให้โปรแกรมนี้น่าดูขึ้นเราสามารถนำสตรักมาใช้ได้โดยที่สตรักเองเป็นประเภทของข้อมูลที่ใช้เก็บฟิลด์ประเภทต่างไว้ได้ ยกตัวอย่างเช่นถ้าเราต้องการสร้าง struct ของ Circle เราก็สามารถทำได้ดังนี้

```

type Circle struct {
    x float64
    y float64
    r float64
}

```

คีย์เวิร์ด `type` เป็นตัวที่ใช้บอกว่าเรามี ไทป์ใหม่และจากนั้นเราจะใส่ชื่อไทป์ลงไปโดยตัวอย่างนี้เราใช้ `Circle` และแน่นอนว่าคีย์เวิร์ดสตรัคทมีไว้เพื่อบอกว่าไทป์ใหม่นี้เป็นสตรัคทโดยที่ภายในเครื่องหมายวงเล็บปีกกาจะเป็นบริเวณที่เราใส่ไว้เพื่อประกาศฟิลด์ที่อยู่ในสตรัคทนั้นๆ อย่างไรก็ตามถ้าเรามีฟิลด์ที่เป็นประเภทเดียวกันหลายๆตัวเราสามารถรวมเป็นกลุ่มไว้เป็นบรรทัดเดียวกันได้เช่นกัน

```

type Circle struct {
    x, y, r float64
}

```

กำหนดค่าเริ่มต้น (Initialization)

หลังจากประกาศ `type` แล้วต่อไปเราจะนำมาใช้งานเราสามารถกำหนดค่าเริ่มต้นได้หลายแบบมากเช่น

```

var c Circle

```

การประกาศแบบนี้จะให้ผลเหมือนการประกาศข้อมูลประเภทอื่นๆคือเราจะได้ตัวแปรโลคอลมาหนึ่งตัวที่มีค่าพื้นฐานเท่ากับศูนย์โดยสำหรับสตรัคส์แล้วการได้ค่าศูนย์แปลว่าทุกๆฟิลด์ภายใต้สตรัคส์นั้นจะมีค่าเริ่มต้นตามประเภทของข้อมูลเช่น 0 ของ `int`, 0.0 ของ `floats`, "" สำหรับ `string` และ `nil` สำหรับ `pointer` นอกจากนี้แล้วคำสั่ง `new` ก็ยังเป็นอีกทางเลือกหนึ่ง

```

c := new(Circle)

```

คำสั่ง `new` จะทำการจองหน่วยความจำสำหรับทุกๆฟิลด์และทำการกำหนดค่าศูนย์ให้กับทุกๆฟิลด์จากนั้นก็ทำการส่ง `pointer (*Circle)` กลับออกมาอย่างไรก็ตามในบางกรณีเราต้องการกำหนดค่าเริ่มต้นให้กับฟิลด์ต่างๆเราก็สามารถทำได้ดังนี้

```

c := Circle{x: 0, y: 0, r: 5}

```

หรือถ้ามันยาวไปเราก็สามารถละชื่อฟิลด์ไว้ในฐานที่เข้าใจในกรณีที่เราริเียงลำดับถูกต้อง


```
c := Circle{0, 0, 5}
```

ฟิลด์ (Fields)

เราสามารถเข้าถึงฟิลด์ต่างๆในสตรักส์ได้ด้วยการใช้เครื่องหมาย . ตามตัวอย่าง

```
fmt.Println(c.x, c.y, c.r)
c.x = 10
c.y = 5
```

ดังนั้นเราจึงสามารถปรับแต่งฟังก์ชัน circleArea เพื่อเปลี่ยนไปใช้งาน Circle ได้แบบนี้

```
func circleArea(c Circle) float64 {
    return math.Pi * c.r*c.r
}
```

และใน main เราจะได้ของหน้าตาแบบนี้

```
c := Circle{0, 0, 5}
fmt.Println(circleArea(c))
```

แต่อย่างไรก็ตามสิ่งหนึ่งที่เราต้องระลึกไว้เสมอคืออาร์กิวเมนต์จะถูกส่งผ่านโดยวิธีคัดลอกค่าเสมอในโก ดังนั้นถ้าเราต้องการเปลี่ยนค่าอะไรก็ตามในฟังก์ชัน circleArea มันจะไม่กลับไปเปลี่ยนค่าที่ต้นทาง นั่นทำให้เราต้องเปลี่ยนการส่งอาร์กิวเมนต์ใหม่ให้ส่งเป็นพอยน์เตอร์(pointer) ไปแทนในกรณีที่เราต้องการแก้ไขอะไรบางอย่างในฟังก์ชัน

```
func circleArea(c *Circle) float64 {
    return math.Pi * c.r*c.r
}
```

และใน main เราจะได้ของหน้าตาแบบนี้

```
c := Circle{0, 0, 5}
fmt.Println(circleArea(&c))
```

9.2 เมธอด(Methods)

โค้ดของเราเริ่มดีขึ้นแต่เรายังสามารถทำให้ดีกว่านี้ได้อีกด้วยการใช้ฟังก์ชันแบบพิเศษที่เรียกว่าเมธอด

```
func (c *Circle) area() float64 {  
    return math.Pi * c.r*c.r  
}
```

เราสามารถสร้างเมธอดให้สตรัคได้ด้วยการประกาศฟังก์ชันขึ้นมาแล้วเพิ่มรีซีฟเวอร์(receiver) เข้าไประหว่างคำว่า func ในที่นี้เราใช้ (c *Circle) และชื่อของฟังก์ชันดังนั้นเราจะเห็นรีซีฟเวอร์คล้ายๆกับเป็นพารามิเตอร์ - เพราะมันมีทั้งชื่อและประเภท - แต่สิ่งที่ต่างจากการประกาศฟังก์ชันทั่วไปคือการประกาศแบบนี้เราสามารถเรียกฟังก์ชันได้ด้วยการใช้เครื่องหมาย .

```
fmt.Println(c.area())
```

จะเห็นว่าโค้ดแบบนี้อ่านง่ายกว่าเดิมเยอะและเราไม่ต้องใช้เครื่องหมาย & อีกต่อไป (เพราะโกจะรู้เองโดยอัตโนมัติว่ามันจะต้องส่งพอยน์เตอร์ของ circle สำหรับเมธอดนี้) และฟังก์ชันนี้เองสามารถถูกใช้งานได้ผ่าน Circle เท่านั้นดังนั้นเราจึงควรเปลี่ยนชื่อฟังก์ชันให้เหลือแค่ area และเราก็ควรทำสิ่งนี้ที่ Rectangle เช่นกัน

```
type Rectangle struct {  
    x1, y1, x2, y2 float64  
}  
  
func (r *Rectangle) area() float64 {  
    l := distance(r.x1, r.y1, r.x1, r.y2)  
    w := distance(r.x1, r.y1, r.x2, r.y1)  
    return l * w  
}
```

โดยใน main จะเปลี่ยนเป็นแบบนี้

```
r := Rectangle{0, 0, 10, 10}
fmt.Println(r.area())
```

ไทป์แบบฝังตัว (Embedded Types)

โดยปกติแล้ว struct จะใช้สื่อความสัมพันธ์แบบ has-a ยกตัวอย่างเช่น "Circle has-a radius" ดังนั้นถ้าเรามี Person สตรัคที่มีหน้าตาแบบนี้

```
type Person struct {
    Name string
}
func (p *Person) Talk() {
    fmt.Println("Hi, my name is", p.Name)
}
```

และเราต้องการสร้าง Android struct ที่มีความสัมพันธ์กับ Person เราสามารถทำได้แบบนี้

```
type Android struct {
    Person Person
    Model string
}
```

โค้ดชุดนี้สามารถทำงานได้ดีแต่สิ่งที่เปลี่ยนไปคือวิธีอ่านเป็น “Android is a Person” แทนที่จะเป็น “Android has a Person” ดังนั้นไทป์ฝังตัวจึงเหมาะสมสำหรับการสร้างความสัมพันธ์แบบนี้ อย่างไรก็ตามการเขียนไทป์ฝังตัวยังสามารถเขียนได้อีกแบบคือ

```
type Android struct {
    Person
    Model string
}
```

