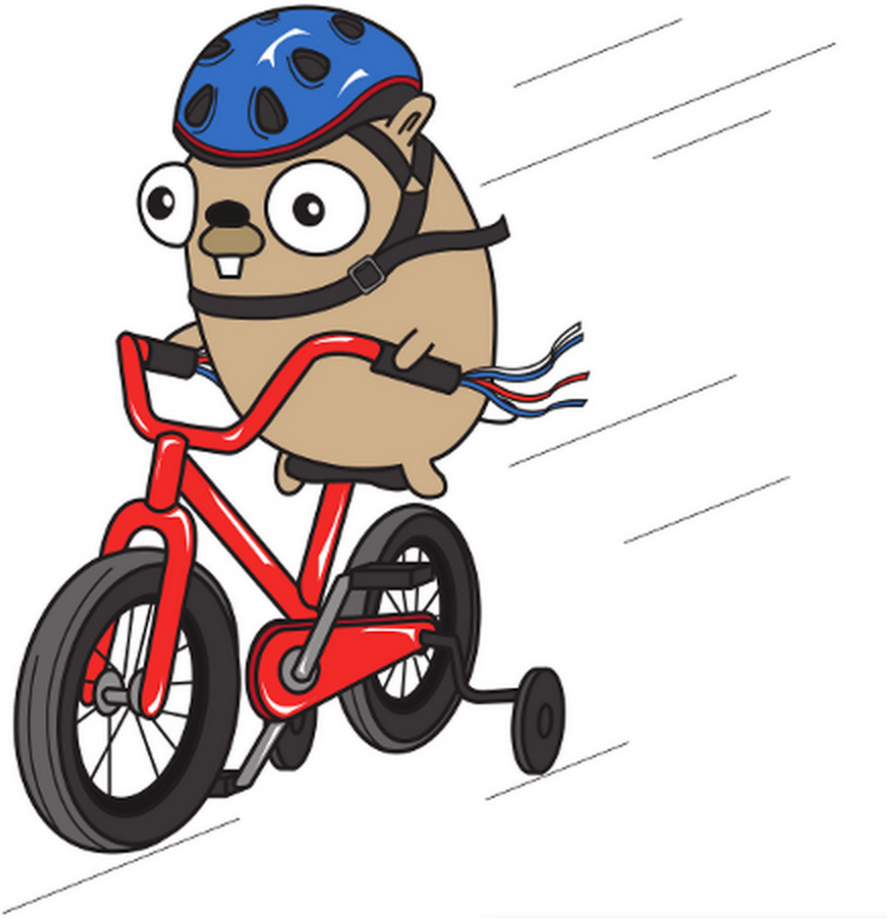
แนะนำการเขียนโปรแกรมด้วยภาษาโก(Golang)



โดย เซเลบ ด๊อกซี่

บทที่1: บทนำ

การเขียนโปรแกรมนั้นเป็นทั้งศิลปะ,งานฝีมือและวิทยาศาสตร์ เพื่อสั่งงานให้คอมพิวเตอร์ ให้ทำงานตามที่เราต้องการ หนังสือเล่มนี้จะสอนให้คุณให้รู้จักกับการเขียนโปรแกรมคอมพิวเตอร์ด้วยภาษาในการเขียนโปรแกรมที่ออกแบบโดย Google ที่ชื่อว่าภาษาโก (Go)

โก เป็นภาษาที่ใช้ในการเขียนโปรแกรมเชิงอรรถประโยชน์ (general purpose programming language) ที่มาพร้อมกับฟีเจอร์ที่ก้าวหน้าต่างๆ มากมาย อีกทั้งไวยากรณ์ของภาษาที่ดูสะอาดสะอ้านและเนื่องจากเป็นภาษาสามารถใช้งานได้ในหลากหลายแพลตฟอร์ม ซึ่งมีไลบรารี่พื้นฐานที่มีความทนทานต่อข้อผิดพลาดและมีเอกสารประกอบที่ครบถ้วนสมบูรณ์ ทั้งยังเป็นภาษาที่คำนึงถึงการออกแบบตามหลักการของวิศวกรรมซอฟท์แวร์ที่ดีอีกด้วย ดังนั้นอาจจะเรียกได้ว่าเป็นภาษาในอุดมคติสำหรับผู้ที่เริ่มเรียนรู้การเขียนโปรแกรมเป็นภาษาแรกเลยทีเดียว

