



Golang

Une introduction au langage Go

Pourquoi croire en Go ?

- En 2009, Google dévoile un langage de programmation dont le début de la conception remonte à 2007
- À la tête de ce projet, Robert Griesemer, Rob Pike, et Ken Thompson, qui continue d'innover sans relâche à 73 ans, après nous avoir déjà tant offert
- Malheureusement, si la nouvelle fait grand bruit, peu de projets adoptent ce nouveau venu, nommé **Go** mais plus souvent désigné par **Golang**, pour éviter toute confusion
- Sept ans après ses débuts officiels, je crois plus que jamais que Golang apporte des solutions aux problématiques actuelles.
- Voici pourquoi

Pourquoi croire en Go ?

- Golang était-il trop en avance sur son temps ?
- C'est bien possible.
- Conçu pour gérer des datacenters, consommer peu de ressources, exploiter au mieux les CPU modernes à cœurs multiples ou encore faciliter la lecture du code, il répondait à des problématiques qui n'était pas forcément encore au cœur de nos préoccupations il n'y a pas si longtemps.
- Quand on analyse la situation, on réalise que l'heure est venue de réévaluer les atouts de Golang, trop longtemps ignoré par beaucoup.
- Car chaque ère technologique a catalysé le développement d'un ou plusieurs langages : les gros systèmes ont vu naître le Cobol, Unix, le C, Windows, les L4G. Sans parler du Web qui aura fait exploser PHP, Java, Python, Ruby, JavaScript...
- Les mutations engendrées par le cloud vont forcément provoquer l'apparition sur le devant de la scène d'un ou plusieurs langages.

Pourquoi croire en Go ?

- J'attends depuis des années l'arrivée de ces derniers, capables de nous permettre de développer efficacement sur les nouvelles infrastructures.
- Et quand on regarde les spécificités du Go, on se dit qu'on a peut-être déjà la solution à portée de la main :
 - il est à la base de tous les outils de l'ère du Cloud : Docker, Kubernetes, etc.
 - il est simple, mais offre nativement tout ce qu'il faut pour développer une plateforme de services.
 - il intègre nativement la concurrence d'accès.

Pourquoi croire en Go ?

- il permet de déployer des conteneurs légers... vraiment légers ! À quoi ça sert d'avoir des conteneurs légers s'il faut booter des machines virtuelles (JVM, CLR, V8) énormes en occupation mémoire dedans ?
- il est idéal pour construire des micro services.
- il supporte nativement JSON et HTTP (client et serveur).
- il est poussé par une communauté bienveillante.
- il va être largement adopté par les universités pour enseigner la programmation concurrente. Dans quelques années, tous les étudiants auront dans leur bibliothèque le nouveau Kernighan, qui après avoir écrit le bestseller « The C Programming Language », vient de sortir « The Go Programming Language ».

Pourquoi croire en Go ?

- Cette liste de points cruciaux démontre que Golang est totalement en adéquation avec nos préoccupations de 2017.
- Il est capable de résoudre un nombre de problématiques incroyable, des problématiques dont nous n'avons pas forcément conscience en 2009.
- Est-il la solution parfaite ? Je ne suis pas loin de le penser.
- Mais une chose est certaine : il est impossible d'imaginer que les langages créés avec les contraintes technologiques d'il y a 20 ans soient capables de répondre aux défis d'aujourd'hui.
- Il est plus que temps de donner sa chance au Go !

Golang ?

- Go est un langage compilé
- Inspiré de C et Pascal
- Développé par Google (Robert Griesemer, Rob Pike et Ken Thompson)
- Rob Pike à propos des jeunes développeurs :

« Ils ne sont pas capables de comprendre un langage brillant, mais nous voulons les amener à réaliser de bons programmes. Ainsi, le langage que nous leur donnons doit être facile à comprendre et facile à adopter »
- Go veut faciliter et accélérer la programmation à grande échelle : en raison de sa simplicité, sa compilation serait de 80 % à 90 % plus rapide que la compilation classique du C, et il est donc concevable de l'utiliser aussi bien pour écrire des applications, des scripts ou de grands systèmes. Cette simplicité est nécessaire aussi pour assurer la maintenance et l'évolution des programmes sur plusieurs générations de développeurs.

Golang ?

- S'il vise aussi la rapidité d'exécution, indispensable à la programmation système, il considère le multithreading comme le moyen le plus robuste d'assurer sur les processeurs actuels cette rapidité tout en rendant la maintenance facile par séparation de tâches simples exécutées indépendamment afin d'éviter de créer des « usines à gaz ».
- Cette conception permet également le fonctionnement sans réécriture sur des architectures multi-cœurs en exploitant immédiatement l'augmentation de puissance correspondante

Caractéristiques

- Le langage Go a été créé pour la programmation système et a depuis été étendu aux applications, ce qui constitue la même cible que le C et surtout le C++
- Il s'agit d'un langage impératif et concurrent

Caractéristiques : Concurrency

- Le langage Go a été créé pour la programmation système et a depuis été étendu aux applications, ce qui constitue la même cible que le C et surtout le C++. Il s'agit d'un langage impératif et concurrent
- Go intègre directement les traitements de code en concurrence (goroutine)
- Le programme prendra alors avantage de la topologie de l'ordinateur pour exécuter au mieux les goroutines, pas forcément dans un nouveau thread, mais il est aussi possible qu'un groupe de goroutines soit multiplexé sur un groupe de threads

Caractéristiques : Concurrency

- Pour appeler une fonction **f**, on écrit **f()**
- Pour l'appeler en tant que goroutine, on écrit simplement **go f()**, ce qui est très semblable au *call f task;* de PL/I ; langage gérant également le multitâche depuis 1970
- Les goroutines communiquent entre elles par passage de messages, en envoyant ou en recevant des messages sur des canaux
- Ces messages synchronisent les goroutines entre elles, conformément au modèle CSP, considéré par les auteurs comme plus intuitif que le modèle multithreads (avec synchronisation par sémaphores comportant des verrous, notion introduite aussi elle-même par Dijkstra)