เราใช้ไทป์ Person โดยที่เราไม่ได้ประกาศชื่อเลย ซึ่งการประกาศแบบนี้เราสามารถเรียก Person struct ได้ด้วยการเรียกชื่อไทป์ได้เลย

```
a := new(Android)
a.Person.Talk()
```

แต่ถ้าการเรียกผ่านไทป์ดูเว็บเราสามารถเรียกใช้เมธอดของ Person ตรงๆได้เลยเช่นกัน

```
a := new(Android)
a.Talk()
```

ดังนั้นเราจึงสามารถอ่านความสัมพันธ์แบบ is-a ได้ในลักษณะนี้ “People can talk, an android is a person, therefore an android can talk”

9.3 Interfaces

จากตัวอย่างด้านบนเราจะเห็นว่าเราสามารถตั้งชื่อเมธอด area ให้ Rectangle และมีเมธอด area สำหรับ Circle อีกเช่นกันเราจะเห็นว่ามันซ้ำกัน และรูปแบบนี้เกิดขึ้นได้เสมอในการทำงาน เราจะเห็นว่าทั้ง Rectangle และ Circle มีของที่เหมือนกันอยู่ ดังนั้นสำหรับโกเองเมื่อเกิดรูปแบบนี้ขึ้นเราสามารถใช้อ Interface เข้ามาช่วยแก้ปัญหาได้ ตัวอย่างด้านล่างเป็นตัวอย่างการสร้าง Shape Interface

```
type Shape interface {
    area() float64
}
```

เราจะเห็นว่าการสร้าง interface นั้นจะคล้ายคล้าย struct มากโดยเราจะใช้คำสั่ง type ก่อนจากนั้นตามด้วยชื่อของ interface และจบด้วยคำว่า interface แต่สิ่งที่ต่างกันระหว่างสตรัคกับอินเทอร์เฟสคือสำหรับอินเทอร์เฟสเราจะประกาศชุดของเมธอดแทนชุดของฟิลด์ ชุดของเมธอดเหล่านี้จะเป็นตัวบังคับว่าไทป์ใดๆก็ตามที่ต้องการอิมพลีเมนต์ interface นี้จะต้องเขียนรายละเอียดให้เมธอดเหล่านั้น

สำหรับกรณีของเราจะเห็นได้ว่าทั้ง Rectangle และ Circle มีเมธอด area ที่คืนค่า float64s ดังนั้นเราจึงสามารถบอกได้ว่าทั้งคู่เป็น implementation ของ Shape

ดังนั้นเราจึงสามารถใช้ความเหมือนของทั้งสองสตรัคให้เป็นประโยชน์ได้ด้วยการ ส่งอินเทอร์เฟสเข้าไปเป็นอาร์กิวเมนต์ของฟังก์ชันดังตัวอย่างได้

```
func totalArea(shapes ...Shape) float64 {
    var area float64
    for _, s := range shapes {
        area += s.area()
    }
}
```

```
    return area
}
```

และเราสามารถเรียกใช้เมธอดได้แบบนี้

```
fmt.Println(totalArea(&c, &r))
```

และ interface เองยังสามารถถูกใช้เป็นพารามิเตอร์ได้

```
type MultiShape struct {
    shapes []Shape
}
```

นอกจากนี้เรายังสามารถเปลี่ยน MultiShape ให้เป็น Shape ได้เพื่อดึงเอา area ออกมาด้วยการเรียกเมธอดได้ดังนี้

```
func (m *MultiShape) area() float64 {
    var area float64
    for _, s := range m.shapes {
        area += s.area()
    }
    return area
}
```

ดังนั้นตอนนี้เราจะเห็นว่า MultiShape สามารถบรรจุได้ทั้ง Circle, Rectangle หรือแม้กระทั่ง MultiShape

ภาวะพร้อมกัน (Concurrency)

โปรแกรมขนาดใหญ่มักจะประกอบด้วยโปรแกรมขนาดเล็กหลายตัว อย่างเช่น เว็บเซิร์ฟเวอร์ที่ต้องจัดการรีเควสต์ จากเบราว์เซอร์และส่ง HTML กลับไป ตอนที่จัดการกับรีเควสต์ แต่ละตัวมันก็เหมือนกับโปรแกรมเล็กๆ ตัวนึง

ในทางอุดมคติโปรแกรมแบบนี้จะสามารถรันแต่ละส่วนพร้อมกันได้ (กรณีที่เว็บเซิร์ฟเวอร์จัดการรีเควสต์หลายตัว) การทำให้งานแต่ละงานสามารถทำในเวลาเดียวกันได้นี้แหละเราเรียกว่า concurrency ซึ่ง Go มันโคตรจะสนับสนุนการทำ concurrency โดยใช้สิ่งที่เรียกว่า goroutines และ channels

โกรูทีน (Goroutines)

โกรูทีนเป็นฟังก์ชันที่สามารถทำงานได้ในเวลาเดียวกันกับฟังก์ชันอื่นได้ วิธีสร้างโกรูทีนก็แค่ใส่คำว่า `go` นำหน้าการเรียกใช้งานฟังก์ชัน

```
package main

import "fmt"

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
    }
}

func main() {
    go f(0)
    var input string
    fmt.Scanln(&input)
}
```

โปรแกรมนี้นี้มีโกรูทีนสองตัว ตัวแรกมันซ่อนอยู่ ซึ่งก็คือฟังก์ชัน `main` นั่นเอง ส่วนตัวที่สองมันถูกสร้างตอนที่เราเรียก `go f(0)` ปกติแล้วเวลาเราเรียกฟังก์ชันโปรแกรมมันจะทำงานและคืนค่าผลลัพธ์ของฟังก์ชันก่อนทำคำสั่งในบรรทัดถัดไป แต่โกรูทีนจะคืนค่ากลับมาทันทีและทำบรรทัดถัดไปโดยไม่ต้องรอให้ฟังก์ชันทำงานจบ เป็นที่มั่วว่าทำไมต้องเรียก `Scanln` เพราะถ้าไม่เรียกโปรแกรมก็จะจบการทำงานไปเลยโดยไม่มีโอกาสที่จะได้พิมพ์ค่าตัวเลขออกทางหน้าจอ

โกรูทีนมันเบา (lightweight) และเราสามารถสร้างมันขึ้นมาเป็นพันๆ ได้โดยง่าย เราสามารถแก้โปรแกรมให้รัน 10 โกรูทีน ได้ตามนี้

```
func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

เราอาจจะเห็นเหมือนว่าโปรแกรมมันรันโกรูทีนตามลำดับแทนที่จะรันพร้อมกัน ทีนี้เราลองมาหน่วงเวลามันซักหน่อยด้วยคำสั่ง `time.Sleep` และ `rand.Intn`

```
package main

import (
    "fmt"
    "time"
    "math/rand"
)

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
        amt := time.Duration(rand.Intn(250))
        time.Sleep(time.Millisecond * amt)
    }
}

func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

f พิมพ์ตัวเลขจาก 0 ถึง 10, แล้วก็รอ 0 - 250 มิลลิวินาทีในแต่ละรอบ นั่นแน่! เห็นแล้วใช่มั้ยว่าโกรูทีน มันรันพร้อมกัน

แชนแนล (Channels)

แชนแนลจัดวิธีที่จะทำให้โกรทีน คุยกันและทำงานประสานจังหวะ (synchronize) กันได้ มาดูตัวอย่างการใช้แชนแนลกันเลย

```
package main

import (
    "fmt"
    "time"
)

func pinger(c chan string) {
    for i := 0; ; i++ {
        c <- "ping"
    }
}

func printer(c chan string) {
    for {
        msg := <- c
        fmt.Println(msg)
        time.Sleep(time.Second * 1)
    }
}

func main() {
    var c chan string = make(chan string)

    go pinger(c)
    go printer(c)

    var input string
    fmt.Scanln(&input)
}
```

จากตัวอย่างนี้จะพิมพ์ “ping” ออกมาเรื่อยๆ (กด enter เพื่อให้จบการทำงาน) แชนแนลประกาศโดยใช้คำว่า `chan` ตามด้วยประเภทของสิ่งที่ส่งเข้าไปในแชนแนล (ในตัวอย่างเราส่ง string) เครื่องหมาย `<-` (ลูกศรชี้ไปทางซ้าย) ใช้เพื่อส่งและรับข้อความบนแชนแนล `c <- “ping”` หมายถึงส่ง “ping” เข้าไปในแชนแนล ส่วนการรับข้อความจะเขียนในรูป `msg := <- c` ซึ่งหมายถึง รับข้อความจากแชนแนล และเก็บข้อความไว้ใน `msg` ที่จริงการพิมพ์ข้อความออกหน้าจอจะลบบรรทัดก่อนหน้าออกแล้วเขียนแค่ `fmt.Println(<-c)` ก็ได้นะ