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

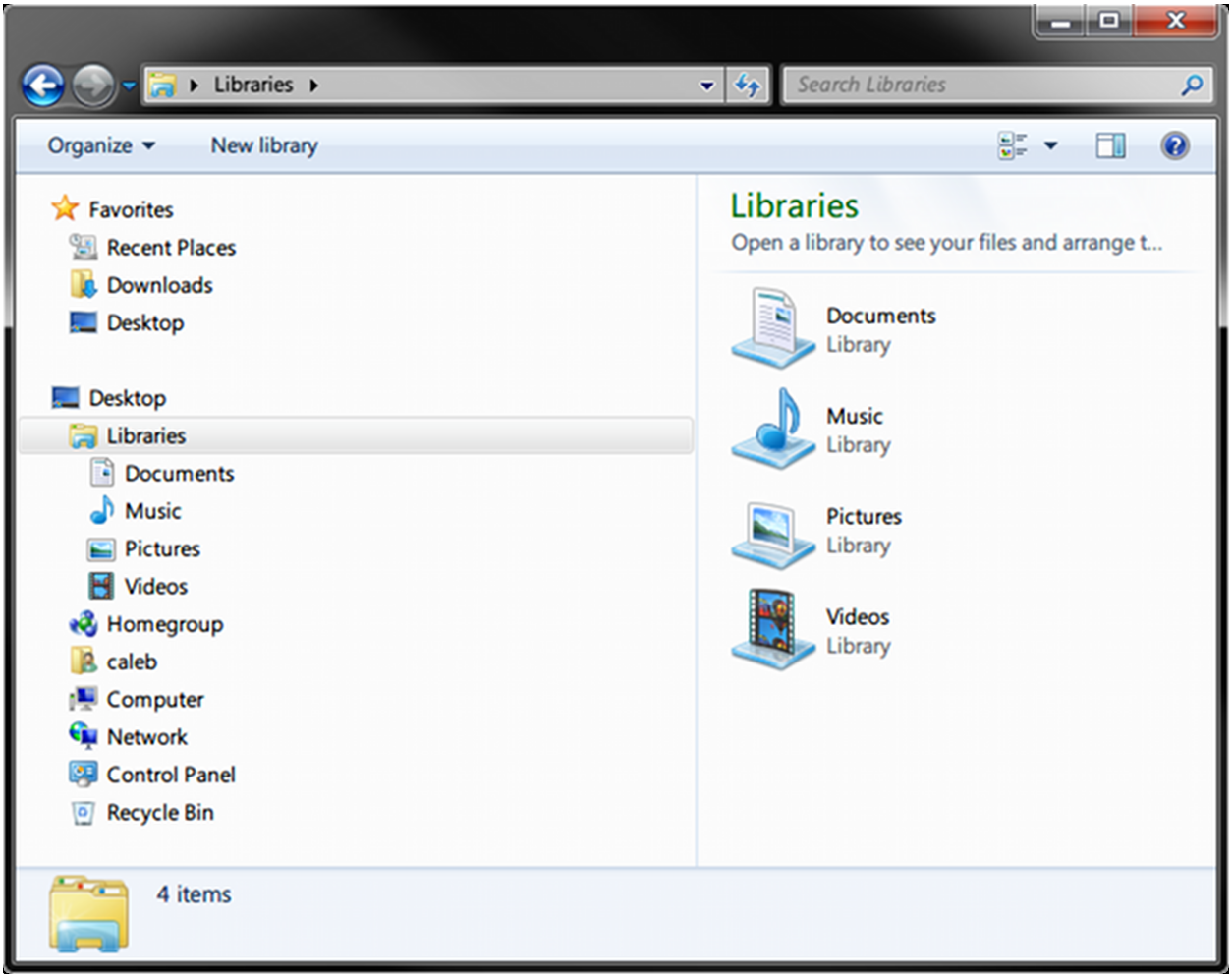
ไฟล์ คือกลุ่มของข้อมูลที่เก็บอยู่เป็นหน่วยเดียวกันโดยมีชื่อเรียก ระบบปฏิบัติการสมัยใหม่ (อย่าง วินโดวส์ หรือ Mac OSX) จะบรรจุไปด้วยไฟล์นับล้านไฟล์ ซึ่งจัดเก็บสารสนเทศมากมายหลากหลายประเภท นับตั้งแต่เทกท์ (text) จนถึงไฟล์ที่พร้อมทำงาน (executable) และไฟล์ประเภทมัลติมีเดียทั้งหลาย ไฟล์ทุกอันจะถูกจัดเก็บในแบบเดียวกันในคอมพิวเตอร์ โดยแต่ละไฟล์จะมีชื่อ ขนาดที่ชัดเจน (หน่วยเป็นไบต์) และประเภทของไฟล์ โดยทั่วไปประเภทของไฟล์จะถูกบ่งบอกด้วยนามสกุลของไฟล์ (คือส่วนของชื่อไฟล์ที่ตามหลังเครื่องหมาย `.` ยกตัวอย่างเช่น `hello.txt` จะมีนามสกุลเป็น `txt` ซึ่งจะจัดเก็บข้อมูลประเภทตัวอักษร)

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

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

วินโดวส์

ในระบบวินโดวส์ นั้น เราจะสามารถเปิดดูไฟล์และโฟลเดอร์ได้โดยใช้ Windows Explorer (เรียกใช้งานโดยดับเบิ้ลคลิ๊ก “”My Computer หรือกดปุ่ม win+e)



OSX

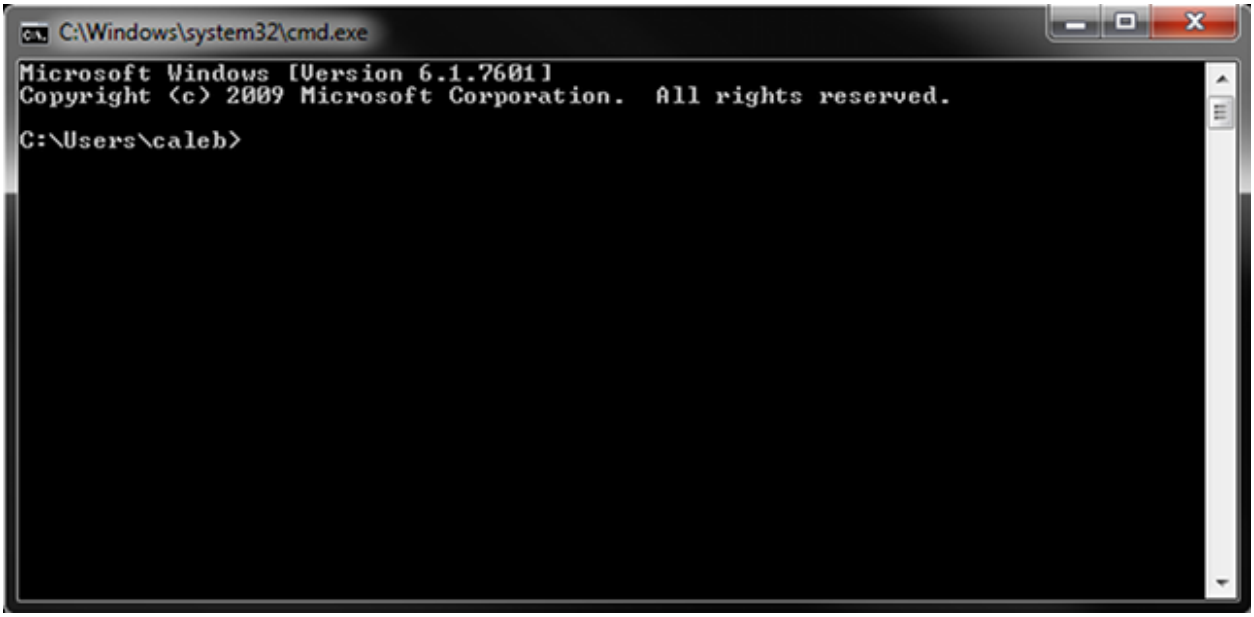
ส่วนใน OSX นั้น เราสามารถเรียกดูไฟล์และโฟลเดอร์ได้โดยใช้ Finder (เรียกใช้งานได้โดยคลิ๊กที่ไอคอน Finder - ไอคอนรูปใบหน้าที่อยู่ด้านล่างซ้ายมือของบาร์)

1.2 เทอร์มินัล

การใช้งานคอมพิวเตอร์ในปัจจุบันนั้น กระทำผ่านทางส่วนติดต่อผู้ใช้งาน แบบกราฟฟิกที่ซับซ้อน (GUIs) โดยการใช้งานคีย์บอร์ด เมาส์และทัชสกรีนในการติดต่อกับปุ่ม หรือคอนโทรลแบบต่างๆ ที่แสดงบนหน้าจอ แต่ก็ไม่ได้เป็นเช่นนี้เสมอไป ก่อนที่เราจะมี GUI เราใช้งานผ่านทาง เทอร์มินัล – ส่วนติดต่อคอมพิวเตอร์แบบอักขระ โดยแทนที่เราจะจัดการปุ่มบนหน้าจอ เราก็สั่งงานด้วยคำสั่งและรอรับผลลัพธ์ตอบกลับ เสมือนเรากำลังคุยกับคอมพิวเตอร์ ถึงแม้ในโลกของคอมพิวเตอร์ปัจจุบันดูเหมือนจะทอดทิ้งเทอร์มินัลให้เป็นเสมือนวัตถุโบราณ แต่ความจริงก็คือว่าเทอร์มินัล ยังเป็นส่วนติดต่อพื้นฐานที่ภาษาโปรแกรมส่วนใหญ่ใช้ในการติดต่อกับกับคอมพิวเตอร์ โดยภาษาโกนั้น ก็ไม่ได้แตกต่างแต่อย่างใด ดังนั้นก่อนที่เราจะเริ่มเขียนโปรแกรมด้วยภาษาโก เราควรมีความเข้าใจพื้นฐานในการทำงานของเทอร์มินัล