Caractéristiques : Système de types

- Go a un système de type statique, fortement typé, structurel et sûr, fondé sur l'inférence de types avec la possibilité d'utiliser un typage explicite
- La compatibilité des types composés est fondée sur les propriétés plutôt que sur le nom. C'est-à-dire que deux types composés seront équivalents si leurs propriétés sont équivalentes : même nom pour la propriété et équivalence de type
- C'est le typage structurel

Caractéristiques : Système de types

- Cela a pour conséquence que le langage n'est pas objet au sens classique (soit avec classes, soit avec prototype), cependant les concepteurs du langage ont fait un choix plus original pour un langage statique
- Il est possible de définir des interfaces portant des méthodes décrivant le comportement d'un objet (Il est aussi facilement possible de mélanger plusieurs interfaces en une seule)
- Les fonctions Go peuvent déclarer accepter un argument de cette interface. Un objet déclarant toutes les méthodes de cette interface, avec la même signature, peut être passé en argument de cette méthode
- La vérification du type est effectuée statiquement par le compilateur

Caractéristiques : Système de types

- Le fait que Go ne soit pas objet au sens classique fait que Go n'a pas d'héritage de type et pas de sous-classage
- Ceci permet de contourner les problèmes posés par ces systèmes tels l'héritage multiple dans les langages qui le permettent (en C++ par exemple), ou l'héritage simple (en Java par exemple)
- Grâce à l'équivalence de types fondée sur les propriétés, Go n'a pas besoin d'héritage de type
- Le sous-classage est émulé par l'« embarquement de type ». Ceci permet de mélanger facilement deux bases de code conçues indépendamment, sans qu'elles aient besoin de partager des types communs.

Caractéristiques : Système de types

- La visibilité des structures, attributs, variables, constantes, méthodes, types de haut niveau et des fonctions hors de leur paquetage de déclaration est définie par la casse du premier caractère de leurs identificateurs.

Caractéristiques : Divers

- Dans Go, la gestion de la mémoire est laissée à un ramasse-miettes
- Il n'y a pas encore de programmation générique même si les concepteurs du langage y réfléchissent
- Il n'y a pas de surcharge de méthodes ou d'arithmétique des pointeurs.
- Enfin, il n'y a pas d'assertions ou d'exceptions
- Pour remplacer ces deux derniers, Go fournit les mots clés `defer`, `panic` et `recover`¹³ qui donnent des mécanismes similaires aux systèmes de gestion des exceptions de langages tels que C++ et Java (mots clés `try`, `catch`, `finally` et `throw`).

Caractéristiques : Divers

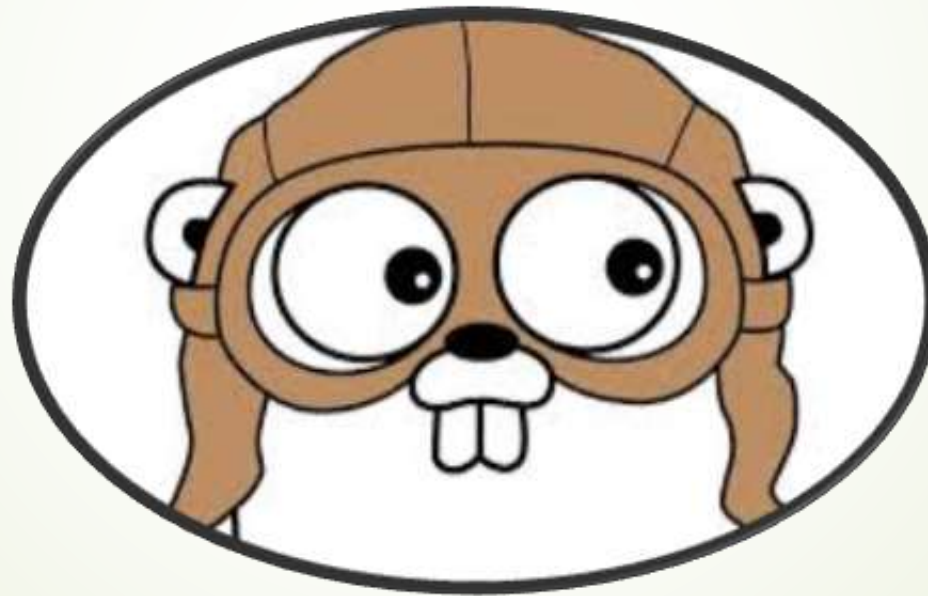
- Go peut s'interfacer avec des bibliothèques en C/C++, des développeurs tiers ayant déjà développé des bindings pour SDL et MySQL
- Go définit un format de code standard (au niveau des indentations, et de la présentation des structures de contrôle) et fournit un outil pour l'appliquer (`go fmt`)
- Go propose également un système de documentation à partir du code et un framework de test
- L'unité de compilation de go est le package qui est représenté dans l'implémentation standard par un répertoire et les fichiers directement contenus dans ce répertoire

Caractéristiques : Divers


- L'import d'un package se fait par son chemin d'importation et peut préciser soit une bibliothèque standard, soit également des packages tiers installés dans des dépôts de sources distants (actuellement supporté : dépôt sous svn, git, mercurial et bazaar)

Golang

➤ Welcome to



Hello, Go!



```
hello.go x
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      fmt.Println("Hello Go!")
9  }
10
```


Hello, Go!