การใช้เซนแนลแบบนี้จะประสานจังหวะของโกรูทีนสองตัว เมื่อ `pinger` พยายามที่จะส่งข้อความบนเซนแนลมันก็จะรอจนกว่า `printer` พร้อมที่จะรับข้อความ (เราเรียกอาการแบบนี้ว่า `blocking`) มาลองเพิ่มตัวส่งข้อความอีกตัวเข้าไปในโปรแกรมแล้วดูว่าจะเกิดอะไรขึ้น เราเพิ่มฟังก์ชันเข้าไป

```
func ponger(c chan string) {  
    for i := 0; ; i++ {  
        c <- "pong"  
    }  
}
```

แล้วแก้ฟังก์ชัน `main` อีกซักหน่อย

```
func main() {  
    var c chan string = make(chan string)  
  
    go pinger(c)  
    go ponger(c)  
    go printer(c)  
  
    var input string  
    fmt.Scanln(&input)  
}
```

ตอนนี้โปรแกรมก็จะสลับกับพิมพ์ข้อความ “ping” และ “pong”

ทิศทางของเซนแนล (Channel Direction)

เราสามารถระบุได้ว่าเซนแนลนี้จะให้ทำเฉพาะรับหรือส่ง เช่นฟังก์ชัน `pinger` เราสามารถเขียนเป็น

```
func pinger(c chan<- string)
```

ตอนนี้เราจะทำได้แค่ส่งข้อความเข้าไปใน `c` เท่านั้น ถ้าเราพยายามที่จะรับข้อมูลจาก `c` มันจะเกิดความผิดพลาดขึ้นตอนที่เราคอมไพล์ เราสามารถเปลี่ยน `printer` ได้โดยใช้วิธีคล้ายๆกัน

```
func printer(c <-chan string)
```

เซนแนลที่ไม่ได้ระบุทิศทางจะสามารถใช้งานได้ทั้งสองทิศทาง ซึ่งเซนแนลแบบสองทิศทางสามารถส่งเข้าฟังก์ชันที่รับเซนแนลแบบส่งอย่างเดียว หรือ รับอย่างเดียวก็ได้

Select

Go มี statement พิเศษชื่อ **select** ซึ่งทำงานคล้ายกับ **switch** แต่ใช้กับแชนแนล

```
func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        for {
            c1 <- "from 1"
            time.Sleep(time.Second * 2)
        }
    }()
    go func() {
        for {
            c2 <- "from 2"
            time.Sleep(time.Second * 3)
        }
    }()
    go func() {
        for {
            select {
                case msg1 := <- c1:
                    fmt.Println(msg1)
                case msg2 := <- c2:
                    fmt.Println(msg2)
            }
        }
    }()

    var input string
    fmt.Scanln(&input)
}
```

โปรแกรมนี้จะพิมพ์ “from 1” ทุกสองวินาที และ “from 2” ทุกสามวินาที **select** จะเลือกแชนแนลแรกที่มีพร้อมจะรับจากมัน (หรือส่งมาให้มัน) ถ้ามีมากกว่าหนึ่งแชนแนลที่พร้อม มันก็จะสุ่มเอาอันไหนก็ได้ ถ้าไม่มีแชนแนลไหนพร้อมเลย มันก็จะรอจนกว่าจะมีแชนแนลพร้อมขึ้นมาซักตัว

select ถูกใช้บ่อยในการทำ timeout

```
select {
case msg1 := <- c1:
    fmt.Println("Message 1", msg1)
case msg2 := <- c2:
    fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
    fmt.Println("timeout")
}
```

`time.After` จะสร้าง채널ขึ้นมาและส่งไปตอนที่ครบกำหนดเวลา (เราไม่สนใจที่จะเก็บเวลาก็เลยไม่ต้องประกาศตัวแปรมาเก็บค่ามัน) นอกจากนี้เรายังสามารถระบุกรณี `default` ได้ด้วย

```
select {
case msg1 := <- c1:
    fmt.Println("Message 1", msg1)
case msg2 := <- c2:
    fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
    fmt.Println("timeout")
default:
    fmt.Println("nothing ready")
}
```

กรณี `default` นี้จะเกิดขึ้นที่ที่ไม่มี채널ใดพร้อมเลย

Buffered Channels

เราสามารถส่งพารามิเตอร์ตัวที่สองเข้าไปในฟังก์ชัน `make` ตอนที่สร้าง채널ได้ ดังนี้

```
c := make(chan int, 1)
```

เราจะได้ buffered channel ที่มีความจุเป็น 1 โดยปกติแล้ว채널จะเป็น synchronous แต่ละฝั่งของ채널จะรอจนกว่าอีกฝั่งจะพร้อม ซึ่ง buffered channel จะต่างออกไป โดย buffered channel จะเป็น asynchronous แต่ละฝั่งจะไม่รอกันจนกว่า채널จะเต็ม

ปัญหาท้าทาย

- ลองเขียน채널แบบระบุทิศทางดูหน่อย มันเขียนยังไงแล้วนะ?
- เขียนฟังก์ชัน `Sleep` ขึ้นมาเองเลย โดยใช้ `time.After` นะ
- buffered channel มันคืออะไรอะ อธิบายหน่อย แล้วถ้าเราจะสร้างมันให้มีความจุซัก 20 เราจะทำได้ยังไง?

Packages

ภาษา Go นั้นถูกออกแบบมาเพื่อสนับสนุนแนวปฏิบัติที่ดี สำหรับการพัฒนาซอฟต์แวร์ โดยส่วนที่สำคัญของซอฟต์แวร์ที่มีคุณภาพสูงก็คือ การนำ code กลับมาใช้ซ้ำ (reuse) ซึ่งเป็นไปตามแนวคิด Don't Repeat Yourself (DRY)

ในบทที่ 7 เรื่อง Functions นั้นเป็นส่วนแรกของการ reuse เท่านั้น โดยภาษา Go ยังได้เตรียมกลไกหรือวิธีการอื่นๆมาให้ด้วย คือ package ถ้าสังเกตจากโค้ดที่ผ่านมา จะพบว่ามี

```
import "fmt"
```

fmt คือชื่อ package ประกอบไปด้วยฟังก์ชันการทำงานที่เกี่ยวกับการจัดรูปแบบข้อมูล การแสดงผลลัพธ์กลับไปยังหน้าจอ การรวบรวมไว้ในที่เดียวกันนี้มีจุดประสงค์ 3 อย่าง ดังนี้

- ช่วยลดความซ้ำซ้อนของชื่อ ทำให้ชื่อของฟังก์ชันสั้นกระชับ
- ช่วยให้ง่ายต่อการค้นหาโค้ดที่ต้องการใช้ซ้ำ
- ช่วยทำให้ compiler ทำงานที่รวดเร็วขึ้น เนื่องจากจะ compile ในเฉพาะส่วน หรือใน package เท่านั้น

การสร้าง Packages

Package นั้นเป็นแนวคิดพื้นฐานสำหรับการแบ่งกลุ่มของโปรแกรม ดังนั้นเรามาดูกันว่าสามารถสร้างมันได้อย่างไร เริ่มต้นสร้างโฟลเดอร์ ชื่อ chapter11 ใน ~/Go/src/golang-book แล้วสร้างไฟล์ชื่อ main.go ซึ่งมีโค้ดดังนี้

```
package main

import "fmt"
import "golang-book/chapter11/math"

func main() {
    xs := []float64{1,2,3,4}
    avg := math.Average(xs)
    fmt.Println(avg)
}
```

ต่อมาทำการสร้างโฟลเดอร์ ชื่อ math ใน chapter11 โดยภายในโฟลเดอร์ math สร้างไฟล์ชื่อ math.go มีโค้ดดังนี้

```
package math
```

```
func Average(xs []float64) float64 {
    total := float64(0)
    for _, x := range xs {
        total += x
    }
    return total / float64(len(xs))
}
```