วินโดวส์

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



โดยปรกติคอมมานไลน์จะเริ่มทำงานในไดเร็คทอรีโฮม ของคุณ (ในกรณีของผมนั้นคือ `C:\Users\caleb`) เราสั่งงานโดยพิมพ์คำสั่งลงไปแล้วกดปุ่ม enter ลองพิมพ์คำสั่ง `dir` ดู ซึ่งเป็นคำสั่งที่ใช้ในการแสดงรายการของสิ่งที่อยู่ภายในไดเร็คทอรี ซึ่งเราควรเห็นผลลัพธ์ดังนี้

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

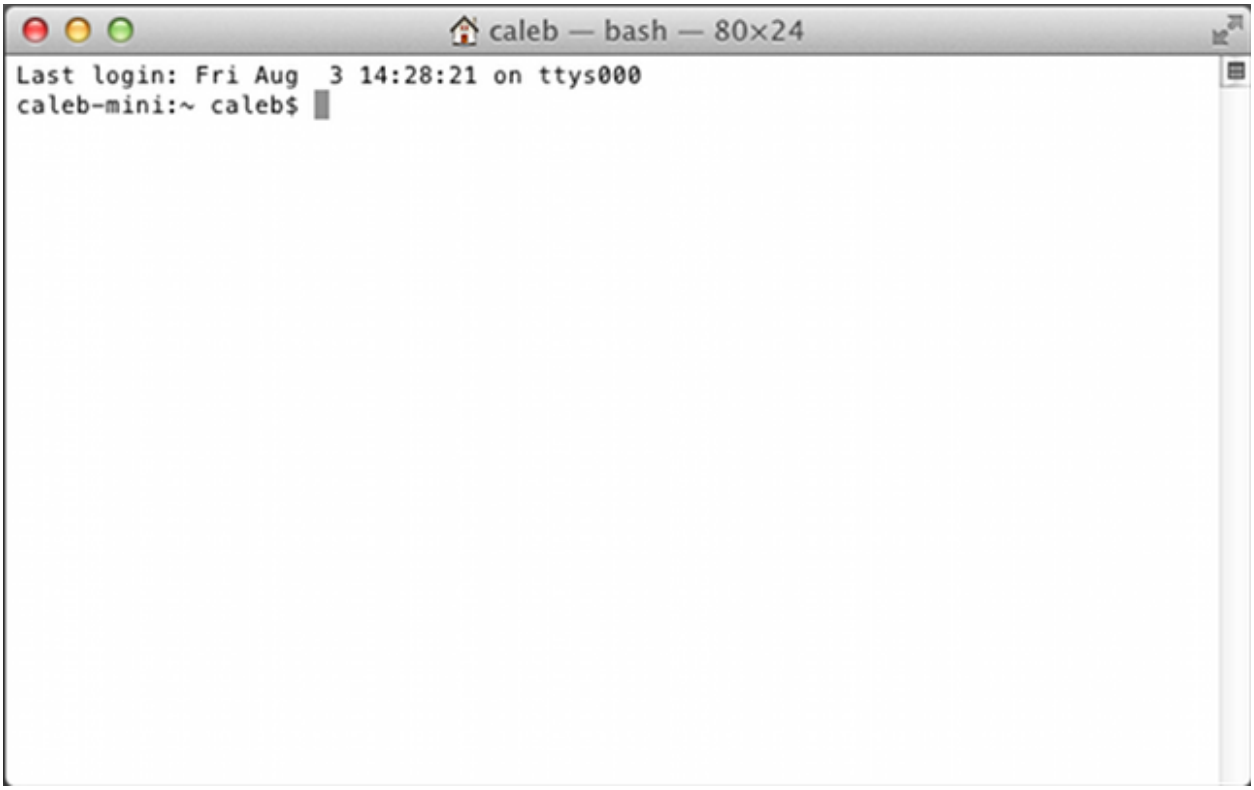
โดยจะตามด้วยรายการของไฟล์และโฟลเดอร์ที่อยู่ในไดเร็คทอรีโฮม ของคุณ และเราสามารถที่จะเปลี่ยนไดเร็คทอรีโดยใช้คำสั่ง `cd` ยกตัวอย่างเช่น อาจจะมีโฟลเดอร์ชื่อ Desktop เราสามารถเข้าไปดูข้างในได้ด้วยคำสั่ง `cd Desktop` ต่อด้วยคำสั่ง `dir` และหากต้องการกลับไปที่ไดเร็คทอรีโฮม สามารถทำได้โดยใช้ชื่อไดเร็คทอรี พิเศษชื่อ `..` (จุดสองอันติดกัน):

```
cd ..
```

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

OSX

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



โดยปรกติเทอร์มินัลจะเริ่มทำงานในไดเร็คทอรีโฮม ของคุณ (ในกรณีของผมคือ `/Users/caleb`) เราสามารถสั่งงานได้โดยพิมพ์คำสั่งลงไปตามด้วย enter ลองพิมพ์คำสั่ง `ls` ดู จะเป็นการใช้ในการเรียกดูเนื้อหาของไดเร็คทอรีใดๆ ซึ่งควรจะได้ผลลัพธ์ดังนี้

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

ระบบจะแสดงรายการของไฟล์และโฟลเดอร์ต่างๆ ที่อยู่ในไดเร็คทอรีโฮม ของคุณ(ในกรณีนี้ตัวอย่างจะมีแต่โฟลเดอร์ไม่มีไฟล์) เราสามารถเปลี่ยนไดเร็คทอรีได้โดยใช้คำสั่ง `cd` ยกตัวอย่างเช่น ในกรณีที่คุณมีโฟลเดอร์ชื่อ Desktop เราสามารถดูสิ่งที่อยู่ข้างในได้โดยการสั่ง `cd Desktop` แล้วตามด้วย `ls` และหากต้องการกลับไปที่ไดเร็คทอรีโฮม คุณสามารถทำได้โดยใช้ชื่อ directory พิเศษชื่อ `..` (จุดสองอันติดกัน):

```
cd ..
```