A screenshot of a code editor window titled 'hello.go'. The code is as follows:

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Println("Hello Go!")
9 }
10
```

- Tous les programmes Go commencent par une déclaration de package
- Deux types de programmes : **exécutables** ou **bibliothèques**
- **import** permet d'inclure le code provenant d'un autre package
- Les lignes commençant par `//` sont de commentaires sur une ligne
// One line comment
- Les commentaires multi lignes sont matérialisés par `/* */`
/*
Block comment
***/**

Hello, Go!

A screenshot of a code editor window titled 'hello.go'. The code is as follows:

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Println("Hello Go!")
9 }
10
```

- Tous les exécutables ont la déclaration **package main**
- La fonction **main** est le point d'entrée de tous les exécutables
- Les fonctions importées sont préfixées du nom du package d'origine :
 - **fmt.Println(...)**

Types de base

- Strings
 - *string*
- Integers
 - *int8, uint8, int16, uint16, int32, uint32, int64, uint64*
- Floats
 - *float32, float64*
- Booleans
 - *true*
 - *false*
- Complex
 - *complex*

Strings

- Une chaîne de caractères est délimitée par :
 - Soit `" "` pour une chaîne contenant des caractères d'échappement
 - *`"Ceci est une chaîne de caractères"`*
 - Soit `` ``
 - *``Ma chaîne de caractères contient des "'" et des "` et ↵ peut contenir des retours à la ligne``*
- Des opérations sont possibles sur les chaînes de caractères :
 - *`len("Hello, Go!")`* équivalent à 11
 - *`"Hello, Go!"[2]`* équivalent à la lettre L minuscule

Opérateurs

- Addition

- +

- Soustraction

- -

- Multiplication

- *

- Division

- /

- Reste (modulo)

- %

- Egalité :

- ==

- Non :

- !

- Incrémentation

- ++

- **Seulement post-incrémentation**

- **Equivalent à : += 1**

Déclaration des variables

- Mot-clé: **var** (optionnel mais bonne pratique)
- Déclarations possibles :
 - **var** <nomDeVariable> <type>
 - **var** <nomDeVariable> <type> = <valeur>
 - <nomDeVariable> := <valeur> *(le type de la variable dépend de la valeur)*
- Exemples :
 - **var** a **int8**
 - **var** b **string** = "A string of characters "
 - c := 112

Portée des variables

- Les variables sont "visibles" au sein du bloc de code dans lequel elles sont déclarées
- `string01` est visible dans les deux fonctions déclarées
- `string02` n'est visible que dans la fonction `main()`
- La ligne 22 provoque une erreur à la compilation

```
cmd\scope\scope.go:22:14: undefined: string02
```

```
scope.go x
1  package main
2
3  import (
4      "fmt"
5  )
6
7  var string01 = "string01"
8
9  func main() {
10
11      var string02 = "string02"
12      fmt.Println(string01)
13      fmt.Println(string02)
14
15      f()
16
17  }
18
19  func f() {
20
21      fmt.Println(string01)
22      fmt.Println(string02)
23
24  }
25
```

Constantes

- Go supporte les constantes
- **const** remplace **var**
- Variables ne pouvant être modifiées après leur création
- La ligne 13 provoque une erreur à la compilation

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8
9     const myConstant float64 = 3.14159
10    fmt.Printf("myConstant= %f", myConstant)
11
12    // This line will fail at compilation time
13    myConstant = 3.14
14
15 }
```

```
cmd\constants\constants.go:13:13: cannot assign to myConstant
```

Les fonctions "Print"...

- ***fmt.Print(...)*** affiche vers la console sans caractère fin de ligne
- ***fmt.Println(...)*** affiche vers la console et ajoute un caractère fin de ligne
- ***fmt.Printf(...)*** affiche vers la console, sans caractère fin de ligne et accepte une chaîne de format
- Ces fonctions retournent le nombre d'octets écrits et un code d'erreur

```
prints.go x
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      // Prints all the arguments to console
10     // No space between arguments
11     // No newline at the end
12     fmt.Print("Thierry", "DECKER", "\n")
13
14     // Prints all the arguments to console
15     // Space between arguments
16     // Newline at the end
17     fmt.Println("Thierry", "DECKER")
18
19     // Prints the formatted string to the console
20     // No newline at the end
21     fmt.Printf("Thierry %s\n", "DECKER")
22
23     // Bytes printed and error code returned
24     n, status := fmt.Printf("Thierry %s\n", "DECKER")
25     fmt.Printf("%d bytes printed, error code %v returned", n, status)
26
27 }
28
```

Les fonctions "Sprint"...

- Ces fonctions créent des chaînes de caractères
- `fmt.Sprint(...)`
- `fmt.Sprintln(...)`
- `fmt.Sprintf(...)`
- Ces fonctions retournent une chaîne de caractères

```
sprints.go x
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      // Sends all the arguments to a string
10     // No space between arguments
11     // No newline at the end
12     var a string = fmt.Sprint("Thierry", "DECKER", "\n")
13     fmt.Print(a)
14
15     // Sends all the arguments to a string
16     // Space between arguments
17     // Newline at the end
18     var b string = fmt.Sprintln("Thierry", "DECKER")
19     fmt.Print(b)
20
21     // Sends the formatted string to a string
22     // No newline at the end
23     var c string = fmt.Sprintf("Thierry %s\n", "DECKER")
24     fmt.Print(c)
25
26 }
27
```


Les fonctions "Fprint"...

- Ces fonctions écrivent dans des fichiers
- *fmt.Sprint(...)*
- *fmt.Sprintln(...)*
- *fmt.Sprintf(...)*
- Ces fonctions retournent le nombre d'octets écrits et un code d'erreur

```
fprints.go x
1  package main
2
3  import (
4      "os"
5      "bufio"
6      "fmt"
7  )
8
9  func main() {
10
11      // Create a file and use bufio.NewWriter.
12      f, _ := os.Create("cmd/fprints/file.txt")
13      w := bufio.NewWriter(f)
14
15      // Use Fprint to write things to the file.
16      // ... No trailing newline is inserted.
17      fmt.Fprint(w, "Hello", "\n")
18      fmt.Fprint(w, 123, "\n")
19      fmt.Fprint(w, "... \n")
20
21      // Use Fprintf to write formatted data to the file.
22      value1 := "cat"
23      value2 := 900
24      fmt.Fprintf(w, "%v %d... \n", value1, value2)
25
26      fmt.Fprintln(w, "DONE...")
27
28      // Done.
29      w.Flush()
30
31 }
```

Les boucles

- Go ne possède qu'un type de boucle: **for**
- **For <init> ; <end condition> ; <post treatment> {}**
- <init> et <post treatment> sont optionnels