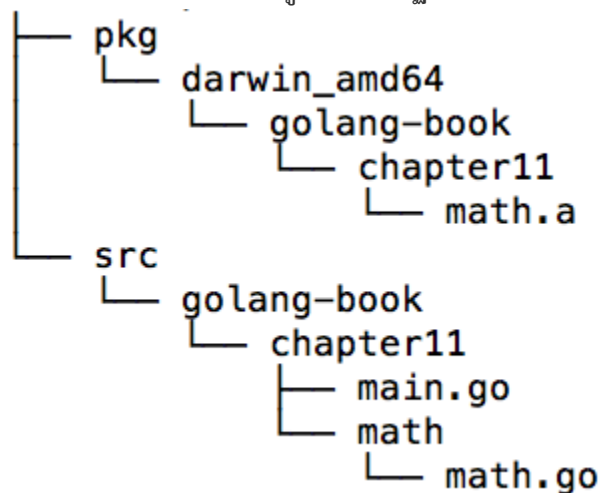
ไปยัง terminal หรือ command line แล้วเข้าไปที่โฟลเดอร์ math พิมพ์คำสั่ง go install

```
$cd math
```

```
$go install
```

Go จะทำการ compile ไฟล์ math.go และสร้างไฟล์ linkable object ดังนี้ `~/Go/pkg/os_arch/golang-book/chapter11/math.a`

โดยชื่อ `<os_arch>` ขึ้นอยู่กับระบบปฏิบัติการที่ใช้งาน ตัวอย่างเช่น `darwin_amd64` ดังรูป



หมายเหตุ

ถ้า run แล้วมีปัญหาหรือแสดงข้อความลักษณะนี้

go install: no install location for directory ... outside GOPATH

ให้ทำการแก้ไขด้วยการกำหนดตัวแปรระบบชื่อว่า `GOPATH` ไปดังนี้

```
$export GOPATH=~/Go/
```

ต่อมาให้กลับไปยังโฟลเดอร์ chapter11 แล้วพิมพ์คำสั่ง go run main.go ดังนี้

```
$cd ..
```

```
$go run main.go
```

จะแสดงค่า 2.5 ขึ้นมา
เรามาดูส่วนที่น่าสนใจกัน

- `math` คือชื่อ `package` ซึ่งถ้าไปดูในเอกสารจะพบว่ามันคือชื่อ `library` พื้นฐานของ Go แต่โครงสร้างของ `package` ทำให้เราสามารถใช้งาน `math` ได้ เนื่องจาก `package math` ของ Go นั้นคือ `math` แต่ `math` ของเราหรือจากตัวอย่างอยู่ที่ `golang-book/chapter11/math` ซึ่งจะเห็นจากภาพด้านบน
- เมื่อต้องการ `import` ใช้งาน `math` ของเราต้องใช้ชื่อเต็มดังนี้ `import "golang-book/chapter11/math"`
- เราสามารถใช้งานผ่านชื่อสั้นๆ เพื่อเรียกใช้งานฟังก์ชันจาก `library` ต่างๆ ได้ ด้วย Go อนุญาตให้ใช้การ `alias` ดังโค้ดด้านล่าง

```
import m "golang-book/chapter11/math"
```

```
func main() {  
    xs := []float64{1,2,3,4}  
    avg := m.Average(xs)  
    fmt.Println(avg)  
}
```

`m` คือชื่อที่ต้องการใช้งาน

- คุณอาจตั้งข้อสังเกตว่า ในทุกๆ ฟังก์ชันใน `package` จะขึ้นต้นด้วยตัวอักษรพิมพ์ใหญ่เสมอ ซึ่งตรงนั้นมันมีความหมาย กล่าวคือ เมื่อใดก็ตามที่ชื่อฟังก์ชันขึ้นต้นด้วยตัวอักษรพิมพ์ใหญ่แล้ว `package` อื่นๆ สามารถเห็นและใช้งานฟังก์ชันนั้นได้
- การจัดการเรื่องการใช้งานของ `package` นั้นถือว่าเป็นแนวปฏิบัติที่ดี ที่จะให้ `package` อื่นใช้งานได้หรือซ่อนก็ได้ ทำให้เราสะดวกต่อการใช้งานและการแก้ไขส่วนต่างๆ ใน `package` โดยไม่กระทบต่อส่วนอื่นๆ ด้วย
- เพื่อให้ง่ายต่อชีวิต แนะนำให้ชื่อ `package` นั้นควรตรงกับชื่อไฟล์เดอร์

เอกสาร

Go นั้นมีความสามารถในการสร้างเอกสารสำหรับ package แบบอัตโนมัติอยู่แล้ว เพียงแค่ใช้คำสั่ง

```
$godoc golang-book/chapter11/math Average
```

ผลการทำงานเป็นดังรูป

PACKAGE DOCUMENTATION

```
package math
import "golang-book/chapter11/math"
```

FUNCTIONS

```
func Average(xs []float64) float64
```

ซึ่งดูแล้วไม่สวย เนื่องจากเรายังไม่ใส่ข้อความ comment อะไรลงไป ใน code ดังนั้นลองเพิ่ม comment ไปใน code หน่อยสิ ดังนี้

```
// Finds the average of a series of numbers
func Average(xs []float64) float64 {
```

แล้วลอง run godoc ใหม่จะได้ผลดังรูป ซึ่งดูดีขึ้นมาน้อย

PACKAGE DOCUMENTATION

```
package math
import "golang-book/chapter11/math"
```

FUNCTIONS

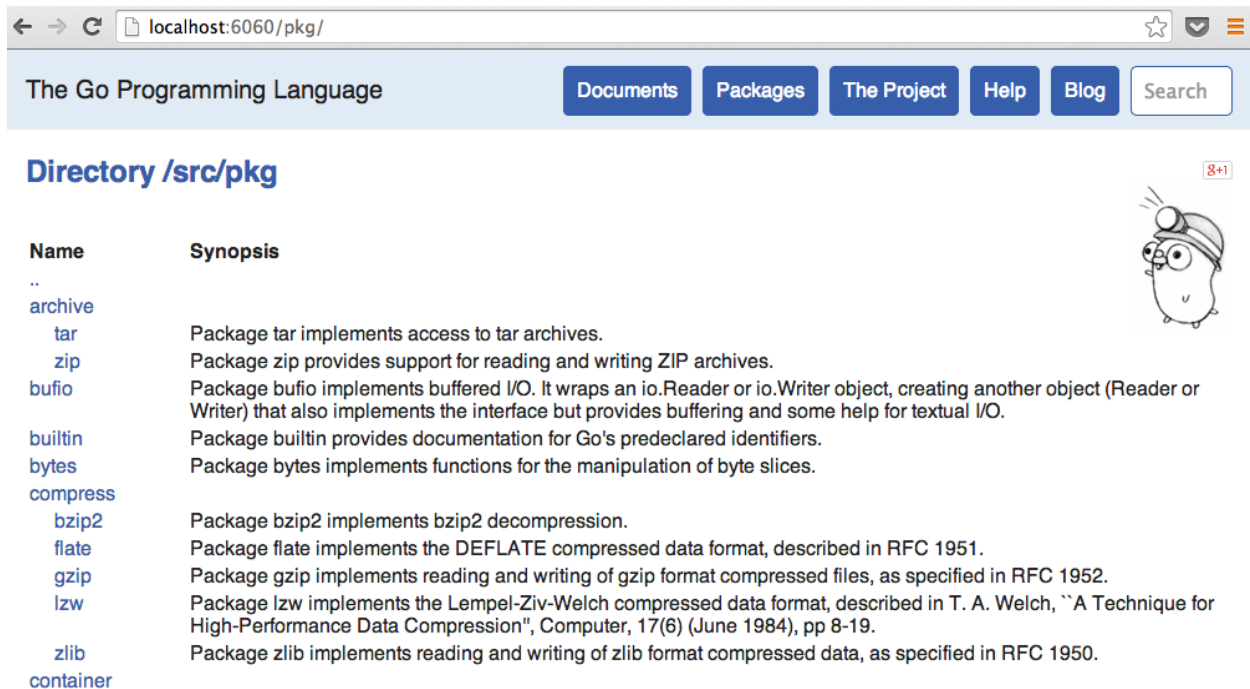
```
func Average(xs []float64) float64
    Finds the average of a series of numbers
```