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

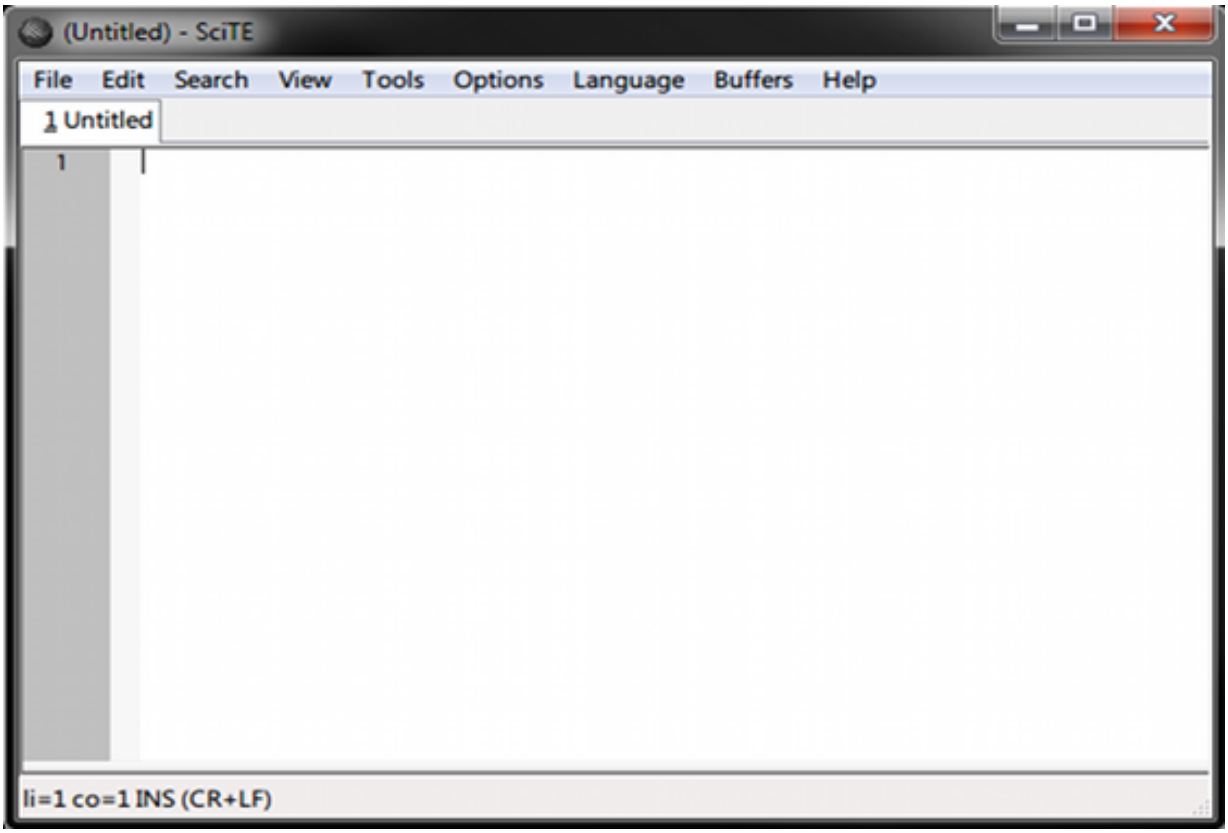
1.3 Text Editors

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

เพื่อเป็นการอำนวยความสะดวกให้การติดตั้งทำได้ง่าย โปรแกรมติดตั้งจะอยู่ที่เวปไซต์ของหนังสือ: http://www.golang-book.com/ โดยโปรแกรมติดตั้งจะทำทั้งติดตั้งชุดเครื่องมือต่างๆ ของภาษาโก และ 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 คอลัมภ์ที่ 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:

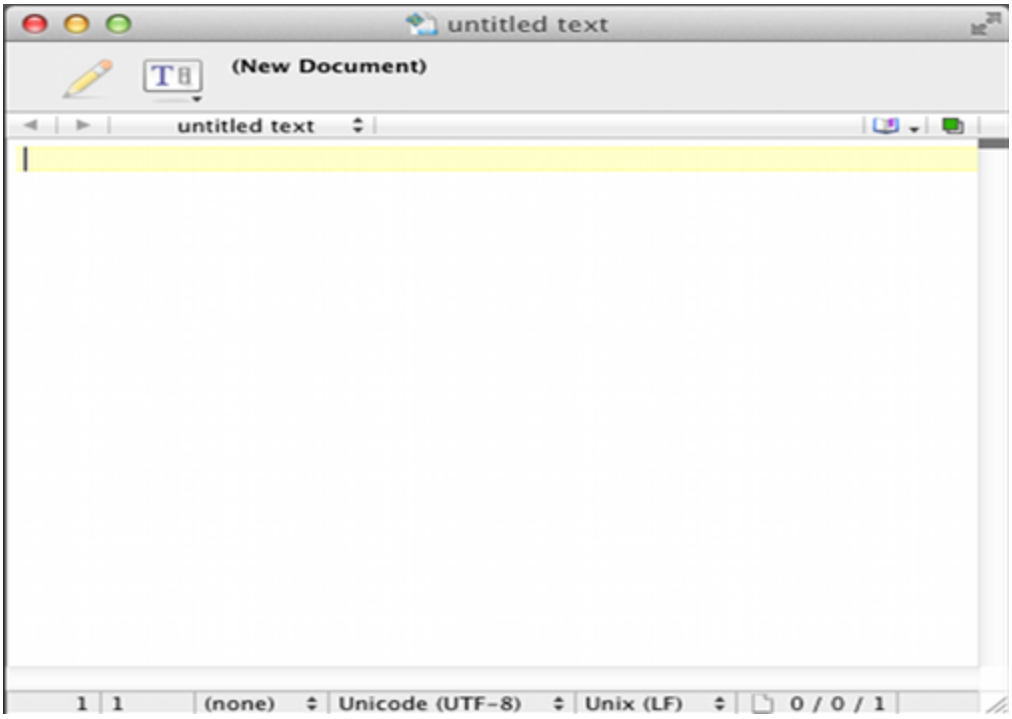
- Ctrl + S – บันทึกไฟล์
- Ctrl + X – ตัดข้อความที่ถูกเลือกนำไปใส่ไว้ในคลิปบอร์ดทำให้สามารถนำไปวางลง ที่อื่นได้ในภายหลัง
- Ctrl + C – คัดลอกข้อความที่ถูกเลือก
- Ctrl + V – วางข้อความที่เก็บอยู่ในคลิปบอร์ด
- ปุมลูกศรใช้ในการเลื่อนเคอร์เซอร์ไปในที่ต่างๆ ปุม Home เพื่อกระโดดไปยังต้นบรรทัด และปุม End เพื่อไปยังท้ายบรรทัด

• กดปุม shift ค้างไว้ พร้อมกับกดปมลูกศร (ปุม Home หรือปุม End) เพื่อเลือกข้อความโดยไม่ต้องใช้เมาส์ลาก