```
loops.go
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      // Classical for loop
10     for i := 0; i < 10; i++ {
11         fmt.Printf("i= %d\n", i)
12     }
13
14     // While loop
15     j := 0
16     for j < 10 {
17         fmt.Printf("j= %d\n", j)
18         j++
19     }
20
21 }
22
```


Les conditions

- Comme pour les boucles, l'expression de la condition n'est pas obligatoirement entourée de `()` mais les `{}` le sont
- La clause **else** est facultative
- Les variables déclarées dans le bloc **if** ne sont pas visibles en dehors de ce bloc

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     for i := 0; i < 10; i++ {
9         if i%2 == 0 {
10             fmt.Printf("%d is even and i*i = %d", i, i*i)
11         } else {
12             fmt.Printf("%d is odd", i)
13         }
14         if i > 5 {
15             fmt.Printf(". (%d is greater than 5)", i)
16         }
17         fmt.Println()
18     }
19 }
20
```

Les switches

- Chaque **case** est évalué à la suite
- Seul le premier **case** vérifié est exécuté
- Le case **default** est optionnel et le dernier évalué
- **Pas de break**
- Les **case** ne sont pas forcément constants et les valeurs testées ne sont obligatoirement des entiers

```
switches.go x
1 package main
2
3 import (
4     "fmt"
5     "runtime"
6 )
7
8 func main() {
9     for i := 10; i < 20; i++ {
10
11         switch i%2 == 0 {
12             case true:
13                 fmt.Printf("%d is even\n", i)
14             default:
15                 fmt.Printf("%d is odd\n", i)
16         }
17     }
18
19     fmt.Print("Go runs on ")
20     switch os := runtime.GOOS; os {
21         case "darwin":
22             fmt.Println("OS X.")
23         case "linux":
24             fmt.Println("Linux.")
25         default:
26             fmt.Printf("%s.", os)
27     }
28     fmt.Println()
29
30     switch {
31         case runtime.GOOS == "windows":
32             fmt.Print("Go runs on windows again!\n")
33     }
34 }
35
36
37
```

Les arrays

- Tableau d'un nombre fixe d'éléments de même type
- Le premier élément à la position 0
- La fonction `len()` retourne le nombre d'éléments du tableau
- La fonction `range` permet d'itérer sur les éléments de l'array dans cet exemple

```
arrays.go x
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8
9     var array01 [5]int
10    array01[4] = 25
11    fmt.Printf("array01 is          : %v\n", array01)
12    fmt.Printf("The element at position 4 of array01: %d\n", array01[4])
13    fmt.Printf("\n")
14
15    var array02 [5]float64
16    var average01 float64
17    var average02 float64
18
19    array02[0] = 98
20    array02[1] = 93
21    array02[2] = 77
22    array02[3] = 82
23    array02[4] = 83
24
25    for i := 0; i < len(array02); i++ {
26        average01 += array02[i]
27    }
28
29    for _, valueAti := range array02 {
30        average02 += valueAti
31    }
32    average01 = average01 / float64(len(array02))
33    average02 = average02 / float64(len(array02))
34
35    fmt.Printf("array02 is          : %v\n", array02)
36    fmt.Printf("Length of array02 is      : %d\n", len(array02))
37    fmt.Printf("The average of array02 elements is : %f\n", average01)
38    fmt.Printf("The average of array02 elements is : %f\n", average02)
39
40 }
41
```

Les ranges

- **range** permet d'itérer sur les éléments d'un objets
- Retourne l'**index** et la **valeur** correspondante de chacun des éléments itérés
- **_** dans la boucle **for** permet de ne pas utiliser la valeur d'index retournée par range. L'ignorer simplement provoquerait une erreur de compilation

```

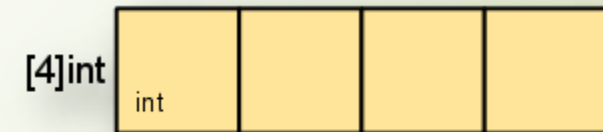
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      array01 := [10]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
10     array02 := [3][3]int{{11, 12, 13}, {21, 22, 23}, {31, 32, 33},}
11     string01 := "Welcome to Go!"
12
13     fmt.Printf("array01: %v\n", array01)
14     fmt.Printf("array02: %v\n", array02)
15
16     for position, value := range array01 {
17         fmt.Printf("Value at position %d is %d\n", position, value)
18     }
19
20     for _, value := range array01 {
21         fmt.Printf("Next value of array01 is %d\n", value)
22     }
23
24     for position, value01 := range array02 {
25         fmt.Printf("Value at position %d is %d\n", position, value01)
26         for position, value02 := range value01 {
27             fmt.Printf("Value at position %d is %d\n", position, value02)
28         }
29     }
30
31     for _, value01 := range array02 {
32         fmt.Printf("Next value of array01 is %d\n", value01)
33         for _, value02 := range value01 {
34             fmt.Printf("Next value of array01 is %d\n", value02)
35         }
36     }
37
38     for _, letter := range string01 {
39         fmt.Printf("Next letter of string01 is %c\n", letter)
40     }
41
42 }
43

```


Slices

- Les **slices** sont basées sur les **arrays**
- Les **arrays** sont des valeurs, pas un pointeur vers le premier élément (comme en C)
- Les **arrays** sont peu flexibles
- Les **slices** n'ont pas de spécification de longueur
- Elles sont déclarées comme des arrays mais sans spécifier le nombre d'éléments
- Leur taille est gérée dynamiquement

```
slices.go
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      // Arrays
10     a := [2]int64{1, 2}
11     fmt.Printf("a: %v, type of a: %T\n", a, a)
12
13     b := [...]int32{1, 2, 3, 4}
14     fmt.Printf("b: %v, type of b: %T\n", b, b)
15
16 }
17
```