เมื่อทำการแก้ไขไฟล์ math.go แล้วอย่าลืมพิมพ์คำสั่ง go install ก่อนที่จะสั่ง godoc นะ
ต่อไปทำการแสดงเอกสารผ่านเว็บดีกว่า เพราะว่ามันจะมีประโยชน์มากขึ้น โดยใช้คำสั่งดังนี้
`$godoc -http=":6060"`

แล้วเข้า browser ไปยัง url นี้

<http://localhost:6060/pkg/>

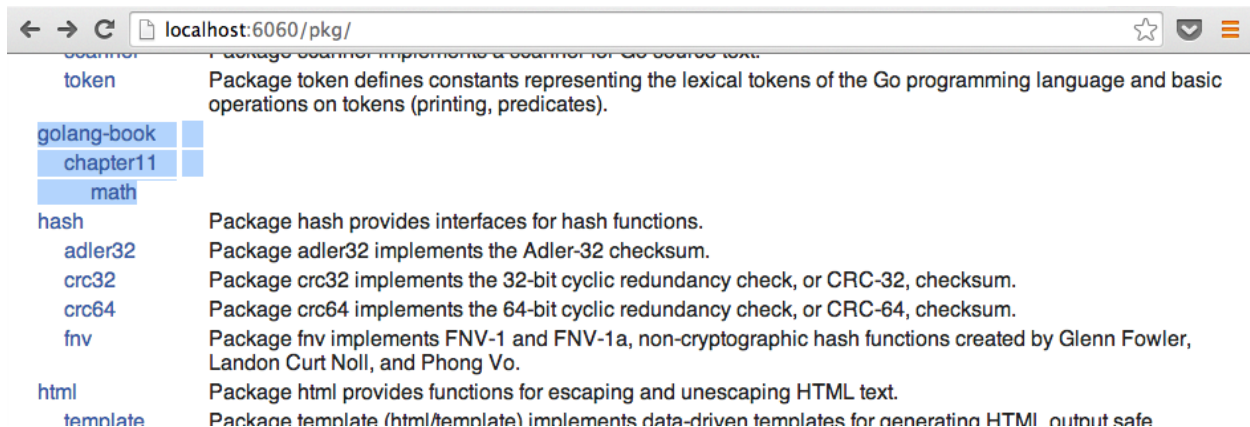
ผลการทำงานเป็นดังนี้



The screenshot shows the Go Programming Language website at localhost:6060/pkg/. The page title is "The Go Programming Language". There are navigation buttons for "Documents", "Packages", "The Project", "Help", "Blog", and a "Search" button. The main heading is "Directory /src/pkg". On the right, there is a cartoon character wearing a hard hat and a small "8+1" badge. The main content is a table with two columns: "Name" and "Synopsis".

Name	Synopsis
..	
archive	
tar	Package tar implements access to tar archives.
zip	Package zip provides support for reading and writing ZIP archives.
bufio	Package bufio implements buffered I/O. It wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.
builtin	Package builtin provides documentation for Go's predeclared identifiers.
bytes	Package bytes implements functions for the manipulation of byte slices.
compress	
bzip2	Package bzip2 implements bzip2 decompression.
flate	Package flate implements the DEFLATE compressed data format, described in RFC 1951.
gzip	Package gzip implements reading and writing of gzip format compressed files, as specified in RFC 1952.
lzv	Package lzv implements the Lempel-Ziv-Welch compressed data format, described in T. A. Welch, "A Technique for High-Performance Data Compression", Computer, 17(6) (June 1984), pp 8-19.
zlib	Package zlib implements reading and writing of zlib format compressed data, as specified in RFC 1950.
container	

เมื่อเลื่อนลงมาจะเห็น package /golang-book/chapter11/math ดังรูป



The screenshot shows the Go Programming Language website at localhost:6060/pkg/. The page title is "The Go Programming Language". There are navigation buttons for "Documents", "Packages", "The Project", "Help", "Blog", and a "Search" button. The main heading is "Directory /src/pkg". On the right, there is a cartoon character wearing a hard hat and a small "8+1" badge. The main content is a table with two columns: "Name" and "Synopsis".

Name	Synopsis
..	
archive	
tar	Package tar implements access to tar archives.
zip	Package zip provides support for reading and writing ZIP archives.
bufio	Package bufio implements buffered I/O. It wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.
builtin	Package builtin provides documentation for Go's predeclared identifiers.
bytes	Package bytes implements functions for the manipulation of byte slices.
compress	
bzip2	Package bzip2 implements bzip2 decompression.
flate	Package flate implements the DEFLATE compressed data format, described in RFC 1951.
gzip	Package gzip implements reading and writing of gzip format compressed files, as specified in RFC 1952.
lzv	Package lzv implements the Lempel-Ziv-Welch compressed data format, described in T. A. Welch, "A Technique for High-Performance Data Compression", Computer, 17(6) (June 1984), pp 8-19.
zlib	Package zlib implements reading and writing of zlib format compressed data, as specified in RFC 1950.
container	
golang-book	
chapter11	
math	
hash	Package hash provides interfaces for hash functions.
adler32	Package adler32 implements the Adler-32 checksum.
crc32	Package crc32 implements the 32-bit cyclic redundancy check, or CRC-32, checksum.
crc64	Package crc64 implements the 64-bit cyclic redundancy check, or CRC-64, checksum.
fnv	Package fnv implements FNV-1 and FNV-1a, non-cryptographic hash functions created by Glenn Fowler, Landon Curt Noll, and Phong Vo.
html	Package html provides functions for escaping and unescaping HTML text.
template	Package template (html/template) implements data-driven templates for generating HTML output safe

เข้าไปดูใน package math มีข้อมูลดังรูป

Package math

8+1

```
import "golang-book/chapter11/math"
```

Overview
Index

Overview ▾

Index ▾

```
func Average(xs []float64) float64
```

Click to hide Index section

Package files

math.go

func Average

```
func Average(xs []float64) float64
```

Finds the average of a series of numbers

Build version go1.2.1.

Except as noted, the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

ปัญหา

- ทำไมเราต้องใช้ package ละ
- ความแตกต่างระหว่าง Average กับ average คืออะไร
- package alias คืออะไร และใช้มันอย่างไร
- ให้ทำการคัดลอกฟังก์ชัน average จากบทที่ 7 และทำการสร้างฟังก์ชัน Min และ Max ซึ่งใช้สำหรับหาค่าตัวเลขน้อยที่สุดและมากที่สุดจาก Slice มีชนิดเป็น float64
- สร้างเอกสารของ 3 ฟังก์ชันอย่างไร
-

การทดสอบ (Testing)

ใครว่า Programming มันง่าย จริงๆ แล้วไม่เลย ยิ่งถ้าเจอกับ Programmer ที่ชั่วโมงบินน้อย บาดแผลบนร่างกายยังน้อย หรือยังเก่าไม่พอ ความมั่นใจในเรื่องของคุณภาพก็น้อยตามไปด้วย ดังนั้นการทดสอบ (Testing) จึงเป็นส่วนสำคัญมากในกระบวนการพัฒนาซอฟต์แวร์ ดังนั้นการเขียน Code Test จึงเป็นเรื่องที่สำคัญที่ช่วยให้เรามั่นใจว่าคุณภาพและสุขภาพของ Code ของเรายังดีอยู่

Go ได้เตรียมวิธีการเขียน Test แบบใช้งานได้ง่ายๆ ไว้ให้แล้ว ลองมาเริ่มเขียน Test กันจาก Code ที่เราสร้างไว้จากบทที่ 11 ใน folder math ให้สร้างไฟล์ใหม่ชื่อว่า math_test.go ตามตัวอย่าง Code ด้านล่างนี้

```
package math

import "testing"

func TestAverage(t *testing.T) {
    var v float64
    v = Average([]float64{1,2})
    if v != 1.5 {
        t.Error("Expected 1.5, got ", v)
    }
}
```

แล้วพิมพ์คำสั่งเพื่อทำการ Run

```
go test
```

จะได้ผลลัพธ์ออกมาตามนี้

```
$ go test
PASS
ok   golang-book/chapter11/math    0.032s
```