• Ctrl + F – เปิดกล่องค้นหาเพื่อค้นหาข้อความในเนื้อหาของไฟล์นั้น

OSX

สำหรับระบบ OSX โปรแกรมติดตั้งจะติดตั้ง text editor ชื่อ Text Wrangler



และก็คล้ายๆ กับโปรแกรม Scite บน window โปรแกรม Text Wrangler จะมีพื้นที่ให้พิมพ์ข้อมความลงไป โดยสามารเปิดไฟล์ได้โดยเลือก File → Open และบันทึกไฟล์โดยเลือก File → Save หรือ File → Save As และต่อไปนี้คือ shortcuts ที่มีประโยชน์ (Command คือปุม ⌘)

• Command + S – บันทึกไฟล์

• Command + X – ตัดข้อมความที่ถูกเลือกนำไปใส่ไว้ในคลิปบอร์ด ทำให้สามารถนำไปวางที่อื่นได้ในภายหลัง

• Command + C – คัดลอกข้อความที่ถูกเลือก

• Command + V – วางข้อความที่เก็บอยู่ในคลิปบอร์ด

• ปุมลูกศรใช้ในการเลื่อนเคอร์เซอร์ไปในที่ต่างๆ

• Command + F – เปิดกล่องค้นหาเพื่อค้นหาข้อความในเนื้อหาของไฟล์นั้น

1.4 เครื่องมือต่างๆ ของภาษาโก

ภาษาโก เป็นภาษาที่ต้องทำการคอมไพล์ก่อน (compiled programming language) ซึ่งหมายความว่า ซอร์สโค้ด (โปรแกรมที่เราเขียน) จะต้องถูกแปลงไปเป็นภาษาที่คอมพิวเตอร์สามารถเข้าใจได้ ดังนั้นก่อนที่เราจะสามารถเขียนโปรแกรมภาษาโกได้ เราต้องมีคอมไพเลอร์ของภาษาโกเสียก่อน
โปรแกรมติดตั้งจะติดตั้งภาษาโกให้เราแบบอัตโนมัติ โดยเราจะใช้เวอร์ชั่น 1 (สามารถหาข้อมูลเพิ่มเติมได้ที่ http://www.golang.org) เสร็จแล้วให้ตรวจสอบว่าทุกอย่างทำงานได้ถูกต้อง โดยการเปิดเทอร์มินัลแล้วพิมพ์ดังนี้:

```
go version
```

เราควรจะเห็นผลลัพธ์ดังนี้:

```
go version go1.0.2
```

โดยหมายเลขเวอร์ชั่นนั้นอาจจะต่างออกไปเล็กน้อย ถ้าระบบแสดงข้อผิดพลาดว่า ไม่รู้จักคำสั่งแล้วหละก็ ให้ลอง restart เครื่องคอมพิวเตอร์ดู
ชุดโปรแกรมของภาษาโก จะประกอบไปด้วยคำสั่ง และคำสั่งย่อยต่างๆ มากมาย หากเราต้องการดูรายการของคำสั่งเหล่านั้น สามารถทำได้โดยการพิมพ์คำสั่ง:

```
go help
```

ซึ่งในบทถัดๆ ไป เราจะได้เห็นว่าคำสั่งเหล่านี้จะถูกใช้งานอย่างไร

บทที่ 2: โปรแกรมแรกของคุณ (Your First Program)

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

เริ่มจากสร้างโฟลเดอร์ใหม่ในที่ ที่คุณจะสามารถเก็บโปรแกรมของคุณไว้ได้ ซึ่งตัวติดตั้งที่คุณใช้ในบทที่ 1 ได้สร้างโฟลเดอร์ชื่อว่า โก ไว้ใน โฮมไดเรคทอรี่ ของคุณ จากนั้นให้สร้างโฟลเดอร์ชื่อว่า ~/Go/src/golang-book/chapter2 (โดยที่ ~ หมายถึง โฮมไดเรคทอรี่ 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")
}
```

ตรวจสอบให้แน่ใจว่าไฟล์ของคุณเหมือนตามที่แสดงให้เห็นนี้ 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 ได้และบันทึกลงไปในไดเร็คทอร์รี่ชั่วคราว(temporary directory) และสั่งรันโปรแกรมนั้น ถ้าคุณไม่เห็น Hello World คุณอาจจะทำอะไรผิดสักอย่างตอนที่คุณเขียนโปรแกรม โดย โก คอมไพล์เลอร์ จะให้คำแนะนำคุณเกี่ยวกับจุดที่ผิดพลาดเหมือนอย่างที่คอมไพล์เลอร์อื่นๆทำ และ โก คอมไพล์เลอร์นั้น เข้มงวดมากๆและไม่ใจดีให้กับความผิดพลาดใดๆ

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

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