Slices

- ▶ Elles peuvent être créées à l'aide de la fonction **make**
- ▶ **make** prend en entrée :
 - ▶ Une array
 - ▶ Une longueur
 - ▶ Une capacité
- ▶ Lorsque la capacité est omise, elle a par défaut la valeur de la longueur
- ▶ Une slice peut être aussi créée à partir d'une array ou d'une autre slice

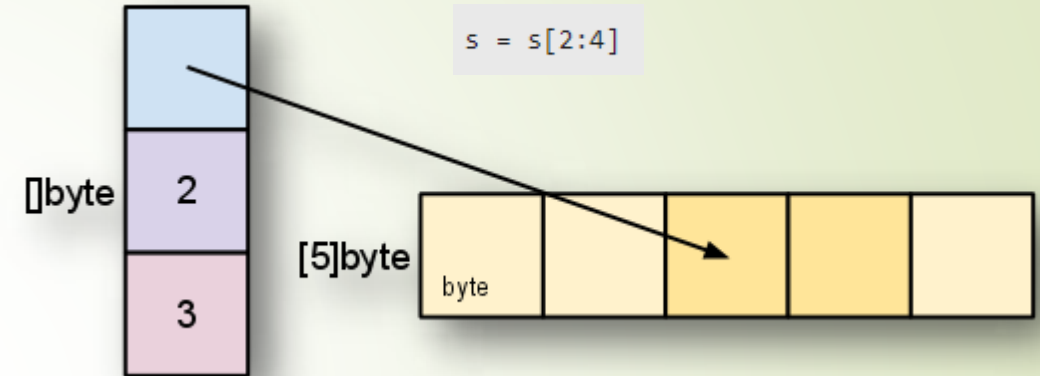
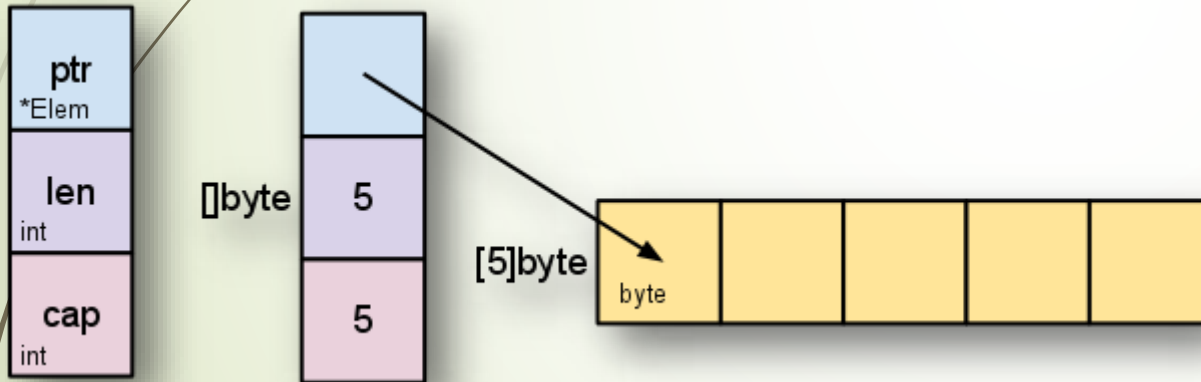
```
func make([]T, len, cap) []T
```

```
var s []byte  
s = make([]byte, 5, 5)  
// s == []byte{0, 0, 0, 0, 0}
```

```
s := make([]byte, 5)
```


Slices

- Une slice est le descripteur d'un segment d'une array, constitué d'un pointeur vers l'array, d'une longueur de segment et d'une capacité



Slices

- Une opération de slice ne copie pas les données d'origine (c'est ce qui les rend efficaces) !
- Modifier les éléments d'origine affecte la slice
- Une slice ne peut être agrandie au delà de sa capacité.
- De même, un slice ne peut être redimensionné en dessous de zéro pour accéder aux éléments précédents

```
d := []byte{'r', 'o', 'a', 'd'}  
e := d[2:]  
// e == []byte{'a', 'd'}  
e[1] = 'm'  
// e == []byte{'a', 'm'}  
// d == []byte{'r', 'o', 'a', 'm'}
```

Slices

```

1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      var01 := [10]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9,}
10     var02 := var01[2:5]
11
12     fmt.Printf("var01 = %v, length; %d, capacity: %d\n", var01, len(var01), cap(var01))
13     fmt.Printf("var02 = %v, length; %d, capacity: %d\n", var02, len(var02), cap(var02))
14     fmt.Printf("\n")
15
16     var02[1] *= 10
17     fmt.Printf("var01 = %v, length; %d, capacity: %d\n", var01, len(var01), cap(var01))
18     fmt.Printf("var02 = %v, length; %d, capacity: %d\n", var02, len(var02), cap(var02))
19     fmt.Printf("\n")
20
21     var02 = var02[:cap(var02)]
22     fmt.Printf("var01 = %v, length; %d, capacity: %d\n", var01, len(var01), cap(var01))
23     fmt.Printf("var02 = %v, length; %d, capacity: %d\n", var02, len(var02), cap(var02))
24     fmt.Printf("\n")
25
26     var02 = var01[:]
27     fmt.Printf("var01 = %v, length; %d, capacity: %d\n", var01, len(var01), cap(var01))
28     fmt.Printf("var02 = %v, length; %d, capacity: %d\n", var02, len(var02), cap(var02))
29     fmt.Printf("\n")
30 }
31
32