คำสั่ง go test ใช้สำหรับสั่ง Run Test ที่อยู่ใน Folder นั้นๆ เมื่อจะเขียน Test คราใด ให้เริ่มต้นชื่อ function ด้วยคำว่า Test ครานั้น และใช้ *testing.T เป็น argument ดังนั้นจากโจทย์ของเรา เราต้องเขียน Test ขึ้นมาเพื่อทดสอบ function Average ดังนั้นชื่อ function Test เราจึงเป็น TestAverage

เมื่อเรามี Test แรกที่ใช้ในการทดสอบการหาค่าเฉลี่ยของ [1,2] ที่จะต้องได้ค่าเฉลี่ยออกมาเป็น 1.5 แล้วนั้น ลองเพิ่มความสนุกในการทดสอบลงไปด้วย Test Cases กรณีต่างๆ เพิ่มเข้าไปเพื่อให้การทดสอบนั้นครอบคลุมมากขึ้น ตามตัวอย่างด้านล่าง

```
package math

import "testing"

type testpair struct {
    values []float64
    average float64
}

var tests = []testpair{
    { []float64{1,2}, 1.5 },
    { []float64{1,1,1,1,1,1}, 1 },
    { []float64{-1,1}, 0 },
}

func TestAverage(t *testing.T) {
    for _, pair := range tests {
        v := Average(pair.values)
        if v != pair.average {
            t.Error(
                "For", pair.values,
                "expected", pair.average,
                "got", v,
            )
        }
    }
}
```

เราสร้าง struct มาเพื่อกำหนดค่า input และ output สำหรับการเรียกใช้งานใน function แล้วเราก็สร้างชุดของข้อมูลไว้ใน struct (pairs) เมื่อเราสั่งคำสั่ง go test คราใด ค่า input และ output แต่ละคู่ใน struct จะถูกส่งเข้าไปใน function TestAverage เพื่อทำการทดสอบ

ปัญหาท้าทาย

- การเขียนชุด Test ให้ดีไม่ใช่เรื่องง่าย แต่ก็ไม่ยากที่จะทำโดยการ ต้องฝึกฝน และเขียนบ่อยๆ จะเกิดการพัฒนาทักษะและเข้าใจมากยิ่งขึ้นเรื่อยๆ ยกตัวอย่างเช่น จะเกิดอะไรขึ้นถ้าค่าของ

[]float64{ } เป็นค่าว่าง (empty list)? เราจะแก้ไข function อย่างไรให้ return 0 กลับมาให้
ในกรณีนี้?

- เขียน Test เพื่อทดสอบ function Min และ Max ที่สร้างไว้จากบทที่แล้ว

แพ็คเกจหลัก (The Core Packages)

ปกติการเขียนทุกอย่างจากจุดเริ่มต้น ในโลกแห่งความเป็นจริงเราจะต้องพึ่งความสามารถของไลบรารีที่มีอยู่แล้ว ในบทนี้จะไปดูแพ็คเกจที่นิยมใช้บ่อยๆ ที่มีอยู่ในโกกัน

คำเตือน: ไลบรารีหลายๆ ตัวจะต้องมีความรู้ในเรื่องนั้นๆ เป็นพิเศษ เช่น วิทยาการเข้ารหัส (cryptography) ซึ่งมันเกินขอบเขตของหนังสือเล่มนี้ไปในการที่อธิบายเทคโนโลยีนั้น

สตริง (Strings)

โกได้รวมฟังก์ชันที่เอาไว้ใช้จัดการกับสตริงอยู่ในแพ็คเกจที่ชื่อว่า strings

```

package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(
        // true
        strings.Contains("test", "es"),

        // 2
        strings.Count("test", "t"),

        // true
        strings.HasPrefix("test", "te"),

        // true
        strings.HasSuffix("test", "st"),

        // 1
        strings.Index("test", "e"),

        // "a-b"
        strings.Join([]string{"a", "b"}, "-"),

        // == "aaaaa"
        strings.Repeat("a", 5),

        // "bbaa"
        strings.Replace("aaaa", "a", "b", 2),

        // []string{"a","b","c","d","e"}
        strings.Split("a-b-c-d-e", "-"),

        // "test"
        strings.ToLower("TEST"),

        // "TEST"
        strings.ToUpper("test"),
    )
}

```

บางครั้งเราต้องการทำงานกับสตริงด้วยข้อมูลแบบไบนารี(binary) การที่จะแปลงสตริงไปเป็น []byte สไลซ์ของไบต์(และจาก []byte สไลซ์ของไบต์ไปเป็นสตริง) ทำได้โดย


```
arr := []byte("test")
str := string([]byte{'t','e','s','t'})
```

อินพุต/เอาต์พุต (Input / Output)

ก่อนที่เราจะได้ดูเรื่องไฟล์เราต้องเข้าใจแพ็คเกจไอโอ(io) ของโกเซก่อน แพ็คเกจ io มีฟังก์ชันให้ใช้ไม่เยอะ แต่ส่วนใหญ่แพ็คเกจอื่นจะมาใช้อินเทอร์เฟซซะมากกว่า โดยอินเทอร์เฟซหลักๆ จะมีสองตัวคือ Reader กับ Writer โดย Reader จะสนับสนุนการอ่านผ่านการทำงานของเมธอดที่ชื่อว่า Read และ Writer จะสนับสนุนการเขียนผ่านการทำงานของเมธอดที่ชื่อว่า Write มีหลายฟังก์ชันในโกที่ใช้ Reader และ Writer เป็นอาร์กิวเมนต์ เช่น ฟังก์ชัน Copy ที่เอาไว้คัดลอกข้อมูลจาก Reader ไปหา Writer

```
func Copy(dst Writer, src Reader) (written int64, err error)
```

โดยจะอ่านหรือเขียนไปที่ []byte หรือ string คุณสามารถใช้ struct ที่ชื่อว่า Buffer ซึ่งอยู่ในแพ็คเกจ bytes

```
package bytes
var buf bytes.Buffer
buf.Write([]byte("test"))
```

Buffer ไม่จำเป็นต้องกำหนดค่าเริ่มต้นให้มัน และสนับสนุนทั้งอินเทอร์เฟซ Reader และ Writer คุณสามารถแปลงมันไปเป็น []byte โดยเรียก buf.Bytes() ถ้าคุณต้องการที่จะอ่านจาก string คุณสามารถที่จะเรียก strings.NewReader ซึ่งจะมีประสิทธิภาพดีกว่าการใช้ buffer

File & Folders

การจะเปิดไฟล์ในโกก็จะใช้ฟังก์ชัน Open ในแพ็คเกจ os ในตัวอย่างนี่จะเป็นวิธีการอ่านเนื้อหาในไฟล์มาแสดงผลที่เทอร์มินัล

```
package main
```

```
import (
    "fmt"
    "os"
)
```

```

func main() {
    file, err := os.Open("test.txt")
    if err != nil {
        // handle the error here
        return
    }
    defer file.Close()

    // get the file size
    stat, err := file.Stat()
    if err != nil {
        return
    }
    // read the file
    bs := make([]byte, stat.Size())
    _, err = file.Read(bs)
    if err != nil {
        return
    }

    str := string(bs)
    fmt.Println(str)
}

```

เราใช้ `defer file.Close()` หลังเปิดไฟล์เพื่อที่จะทำให้มั่นใจได้ว่าจะปิดไฟล์เมื่อฟังก์ชันจบการทำงานลง การอ่านไฟล์เป็นอะไรที่ง่ายมากและเราสามารถทำให้มันสั้นกว่าได้ด้วยการทำแบบนี้

```

package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    bs, err := ioutil.ReadFile("test.txt")
    if err != nil {
        return
    }
    str := string(bs)
    fmt.Println(str)
}

```

อันนี้เป็นวิธีการสร้างไฟล์

```

package main

import (
    "os"
)

func main() {
    file, err := os.Create("test.txt")
    if err != nil {
        // handle the error here
        return
    }
    defer file.Close()

    file.WriteString("test")
}

```

การที่จะดึงเนื้อหาของไฟล์เดอร์เราจะใช้ฟังก์ชัน `os.Open` เหมือนเดิม แต่จะใช้ที่อยู่ของไฟล์เดอร์แทนที่อยู่ของไฟล์ และเราจะเรียกเมธอด `ReadDir` ได้

```

package main

```

```

import (
    "fmt"
    "os"