```
import "fmt"
```

คำหลัก(keyword) `import` คือการบอกว่าเราจะรวมโค้ดจาก แพกเกจ อื่นเข้ามาในโปรแกรมของเราอย่างไร

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

ขอให้รู้ว่า `fmt` ด้านบนนั้นถูกครอบไว้ด้วยเครื่องหมายฟันหนู(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)

หลังจากนี้คุณจะเห็นการประกาศฟังก์ชั่น

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

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

ฟังก์ชั่นทั้งหมดจะเริ่มต้นด้วยคำหลักว่า func ตามด้วยชื่อของฟังก์ชั่น(ในตัวอย่างนี้คือ main) พารามิเตอร์ตั้งแต่ ศูนย์หรือมากกว่านั้นถูกครอบไว้ด้วยเครื่องหมายวงเล็บ การรีเทิร์นค่าเป็นทางเลือกที่จะมีหรือไม่มีก็ได้ และ ส่วนบอร์ดี้(body) หรือเนื้อหา ถูกครอบไว้ด้วยวงเล็บปีกกาฟังก์ชั่นนี้ไม่มีพารามิเตอร์ ไม่มีการรีเทิร์นค่าใดๆ และมีแค่ statment เดียว ชื่อ main เป็นฟังก์ชั่นพิเศษ เพราะมันเป็นฟังก์ชั่นที่ถูกเรียกเมื่อคุณ execute โปรแกรม ส่วนสุดท้ายของโปรแกรมของเราคือบรรทัดนี้

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

statement นี้ถูกสร้างขึ้นจากสามองค์ประกอบ หนึ่ง เราเข้าถึงฟังก์ชั่นอื่นภายใน แพกเกจของ fmt ที่ชื่อ Println (นั่นคือ fmt.Println โดย Println หมายถึงพิมพ์ทีละบรรทัด) จากนั้นเราสร้าง สตริงใหม่ว่า Hello World และ invoke (หรือจะเรียกว่า call หรือ execute ก็ได้) ฟังก์ชั่นโดยส่งสตริงเป็นอากิวเม้นท์ตัวแรกและตัวเดียว

ณ จุดจุดนี้ เราพร้อมแล้วที่จะเห็นระบบคำแบบใหม่ๆอีกมากและคุณอาจจะรู้สึกเงิบนิดๆบางครั้งการตั้งใจอ่านโปรแกรมของคุณดังๆอาจมีประโยชน์ การอ่านโปรแกรมที่เพิ่งเขียนไปอาจจะเป็นแบบนี้:

สร้างโปรแกรมใหม่ที่ execute ได้ ซึ่งอ้างถึง ไลบรารี่ ชื่อ fmt และประกอบไปด้วยหนึ่งฟังก์ชั่นชื่อว่า main มันไม่มีอาร์กิวเม้นท์ ไม่รีเทิร์นค่าอะไรเลย และตามมาด้วยการเรียกใช้ฟังก์ชั่น Println ที่อยู่ใน แพกเกจ fmt และ เรียกใช้มันโดยใส่หนึ่งอาร์กิวเม้นท์ คือสตริง Hello World

ฟังก์ชั่น 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

กลับไปที่ฟังก์ชั่นของเรากันดีกว่า เอกสารนี้กำลังบอกคุณว่า ฟังก์ชั่น `Println` จะส่งอะไรก็ตามที่คุณให้มันไปแสดงที่ standard output - ชื่อของ output ของเทอร์มินัลที่คุณกำลังทำงานอยู่

ฟังก์ชั่นนี้ทำให้เห็น `Hello World`

ในบทต่อไปเราจะลงไปในรายละเอียดว่า โก เก็บและแสดงสิ่งที่เหมือนกับ `Hello World` อย่างไรโดยเรียนรู้เกี่ยวกับ ประเภทข้อมูล(Types)

ปัญหาท้าทาย(Problems)

- whitespace คืออะไร?
- comment คืออะไร? และสองวิธีที่จะเขียน comment คืออะไร?
- โปรแกรมของเราเริ่มต้นด้วย `package main` แล้วแฟ้ม `fmt` ควรจะเริ่มด้วยอะไร?
- เราได้ใช้ฟังก์ชั่น `Println` ที่ถูกประกาศไว้ใน package `fmt` ถ้าเราต้องการใช้ฟังก์ชั่น `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 (เพื่ออธิบายถึงคุณสมบัติทั่วๆไปของสิ่งที่กำลังพูดถึง) “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 เซ็ต (Set) กันอยู่บ่อยๆ ยกตัวอย่างเช่น $\mathbb{R}$ (ชุดของตัวเลขที่เป็นจำนวนจริง) หรือ $\mathbb{N}$ (ชุดของตัวเลขที่เป็นจำนวนนับ (counting number - เป็นจำนวนเต็มบวกเสมอ)) สมาชิกแต่ละตัวที่อยู่ในเซ็ตเหล่านี้จะมีคุณสมบัติที่เหมือนกันทุกตัวในเซ็ต นั้นๆ ยกตัวอย่างเช่น จำนวนนับทุกตัวล้วนสัมพันธ์เชื่อมโยงกัน เช่น มีจำนวนนับ A, B และ C, A + (B + C) จะเท่ากับ (A + B) + C หรือ A x (B x C) จะเท่ากับ (A x B) x C เสมอ ประเภทข้อมูลก็เหมือนกัน ค่าของตัวแปรที่มีประเภทข้อมูลเหมือนกัน จะมีคุณสมบัติเหมือนกันเสมอ

โก เป็นภาษาเขียนโปรแกรมประเภท static type (statically typed programming language) นั่นคือตัวแปรทุกตัวจะต้องระบุประเภทข้อมูลให้กับตัวแปรเสมอ และไม่สามารถเปลี่ยนแปลงประเภทข้อมูลของตัวแปรนั้นได้หลังจากระบุไปแล้ว ในช่วงแรก static type อาจจะดูยุ่งยากซับซ้อน และอาจต้องใช้เวลาอยู่พักนึงกว่าคุณจนแก้โปรแกรมจนสามารถคอมไพล์ได้ 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 (ตอบแบบกำปั้นทุบดินชะมัด) 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 ในระบบคอมพิวเตอร์เลขจำนวนเต็มจะมีการกำหนดขนาดไว้อย่างชัดเจน (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 และยังไม่ใช่เรื่องจำเป็นที่เราจะต้องเข้าใจเกี่ยวกับมัน ตอนนี้ขอให้จำไว้ว่า

- เลขทศนิยมเป็นตัวเลขที่ไม่เที่ยงตรง ตัวอย่างเช่น 1.01 - 0.99 แล้วได้ผลลัพธ์เท่ากับ 0.020000000000000018 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 เริ่มจากสร้างโฟล์เดอร์ชื่อ `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
```

สังเกตว่าโปรแกรมนี้ค่อนข้างเหมือนกับโปรแกรมที่เราเคยเขียนในบทที่ 2 ทั้งชื่อแพ็คเกจและไลบารี่ ที่อิมพอร์ตเข้ามา การประกาศฟังก์ชั่นและยังใช้ฟังก์ชั่น `Println` อีกด้วย แต่ในโปรแกรมนี้แทนที่จะแสดงคำว่า `Hello World` ออกหน้าจอ เราสั่งให้โปรแกรมแสดงผลลัพธ์เป็นข้อความ `1 + 1 =` แล้วตามด้วยผลการคำนวนผลบวกระหว่าง 1 + 1 ในคำสั่งนี้มีส่วนประกอบอยู่ 3 ส่วน คือเลขจำนวนเต็ม `1` , ตัวดำเนินการ `+` (การบวก), และเลขจำนวนเต็ม `1` อีกตัวหนึ่ง ทีนี้ลองทำแบบเดียวกันแต่เปลี่ยนไปใช้เลขทศนิยมแทนดังนี้

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

สังเกตว่าเราใช้ `.0` เพื่อบอก โก ว่านี่คือเลขทศนิยม เมื่อสั่งให้โปรแกรมทำงาน ก็จะยังได้ผลลัพธ์เหมือนเดิม

นอกจากตัวดำเนินการ + (การบวก) โก ยังเตรียมตัวดำเนินการอื่นๆ ไว้ให้ดังนี้

+	การบวก
-	การลบ
*	การคูณ
/	การหาร

%	การหาเศษ

สตริง (Strings)