```

```

var01 = [0 1 2 3 4 5 6 7 8 9], length; 10, capacity: 10
var02 = [2 3 4], length; 3, capacity: 8

var01 = [0 1 2 30 4 5 6 7 8 9], length; 10, capacity: 10
var02 = [2 30 4], length; 3, capacity: 8

var01 = [0 1 2 30 4 5 6 7 8 9], length; 10, capacity: 10
var02 = [2 30 4 5 6 7 8 9], length; 8, capacity: 8

var01 = [0 1 2 30 4 5 6 7 8 9], length; 10, capacity: 10
var02 = [0 1 2 30 4 5 6 7 8 9], length; 10, capacity: 10

```

Slices (growing, copying)

- Pour agrandir la capacité d'une slice, on doit en créer une plus grande et copier le contenu d'origine dans la nouvelle slice

```
slices-grow.go
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      slice01 := make([]byte, 1, 1)
10     slice01[0] = 1
11     fmt.Printf("slice01: %v, length: %d, capacity: %d\n", slice01, len(slice01), cap(slice01))
12
13     slice02 := make([]byte, len(slice01), cap(slice01)*2)
14     for i := range slice01 {
15         slice02[i] = slice01[i]
16     }
17     slice01 = slice02
18     fmt.Printf("slice02: %v, length: %d, capacity: %d\n", slice02, len(slice02), cap(slice02))
19
20     slice03 := make([]byte, len(slice02), cap(slice02)*2)
21     copy(slice03, slice02)
22     fmt.Printf("slice03: %v, length: %d, capacity: %d\n", slice03, len(slice03), cap(slice03))
23
24 }
25
```

```
slice01: [1], length: 1, capacity: 1
slice02: [1], length: 1, capacity: 2
slice03: [1], length: 1, capacity: 4
```

Slices (appending)

- Ajouter des éléments à une slice peut se faire à l'aide de la fonction **append** qui gère l'augmentation de la taille dynamiquement

```
slices-append.go
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     a := []int16{1, 2,}
8     b := []int16{10, 20, 30,}
9     c := []int16{100, 200, 300, 400}
10
11     d := make([]int16, 0)
12
13     d = append(d, 0)
14     fmt.Printf("d: %v, len(d): %d, cap(d): %d\n", d, len(d), cap(d))
15
16     d = append(d, a...)
17     fmt.Printf("d: %v, len(d): %d, cap(d): %d\n", d, len(d), cap(d))
18
19     d = append(d, b...)
20     fmt.Printf("d: %v, len(d): %d, cap(d): %d\n", d, len(d), cap(d))
21
22     d = append(d, c...)
23     fmt.Printf("d: %v, len(d): %d, cap(d): %d\n", d, len(d), cap(d))
24 }
25
```

```
d: [0], len(d): 1, cap(d): 4
d: [0 1 2], len(d): 3, cap(d): 4
d: [0 1 2 10 20 30], len(d): 6, cap(d): 8
d: [0 1 2 10 20 30 100 200 300 400], len(d): 10, cap(d): 16
```


Maps

- Les maps sont des ensembles non ordonnés de paires "Clé-Valeur"
- Souvent appelées tableaux associatifs ou dictionnaires

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

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

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


Maps

```
maps.go x
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8
9     elements := make(map[string]string)
10
11     elements["H"] = "Hydrogen"
12     elements["He"] = "Helium"
13     elements["Li"] = "Lithium"
14     elements["Be"] = "Beryllium"
15     elements["B"] = "Boron"
16     elements["C"] = "Carbon"
17     elements["N"] = "Nitrogen"
18     elements["O"] = "Oxygen"
19     elements["F"] = "Fluorine"
20     elements["Ne"] = "Neon"
21
22     // Print elements
23     fmt.Printf("%v\n", elements)
24     // Iterate over elements
25     for key, value := range elements {
26         fmt.Printf("Key: %v, value: %v\n", key, value)
27     }
28     // Find the value of a key
29     var key string
30     key = "N"
31     if name, ok := elements[key]; ok {
32         fmt.Printf("Key: %v, value: %v\n", key, name)
33     } else {
34         fmt.Printf("Key: %v was not found\n", key)
35     }
36     key = "Z"
37     if name, ok := elements[key]; ok {
38         fmt.Printf("Key: %v, value: %v\n", key, name)
39     } else {
40         fmt.Printf("Key: %v was not found\n", key)
41     }
42 }
43
```

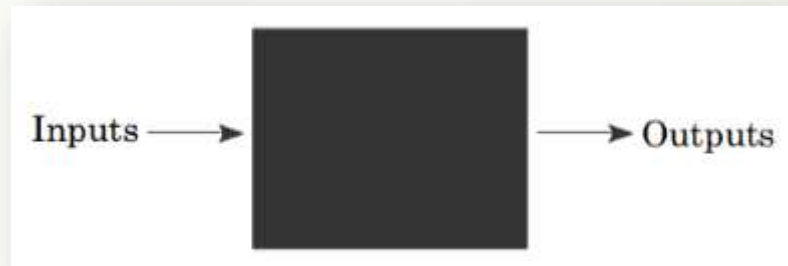
```
Key: Ne, value: Neon
Key: H, value: Hydrogen
Key: He, value: Helium
Key: Li, value: Lithium
Key: N, value: Nitrogen
Key: O, value: Oxygen
Key: F, value: Fluorine
Key: Be, value: Beryllium
Key: B, value: Boron
Key: C, value: Carbon
Key: N, value: Nitrogen
Key: Z was not found
```

Maps of maps

```
maps-of-maps.go
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      vehicules := map[string]map[string]int{
9          "Car": {
10             "Wheels": 4,
11             "Seats": 5,
12             "Airbags": 7,
13          },
14          "Bus": {
15             "Wheels": 4,
16             "Seats": 30,
17          },
18          "Bike": {
19             "Wheels": 2,
20             "Seats": 2,
21          },
22      }
23
24      for vehicle := range vehicules {
25          characteristics := vehicules[vehicle]
26          fmt.Printf("%v, ", vehicle)
27          for characteristic := range characteristics {
28              fmt.Printf("%v: %v, ", characteristic, characteristics[characteristic])
29          }
30          fmt.Printf("\b\b\n")
31      }
32  }
```

```
Car, Wheels: 4, Seats: 5, Airbags: 7
Bus, Seats: 30, Wheels: 4
Bike, Wheels: 2, Seats: 2
```