```

```
)
```

```
func main() {  
    dir, err := os.Open(".")  
    if err != nil {  
        return  
    }  
    defer dir.Close()  
  
    fileInfos, err := dir.Readdir(-1)  
    if err != nil {  
        return  
    }  
    for _, fi := range fileInfos {  
        fmt.Println(fi.Name())  
    }  
}
```

หากเราต้องการที่จะเข้าไปในโฟลเดอร์ เพื่อที่จะอ่านเนื้อหาใน sub-folder หรือ sub-sub-folder โทมีวิธีการที่ง่ายมาก นั่นคือการใช้ฟังก์ชัน Walk ซึ่งอยู่ในแพ็คเกจ path/filepath

```
package main  
  
import (  
    "fmt"  
    "os"  
    "path/filepath"  
)  
  
func main() {  
    filepath.Walk(".", func(path string, info os.FileInfo, err  
error) error {  
        fmt.Println(path)  
        return nil  
    })  
}
```

จากตัวอย่างด้านบน ฟังก์ชันที่ใส่เข้าไปใน Walk จะถูกเรียกทุกไฟล์และโฟลเดอร์ในรูปโฟลเดอร์

ข้อผิดพลาด (Errors)

โกมีไทป์ที่ถูกสร้างมาแล้วสำหรับข้อผิดพลาดซึ่งเราได้เคยเห็นมาแล้ว (type error) เราสามารถสร้าง error ของตัวเองโดยใช้ฟังก์ชัน New ในแพ็คเกจ errors

```
package main

import "errors"

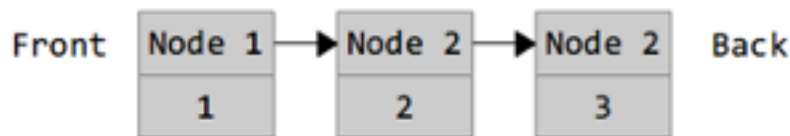
func main() {
    err := errors.New("error message")
}
```

Containers & Sort

lists และ maps ของโกมี collections ให้เยอะมากภายใต้แพ็คเกจ container เราจะดูในแพ็คเกจ container/list จากตัวอย่าง

List

แพ็คเกจ container/list ได้ิฉิมพลีเม้นต์ doubly-linked list ซึ่ง linked list คือโครงสร้างข้อมูลที่มีหน้าตาตามรูปด้านล่าง



ในแต่ละโหนดของ list จะมีค่า (1, 2, 3 ในตัวอย่าง) และมีพอยต์เตอร์ชี้ไปหาโหนดถัดไป แต่ doubly-linked list ในแต่ละโหนดจะมีพอยต์เตอร์ที่ชี้กลับไปหาโหนดก่อนหน้าด้วย ในลิสต์นี้สามารถสร้างโดยโปรแกรมจากตัวอย่างด้านล่าง

```
package main

import ("fmt" ; "container/list")

func main() {
    var x list.List
    x.PushBack(1)
    x.PushBack(2)
    x.PushBack(3)
```

```
    for e := x.Front(); e != nil; e=e.Next() {  
        fmt.Println(e.Value.(int))  
    }  
}
```

zero value ของ List คือ list ว่าง (*List สามารถสร้างได้โดย list.New) Values จะถูกต่อเข้าไปใน
ลิสต์ด้วยการใช้ PushBack เรวนรอบแต่ละไอเท็มในลิสต์โดยดึงค่าอีลิเมนต์ตัวแรก และไปเรื่อยๆ จน
กระทั่งพบว่าค่านั้นเป็น nil

Sort

แพ็คเกจ sort บรรจุฟังก์ชันสำหรับเรียงข้อมูล ซึ่งมีฟังก์ชันสำหรับ slices ints และ floats ไว้ให้ก่อน
แล้ว ในตัวอย่างนี้เป็นวิธีการเรียงข้อมูลของเราเอง

```

package main

import ("fmt" ; "sort")

type Person struct {
    Name string
    Age int
}

type ByName []Person

func (this ByName) Len() int {
    return len(this)
}
func (this ByName) Less(i, j int) bool {
    return this[i].Name < this[j].Name
}
func (this ByName) Swap(i, j int) {
    this[i], this[j] = this[j], this[i]
}

func main() {
    kids := []Person{
        {"Jill", 9},
        {"Jack", 10},
    }
    sort.Sort(ByName(kids))
    fmt.Println(kids)
}

```

ฟังก์ชัน Sort ใน sort ใช้ sort.Interface และเรียงมัน sort.Interface ต้องการ 3 เมธอดคือ Len, Less และ Swap การประกาศ sort ของเราเองโดยสร้าง type ที่ชื่อว่า ByName และสร้างมันเหมือนกับ slice ที่เราต้องการจะเรียง เราก็ประกาศ 3 methods ดังกล่าว

การเรียง list ของ people นั้นง่ายมากด้วยการ casting list ไปเป็น type ใหม่ หากเราต้องการเรียงตามอายุได้โดย

```

type ByAge []Person
func (this ByAge) Len() int {
    return len(this)
}
func (this ByAge) Less(i, j int) bool {
    return this[i].Age < this[j].Age
}
func (this ByAge) Swap(i, j int) {
    this[i], this[j] = this[j], this[i]
}

```

Hashes & Cryptography

ฟังก์ชันแฮช(hash) ช่วยทำให้เกิดกลุ่มของข้อมูลและลดขนาดให้เล็กกว่าเดิมโดยมีขนาดที่แน่นอน ซึ่งแฮชจะถูกใช้บ่อยในการเขียนโปรแกรมเพื่อที่จะตรวจสอบว่าข้อมูลมีการเปลี่ยนแปลง ในโกจะแบ่งแฮชออกเป็นสองกลุ่มหลักคือ กลุ่มที่เกี่ยวข้องกับการเข้ารหัส (cryptographic) และกลุ่มที่ไม่เกี่ยวข้องกับการเข้ารหัส (non-cryptographic)

ฟังก์ชันแฮชในกลุ่มที่ไม่เกี่ยวข้องกับการเข้ารหัสจะอยู่ในแพคเกจที่ชื่อว่า hash และรวมถึง `adler32`, `crc32`, `crc64` และ `fnv` เราจะแสดงให้ดูถึงตัวอย่างการใช้ `crc32`

```

package main

import (
    "fmt"
    "hash/crc32"
)

func main() {
    h := crc32.NewIEEE()
    h.Write([]byte("test"))
    v := h.Sum32()
    fmt.Println(v)
}

```

แฮชแบบ `crc32` จะอิมพลีเมนต์ interface `Writer` ทำให้เราสามารถเขียน bytes ลงไปได้เหมือน `Writer` ตัวอื่นๆ หลังจากที่เราเขียนสิ่งที่เราต้องการลงไป เราก็เรียก `Sum32()` ซึ่งจะคืน `uint32` กลับมา ในการใช้งานทั่วไป `crc32` จะเอาไว้ใช้เปรียบเทียบไฟล์สองไฟล์ ถ้า `Sum32` มีค่าเหมือนกัน มีโอกาสสูงมาก (แต่ไม่ถึง 100%) ที่จะเป็นไฟล์เดียวกัน ถ้าค่าต่างก็แสดงว่าทั้งสองไฟล์นั้นไม่เหมือนกัน


```

package main

import (
    "fmt"
    "hash/crc32"
    "io/ioutil"
)

func getHash(filename string) (uint32, error) {
    bs, err := ioutil.ReadFile(filename)
    if err != nil {
        return 0, err
    }
    h := crc32.NewIEEE()
    h.Write(bs)
    return h.Sum32(), nil
}

func main() {
    h1, err := getHash("test1.txt")
    if err != nil {
        return
    }
    h2, err := getHash("test2.txt")
    if err != nil {
        return
    }
    fmt.Println(h1, h2, h1 == h2)
}

```

ฟังก์ชันแฮชในกลุ่มที่เกี่ยวข้องกับการเข้ารหัส จะคล้ายๆ กับกลุ่ม non-cryptographic แต่จะเพิ่มคุณสมบัติที่ทำให้คำนวณย้อนกลับได้ยาก เพื่อให้ยากต่อการหาข้อมูลตั้งต้น ฟังก์ชันแฮชในกลุ่มนี้เราจะพบเห็นใน security applications