อย่างที่เราได้เห็นในบทที่ 2 สตริง คือการเรียงกันของตัวอักษรด้วยความยาวที่แน่นอน เพื่อใช้เป็นตัวแทน “ข้อความ” สตริงใน โก นั้นเป็นการประกอบกันขึ้นมาจากไบต์ ซึ่งหนึ่งไบต์คือหนึ่งตัวอักษร (สำหรับตัวอักษรในภาษาอื่นๆ อย่างเช่น ภาษาจีนนั้น อาจจะต้องใช้มากกว่าหนึ่งไบต์) สตริงที่เป็นสัญพจน์สามารถสร้างได้ด้วยการใส่สัญพจน์ใน “” (double quote) เช่น `“Hello World”` หรือใน `` (back ticks) เช่น `` `Hello World` `` สิ่งที่ต่างกันคือ 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”` เอาล่ะมาลองแก้โปรแกรมที่เคยเขียนก่อนหน้าเพื่อทดลองสิ่งที่เราได้เรียนรู้เกี่ยวกับสตริงกัน

```
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 โกจะประมวลผลโดยขึ้นอยู่กับประเภทข้อมูลที่เรียกใช้ เมื่อ นิพจน์ทั้งซ้ายและขวาของเครื่องหมาย + เป็นสตริง คอมไพเลอร์จะรู้ว่าคุณหมายถึงการนำข้อความสองข้อความมาประกอบกัน ไม่ใช่การบวกตัวเลข**

บูลีน (Booleans)

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

&&	and
||	or
!	not

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: 101-1 = 9, 102-1 = 99)
- ให้ลองเขียนโปรแกรมเพื่อคำนวนผลคูณของ `32132 x 42452` แล้วแสดงผลออกมาทางหน้าจอดู (ให้ใช้เครื่อง `*` แทน x สำหรับการคูณในตอนที่เขียนโปรแกรม)
- 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” ซึ่งจะต้องกระทำการฝั่งขวาของเครื่องหมาย “=” ให้เรียบร้อยก่อน แล้วจึงเก็บค่าที่ได้ไปยังฝั่งซ้ายของเครื่องหมาย “=”

รูปแบบ x = x + y เป็นรูปแบบทั่วไปของการเขียน Code สำหรับ Go เราสามารถที่จะใช้ “+=” แทนได้ ดังนั้นจาก x = x + ”second” เราสามารถเขียนใหม่ได้เป็น x += “second” ซึ่งก็ได้ผลลัพธ์ออกมา แบบเดียวกัน และสามารถใช้กับ Operators อื่นๆ ได้เช่นกัน

อีกหนึ่งความแตกต่างระหว่าง Go และคณิตศาสตร์ คือ Go ใช้เครื่องหมาย “==” เพื่อเทียบว่า เท่ากัน หรือไม่โดยจะให้ 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 Complier จะมองว่ามีค่าเทียบเท่ากับ

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

เราสามารถใช้วิธีการนี้กับตัวแปรชนิดอื่นๆ ได้เช่นกัน ยกตัวอย่างเช่น

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

แนะนำว่าควรใช้รูปแบบนี้เมื่อมีโอกาส

วิธีการตั้งชื่อตัวแปร

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

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

จะเห็นว่าการตั้งชื่อตัวแปรว่า x ไม่สื่อความหมายของตัวแปรเท่าไร ลองเปลี่ยนเป็นแบบนี้

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

หรือ

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

ในกรณีนี้เราสามารถเลือกใช้วิธีการตั้งชื่อตัวแปลเป็นคำๆ โดยใช้รูปแบบที่เรียกว่า Camel Case โดยเริ่มต้นคำแรกด้วยชุดอักษรตัวพิมพ์เล็กทั้งหมด แล้วคำต่อๆ ไปตัวแรกของแต่ละคำเป็นตัวพิมพ์ใหญ่ ลักษณะจะเหมือนหลังอูฐ

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 ซึ่งจะขอยก การอธิบาย Scanf ไว้ในบทต่อไป สำหรับตอนนี้เรารู้ก่อนเพียงว่าเราสามารถกำหนดค่าตัวแปน input โดย Scanf จากการกรอกเข้ามาของผู้ใช้ได้

ปัญหาท้าทาย

- **เราสามารถสร้างตัวแปรขึ้นมาใหม่ได้ 2 วิธี ได้แก่?**

- ค่าของ x หลังจากที่ run มีค่าเท่ากับเท่าไร?: x := 5; x += 1
- ขอบเขตคืออะไร และเราสามารถจะกำหนดขอบเขตของตัวแปรได้อย่างไร?
- var และ const แตกต่างกันอย่างไร?
- 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`)
}
```

แต่กระนั้นตัวโปรแกรมข้างบนทั้งคู่ก็ยังดูเยิ่นเย้อ เราต้องหาทางซักทางมาจัดการงานที่มันดูซ้ำๆกันแล้วล่ะ