Fonctions



- Portion indépendante de code possédant zéro ou plusieurs paramètres d'entrée et zéro ou plusieurs paramètres de sortie (retour)
- L'ensemble des entrées et des sortie est appelé signature de la fonction

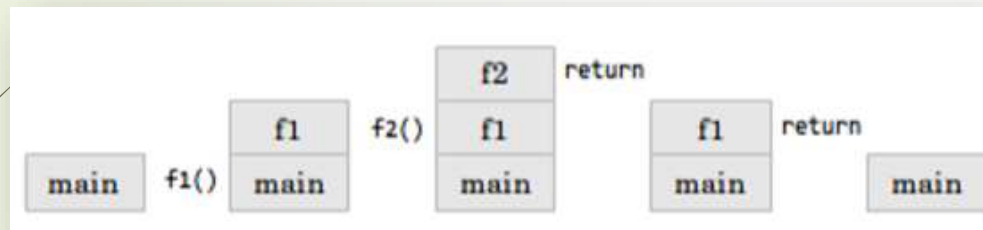
Fonctions

- Une fonction débute par le mot-clé **func**
- Suivi du nom de la fonction et de ses entrées (entre parenthèses)
- Suivi du type des sorties
- Se termine par **return**, suivi des variables retournées
- Le corps de la fonction est inséré entre des accolades
- Le nom des paramètres peut être différents en la fonction appelante et la fonction appelée
- Seules les valeurs sont transmises
- La fonction appelée n'a pas accès aux variables de la fonction appelante

```
functions-01.go x
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func average(elements []float64) float64 {
8      total := 0.0
9      for _, v := range elements {
10         total += v
11     }
12     return total / float64(len(elements))
13 }
14
15 func main() {
16     elementsToAdd := []float64{10,20,11,12,35}
17     fmt.Println(average(elementsToAdd))
18 }
19
20
```


Fonctions

- Les fonctions appelées sont insérées au sommet de la pile (stack)



```

Entering main()
Calling f1()
Entering f1()
Executing f1()
Calling f2()
Entering f2()
Executing f2()
Exiting f2()
Exiting f1()
Exiting main()

```

```

functions-02.go x
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func f1() {
8      fmt.Printf("Entering f1()\n")
9      fmt.Printf("Executing f1()\n")
10     fmt.Printf("Calling f2()\n")
11     f2()
12     fmt.Printf("Exiting f1()\n")
13 }
14
15 func f2() {
16     fmt.Printf("Entering f2()\n")
17     fmt.Printf("Executing f2()\n")
18     fmt.Printf("Exiting f2()\n")
19 }
20
21 func main() {
22     fmt.Printf("Entering main()\n")
23     fmt.Printf("Calling f1()\n")
24     f1()
25     fmt.Printf("Exiting main()\n")
26 }
27

```

Fonctions

- Formats possibles d'appels de fonctions

```
Functions-03.go
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func fa() {
8      fmt.Printf("No input, no output, just printing this message\n")
9  }
10
11 func fb(a int16, b int16) {
12     fmt.Printf("Only inputs, Sum equal %d\n", a+b)
13 }
14
15 func fc() int16 {
16     fmt.Printf("Only output... ")
17     return 100
18 }
19
20 func fd() (int16, string) {
21     fmt.Printf("Only outputs... ")
22     return 100, "This is a string"
23 }
24
25 func fe(a int32, b int32) (mult int32) {
26     mult = a * b
27     fmt.Printf("Named returns... ")
28     return
29 }
30
31 func main() {
32     fa()
33     fb(1, 2)
34     fmt.Printf("%d\n", fc())
35     a, b := fd()
36     fmt.Printf("%d, %s\n", a, b)
37     c := fe(100, 10)
38     fmt.Printf("%d\n", c)
39 }
40
```


Fonctions variadiques

- Fonctions acceptant un nombre variable d'arguments en entrée
- L'opérateur `...` est utilisé pour passer et recevoir les paramètres

```
functions-variadic.go
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Printf("Total= %d\n", summarize(1, 2, 3))
9     fmt.Printf("Total= %d\n", summarize(1, 2, 3, 4, 5))
10    numbers := []int64{10, 20, 30}
11    fmt.Printf("Total= %d\n", summarize(numbers...))
12 }
13
14 func summarize(numbers ...int64) (total int64) {
15     for _, number := range numbers {
16         total += number
17     }
18     return
19 }
20
```

```
Total= 6
Total= 15
Total= 60
```

Closures

- Il est possible de créer une fonction à l'intérieur d'une fonction
- *add* est une variable locale qui a pour signature *func(int, int) int*
- Deux *int* en entrée et un *int* en sortie
- Une fonction ainsi définie à accès aux autres variables locales

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

Closures

- **increment** ajoute 1 à la variable **x** qui est définie dans le scope de la fonction main
- La variable **x** peut être accédée et modifiée par la fonction **increment**
- C'est ainsi qu'au premier appel, 1 est affiché et qu'au second appel, 2 est affiché
- **increment** et **x** forment se que l'on appelle une **closure**

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

Closures

- Une manière d'utiliser une closure est d'écrire une fonction qui retourne une fonction qui, lorsqu'elle est appelée, peut générer une séquence de nombres
- *i* est une variable locale de **makeEvenGenerator** qui retourne elle-même une fonction (dite anonyme) ayant accès à *i*

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func makeEvenGenerator() func() int {
8     i := 0
9     return func() (ret int) {
10         ret = i
11         i += 2
12         return
13     }
14 }
15
16 func main() {
17     nextEven := makeEvenGenerator()
18     for i := 1; i <= 5; i++ {
19         fmt.Printf("%d - %d\n", i, nextEven())
20     }
21 }
22
```