หนึ่งในตัวอย่างที่เรารู้จักกันคือ SHA-1 โดยวิธีการใช้เป็นตามตัวอย่างด้านล่าง

```

package main

import (
    "fmt"
    "crypto/sha1"
)

func main() {
    h := sha1.New()
    h.Write([]byte("test"))
    bs := h.Sum([]byte{})
    fmt.Println(bs)
}

```

จากตัวอย่างจะเห็นว่าคล้ายกับการใช้ crc32 ในตัวอย่างก่อนหน้านี้ เพราะทั้ง crc32 และ sha1 ได้
 อิมพลีเมนต์อินเทอร์เฟส hash.Hash ความแตกต่างหลักของทั้งสองแบบคือ crc32 นั้นจะคำนวณด้วย
 แอส 32 bit ส่วน sha1 จะคำนวณด้วยแอส 160 bit ไม่มี native type อันไหนที่เป็นตัวเลข 160 bit
 ดังนั้นเราเลยใช้สไลซ์ของ 20 ไบท์แทน

เซิร์ฟเวอร์ (Servers)

การเขียน network server ในโกนั้นง่ายมาก เราจะแสดงให้เห็นว่าเราจะสร้าง TCP server
 อย่างไร

```
package main
```

```
import (  
    "encoding/gob"  
    "fmt"  
    "net"  
)
```

```
func server() {  
    // listen on a port  
    ln, err := net.Listen("tcp", ":9999")  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
    for {  
        // accept a connection  
        c, err := ln.Accept()  
        if err != nil {  
            fmt.Println(err)  
            continue  
        }  
        // handle the connection  
        go handleServerConnection(c)  
    }  
}
```

```
func handleServerConnection(c net.Conn) {  
    // receive the message  
    var msg string  
    err := gob.NewDecoder(c).Decode(&msg)  
    if err != nil {  
        fmt.Println(err)  
    } else {  
        fmt.Println("Received", msg)  
    }  
}
```

```
    c.Close()  
}
```

```

func client() {
    // connect to the server
    c, err := net.Dial("tcp", "127.0.0.1:9999")
    if err != nil {
        fmt.Println(err)
        return
    }

    // send the message
    msg := "Hello World"
    fmt.Println("Sending", msg)
    err = gob.NewEncoder(c).Encode(msg)
    if err != nil {
        fmt.Println(err)
    }

    c.Close()
}

func main() {
    โกserver()
    โกclient()

    var input string
    fmt.Scanln(&input)
}

```

ในตัวอย่างจะใช้แพ็คเกจ `encoding/gob` ซึ่งจะช่วยให้ง่ายในการเข้ารหัสค่าของโกไปที่โกโปรแกรมอื่นๆ (หรือในโกโปรแกรมเดียวกัน) สามารถอ่านได้ การ encoding จะอยู่ในแพ็คเกจที่อยู่ใต้แพ็คเกจ encoding เช่น `encoding/json` หรือใน 3rd party แพ็คเกจ (เช่นในตัวอย่างเราใช้ `labix.org/v2/mgo/bson` สำหรับ `bson`)

HTTP

HTTP server ง่ายมากในการสร้างและใช้งาน:

```

package main

import ("net/http" ; "io")

func hello(res http.ResponseWriter, req *http.Request) {
    res.Header().Set(
        "Content-Type",
        "text/html",
    )
    io.WriteString(
        res,
        `<doctype html>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    Hello World!
  </body>
</html>`,
    )
}

func main() {
    http.HandleFunc("/hello", hello)
    http.ListenAndServe(":9000", nil)
}

```

HandleFunc จะ handle URL route (/hello) โดยเรียกใช้ฟังก์ชันที่ให้เรา เราสามารถ handle static files โดยใช้ FileServer

```
http.Handle(  
    "/assets/",  
    http.StripPrefix(  
        "/assets/",  
        http.FileServer(http.Dir("assets")),  
    ),  
)
```

RPC

แพ็คเกจ `net/rpc` (remote procedure call) และแพ็คเกจ `net/rpc/jsonrpc` จะทำให้เมธอดถูกเรียกใช้งานข้ามเครือข่ายได้ (แทนที่จะเรียกได้เฉพาะโปรแกรมที่รันมันขึ้นมา)

```
package main
```

```
import (  
    "fmt"  
    "net"  
    "net/rpc"  
)
```

```
type Server struct {}  
func (this *Server) Negate(i int64, reply *int64) error {  
    *reply = -i  
    return nil  
}
```

```
func server() {  
    rpc.Register(new(Server))  
    ln, err := net.Listen("tcp", ":9999")  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
    for {  
        c, err := ln.Accept()  
        if err != nil {  
            continue  
        }  
        go rpc.ServeConn(c)  
    }  
}
```

```
func client() {  
    c, err := rpc.Dial("tcp", "127.0.0.1:9999")  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
    var result int64  
    err = c.Call("Server.Negate", int64(999), &result)  
    if err != nil {  
        fmt.Println(err)  
    } else {
```

```

        fmt.Println("Server.Negate(999) =", result)
    }
}
func main() {
    โท้กserver()
    โท้กclient()

    var input string
    fmt.Scanln(&input)
}

```

ในโปรแกรมนี้จะคล้ายกับตัวอย่างใน TCP ยกเว้นเราสร้างวัตถุที่มีทุกเมธอดที่ต้องการเปิดเผยและเราเรียกเมธอด Negate จาก client ดูในเอกสารของ net/rpc สำหรับข้อมูลเพิ่มเติม

Parsing Command Line Arguments

เมื่อเราต้องการเรียกคำสั่งบนเทอร์มินัลโดยการ pass อาร์กิวเมนต์ของคำสั่งนั้น เราจะได้เห็นได้ในคำสั่ง go:

```
go run myfile.go
```

run และ myfile.go คือ อาร์กิวเมนต์ เราสามารถส่ง flags ไปที่คำสั่งได้โดย

```
go run -v myfile.go
```

แพกเกจ flag ช่วยให้เราสามารถ parse อาร์กิวเมนต์ และ flags ที่จะส่งไปให้โปรแกรมของได้ในตัวอย่างจะเป็นโปรแกรมที่จะสร้างตัวเลขระหว่าง 0 ถึง 6 เราสามารถเปลี่ยนค่าสูงสุดได้โดยส่ง flag -max=100 ไปที่โปรแกรม


```

package main

import ("fmt"; "flag"; "math/rand")

func main() {
    // Define flags
    maxp := flag.Int("max", 6, "the max value")
    // Parse
    flag.Parse()
    // Generate a number between 0 and max
    fmt.Println(rand.Intn(*maxp))
}

```

อะไรก็ตามที่ไม่ใช่ flag อาร์กิวเมนต์สามารถที่จะเอามาได้โดย flag.Args() มันจะคืน []string ออกมา

Synchronization Primitives

เราได้บอกวิธีทางที่จะจัดการการทำงานในเวลาเดียวกัน และ synchronization ในโกซึ่งอยู่ในบทที่ 10 ไปแล้ว อย่างไรก็ตามก็ยังยอมให้ใช้รูปแบบเดิมๆ ในการจัดการ multithreading routines ซึ่งอยู่ในแพ็คเกจ sync และ sync/atomic

Mutexes

mutex จะ locks ส่วนหนึ่งในโค้ดไปที่ single thread ในขณะนั้น และใช้ป้องกันการแย่ง shared resource จาก non-atomic operations นี่เป็นตัวอย่างของ mutex

```

package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    m := new(sync.Mutex)

    for i := 0; i < 10; i++ {
        go func(i int) {
            m.Lock()
            fmt.Println(i, "start")
            time.Sleep(time.Second)
            fmt.Println(i, "end")
            m.Unlock()
        }(i)
    }

    var input string
    fmt.Scanln(&input)
}

```

เมื่อ mutex (ตัวแปร m ในตัวอย่างด้านบน) ได้ lock มันจะบล็อกการทำงานจนกว่าเราจะสั่ง unlock จะต้องระวังมากเมื่อใช้ mutex หรือ synchronization ที่มีให้ในแพ็คเกจ sync/atomic

โดยทั่วไป multithreaded programming มันยาก ซึ่งสามารถสร้างความผิดพลาดได้ง่ายและยากที่จะหา มันหนึ่งในจุดแข็งของโกคือความสามารถของ concurrency ที่ง่ายต่อการเข้าใจได้และนำไปใช้งานได้อย่างเหมาะสมกว่า thread และ locks