5.1 For

ซึ่ง for จะช่วยให้เราจัดการกับชุดคำสั่งแบบเดิมๆที่เกิดขึ้นซ้ำๆ แล้วเราก็จับโปรแกรมก่อนหน้านี้มาอาบน้ำปะแป้งใหม่โดยใช้ for ก็จะได้หน้าตาแบบนี้

```
package main

import "fmt"

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

เป็นไงหล่อขึ้นมั้ย จากตัวอย่างข้างบน เริ่มจากสร้างตัวแปรชื่อ i มาเก็บตัวเลขที่ต้องการจะแสดง ถัดมาก็ใช้ for มาตรวจสอบเงื่อนไขว่าเป็น true หรือ false สุดท้ายก็จะเป็นส่วนภายใต้ปีกกา ที่จะถูกเรียกให้ทำงานถ้าเงื่อนไขนั้นถูกต้อง โดยเจ้าตัว for loop นั้นก็จะทำงานแบบนี้:

i. เริ่มจากประเมินตัว `i <= 10` (i มีค่าน้อยกว่าหรือเท่ากับ 10) ซึ่งถ้าเงื่อนไขเป็นจริงก็จะไปทำงานในส่วนที่อยู่ภายในปีกกา ถ้าไม่ใช่ก็จะกระโดดข้ามไปทำงานต่อในส่วนที่อยู่หลังปีกกา (ในที่นี้ก็จะจบการทำงานไปเลยเพราะไม่มีอะไรต่อจาก for loop แล้ว อิอิ)

ii. หลังจากโปรแกรมทำการสั่งประมวลผล คำสั่งที่อยู่ภายในปีกกาเสร็จแล้ว ก็จะวนกลับไปจุดเริ่มต้นของ for แล้วเริ่มต้นทำข้อ 1 ใหม่

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

จากแบบฝึกหัดนี้ถ้าเราไล่ตัวโปรแกรม โดยคิดซะว่าตัวเองเป็น คอมพิวเตอร์ ก็น่าจะเห็นเหตุการณ์ที่เกิดขึ้นแบบนี้:

- สร้างตัวแปร i โดยมีค่าเป็น 1
- ตรวจสอบค่า i <= 10 โอ๊โอ ยังเป็นจริงอยู่
- พิมพ์ i
- กำหนดค่า i เท่ากับ i + 1 (ตอนนี้ i ก็เท่ากับ 2)
- ตรวจสอบค่า i <= 10 โอ๊โอ ยังเป็นจริงอยู่
- พิมพ์ i
- กำหนดค่า i เท่ากับ i + 1 (ตอนนี้ i ก็เท่ากับ 3)
- ...
- กำหนดค่า i เท่ากับ i + 1 (ตอนนี้ i ก็เท่ากับ 11)
- ตรวจสอบค่า i <= 10 โอ๊โอ 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 ก็จะเห็นได้บ่อยๆ แต่ก็มีท่าพิเศษให้ใช้คือ ++ หรือว่าจะลดค่าลงทีละ 1 ก็มีเหมือนกันใช้ --)

เดี๋ยวเราก็จะว่า for loop จะมีท่าอะไรให้ใช้อีกบ้างในบทถัดๆไป 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 ล่ะ! 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** ก็จะมีความคล้ายคลึง **for** ที่ว่า จะประกอบไปด้วยส่วนที่เป็นเงื่อนไขแล้วตามด้วย บล๊อคภายใต้เครื่องหมายปีกกา แต่จะมี **else** เป็น option เพิ่มเข้ามา ตัวอย่าง ถ้าเงื่อนไขหลัง **if** เป็น **true** บล๊อคที่ตามหลังก็จะถูกสั่งทำงาน ถ้าไม่ใช่ก็จะข้ามไป แต่ถ้ามี **else** อยู่ด้วยก็จะไปสั่งบล๊อคของ **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 บล๊อคที่ตามหลังก็จะถูกสั่งทำงาน ตัวอื่นๆที่เหลือก็จะไม่ถูกเรียก (ตัวอย่าง ให้ค่าตัวแปร i เท่ากับ 8 ซึ่งหารด้วย 4 และ 2 ลงตัว แต่บล๊อค `// divisible by 4` จะไม่ถูกสั่งทำงาน เพราะ บล๊อค `// 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** เป็นจริงก็ให้กระโดดเข้าไปในบล๊อค
- ถ้า เศษที่เหลือจาก **i** ÷ **2** เท่ากับ **0** เป็น เท็จ ให้กระโดดไปที่บล๊อค **else**
- พิมพ์ **i** แล้วตามด้วย **odd**
- เพิ่มค่าให้กับ **i** (ประโยคที่ตามหลังเงื่อนไข **i <= 10**)
- ถ้า **i** น้อยกว่า หรือเท่ากับ **10** เป็นจริงก็ให้กระโดดเข้าไปในบล๊อค

- ถ้า เศษที่เหลือจาก `i` ÷ `2` เท่ากับ `0` เป็น จริง ให้กระโดดไปที่บล๊อค `if`
- พิมพ์ `i` แล้วตามด้วย `even`
- ...

เครื่องหมาย `%` (remainder operator) เหมือนกับว่าได้ตัดขาดจากเราหลังจากจบ ชั้นประถม (จริงเหรอ เค้าสอนกันด้วย? อิอิ) กลายเป็นว่าสำคัญสุดๆเลยเวลาที่เขียนโปรแกรม จะพบเห็นได้บ่อยๆเลย ต้องแต่ตารางข้อมูลที่แบ่งแถบสีตามแนวนอน(zebra striping tables) หรือใช้ในการแบ่งกลุ่มของชุดข้อมูล

5.3 Switch

สมมุติว่าเราอยากเขียนโปรแกรมที่พิมพ์ชื่อตัวเลข โดยอาศัยวิชาก้นหีบที่เรียนผ่านๆมา 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")
}
```

ถ้าเราเขียนออกมาในท่านี้มันก็ออกจะน่าเบื่อไปหน่อย 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 นีแหละ แล้วก็ตามด้วย เอ็กซ์เพรสชั่น (ในที่นี้คือ i) ลำดับสุดท้ายก็เป็นเคสตามแบบต่างๆ โดยค่าของ เอ็กซ์เพรสชั่น ก็จะถูกเปรียบเทียบกับ เอ็กซ์เพรสชั่น ของแต่ละเคส ซึงถ้าเปรียบเทียบแล้วเท่ากัน ชุดคำสั่งที่อยู่หลังเครื่องหมาย **:** ก็จะถูกเรียกใช้งาน

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

พวก control flow statements หลักๆก็จะมีประมาณเท่านี้ ส่วนตัวอื่นๆก็จะมีเสริมเข้ามาในบทถัดไป

ปัญหาท้ายบท

i. โปรแกรมข้างล่างแสดงผลอะไรออกมาจ๊ะ

```
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
- หารคะแนนรวมด้วยจำนวนของสมาชิกเพื่อหาค่าเฉลี่ย

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 โดยปกติเราจะใช้ใช้ชื่อของชนิดข้อมูลเหมือนกับฟังก์ชันเพื่อแปลงชนิดข้อมูล

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)))
```

_ ถูกใช้เป็นชื่อตัวแปรเพื่อบอกคอมไพเลอร์ว่าเราไม่ต้องการใช้ตัวแปรนั้น

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. สิ่งนี้จำเป็นในภาษาโกและมันทำให้เราสามารถลบสมาชิกออกจากอาเรย์ได้ง่ายโดยการคอมเม้นต์บรรทัดนั้นออกไป:

```
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 ฟังก์ชันสามารถส่งอากิวเมนต์ลำดับที่สามได้:

```
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 (อากิวเมนต์ตัวแรก) และเชื่อมต่ออากิวเมนต์ที่เหลือกับสไลซ์นั้น

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 หรือพจนานุกรม maps ถูกใช้เพื่อค้นหาค่าโดยใช้คีย์เชื่อมโยงของมันนี่คือตัวอย่างของแมปในภาษาโก:

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

ขนิดข้อมูลแมปถูกแทนด้วยคีย์เวิร์ด 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
```

ก่อนหน้าเราเห็นเพียงแค่ข้อผิดพลาดจากการคอมไพล์ นี่เป็นตัวอย่างของข้อผิดพลาดจากกันรัน มันเกิดขึ้นระหว่างที่รันโปรแกรม ขณะที่ข้อผิดพลาดจากการคอมไพล์เกิดขึ้นขณะที่ทดลองคอมไพล์โปรแกรม

สาเหตุของปัญหานี้เกิดจากแมปต้องมีการกำหนดค่าเริ่มต้นก่อนที่พวกมันจะถูกใช้งาน เราควรเขียนในลักษณะนี้:

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

ถ้าคุณรันโปรแกรมนี้ คุณควรจะเห็นผลลัพธ์เป็น 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 อย่างที่สองคีย์ของแมปไม่มีการเรียงลำดับ 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 ถ้าค่าที่เชื่อมโยงกับคีย์นั้นมีอยู่จริงจะรันโค้ดที่อยู่ภายในบล็อค

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` ตอนนี้เรามีแมปของสตริงไปยังแมปของสตริงไปยังสติง แมปที่อยู่ด้านนอกถูกใช้เป็นตารางค้นหาโดยใช้สัญลักษณ์ของธาตุ ขณะที่แมปที่อยู่ด้านในถูกใช้เพื่อเก็บข้อมูลทั่วไปของธาตุนั้น ถึงแม้ว่าแมปจะถูกใช้ในลักษณะนี้อยู่บ่อย ๆ ในบทที่ 9 เราจะเห็นว่ามีวิธีที่ดีกว่านี้เพื่อใช้จัดเก็บข้อมูลเชิงโครงสร้าง

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,
}
```