```
1 - 0
2 - 2
3 - 4
4 - 6
5 - 8
```

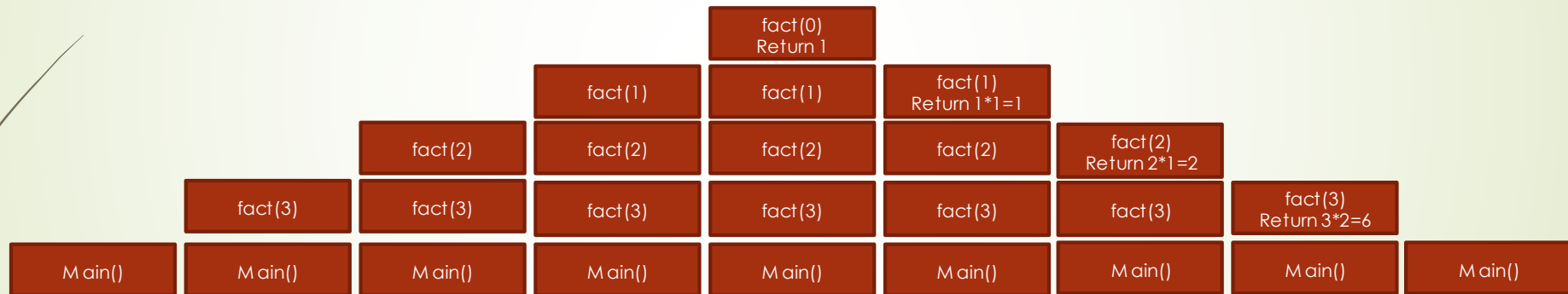
Récurtivité

- ➔ Go supporte les appels de fonctions récursifs (une fonction s'appelant elle même)

```
recursive-functions.go x
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func factorial(x float64) float64 {
8      if x == 0 {
9          return 1
10     }
11     return x * factorial(x-1)
12 }
13
14 func main() {
15     var x float64 = 20
16     fmt.Printf("Factorial(%v) = %v\n", x, factorial(x))
17 }
18
```

```
Factorial(20) = 2.43290200817664e+18
```

Récurtivité (exemple: 3!)



Récurtivité (Suite de Syracuse)

```
func syracuse(x int64) int64 {  
    fmt.Printf("%d,", x)  
    if x == 0 || x == 1 {  
        return 1  
    } else if x%2 == 0 {  
        return syracuse(x / 2)  
    } else {  
        return syracuse(3*x + 1)  
    }  
}
```

```
syracuse(28) = [28,14,7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]
```

Les defers

- Les **defer** sont des fonctions dont l'exécution est différée jusqu'à la fin de l'exécution de sa fonction parente (appelante)
- Les **defer** sont empilées dans la pile d'appels
- Quand la fonction retourne, ces defers sont exécutées dans l'ordre LIFO

```
defers.go x
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      defer fmt.Printf("End of count down!\n")
10
11     for i := 0; i < 10; i++ {
12         defer fmt.Printf("Count: %d\n", i)
13     }
14
15 }
16
```

Les defers

- Les **defer** sont souvent utilisées lorsqu'une ressource a besoin d'être libérée en fin de traitement
- Dans l'exemple, ceci apporte trois avantages majeurs :
 - Garder l'appel de **close** près de l'**open** de façon à rendre le code plus clair
 - Si la fonction à des return multiples, la defer sera appelée dans tous les cas
 - Les fonctions différées seront appelée même en cas de **panic**

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

Panic et recover

- Les **panic** sont déclenchées par des runtime erreurs
- On peut les gérer par la fonction standard **recover**
- **recover** intercepte la **panic** et retourne la valeur qui a été passée à l'appel de **panic**
- On pourrait être tenté de l'utiliser comme dans l'exemple, mais l'appel à **recover** ne sera jamais effectué car l'appel à **panic** stoppe immédiatement l'exécution de la fonction
- A la place, on doit coupler la **panic** avec un **defer**

```
package main

import "fmt"

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

```
package main

import "fmt"

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

Panic et recover

```
panic-recover.go x
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8
9      defer func() {
10         message := recover()
11         fmt.Printf("Message -> %v", message)
12     }()
13
14     myString := "abcdefghijkl"
15
16     for i := 0; i <= len(myString)+1; i++ {
17         fmt.Printf("Letter[%v] is %q\n", i, myString[i])
18     }
19
20 }
21
```

```
Letter[0] is 'a'
Letter[1] is 'b'
Letter[2] is 'c'
Letter[3] is 'd'
Letter[4] is 'e'
Letter[5] is 'f'
Letter[6] is 'g'
Letter[7] is 'h'
Letter[8] is 'i'
Letter[9] is 'j'
Letter[10] is 'k'
Message -> runtime error: index out of range
```


Pointeurs

- Lorsque l'on appelle une fonction qui attend un argument, cet argument est **copié** dans la fonction
- Dans ce programme, la fonction `setToZero` ne modifiera pas la variable originale définie dans `main()`
- Mais que faire si nous avons souhaité le faire ?
- Une façon de faire serait d'utiliser un type de donnée particulier : un pointeur
- Un pointeur identifie l'emplacement mémoire où la valeur est stockée plutôt que la valeur elle-même
- En utilisant un pointeur (`*int8`), la fonction `setToOne` est capable de modifier la variable d'origine

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func setToZero(x int8) {
8     x = 0
9     fmt.Printf("In setToZero(), x is: %d\n", x)
10 }
11
12 func setToOne(xPtr *int8) {
13     *xPtr = 1
14     fmt.Printf("In setToOne(), x is : %d\n", *xPtr)
15 }
16
17 func main() {
18     var x int8 = 10
19     fmt.Printf("In main(), x is : %d\n", x)
20     setToZero(x)
21     fmt.Printf("In main(), x is : %d\n", x)
22     setToOne(&x)
23     fmt.Printf("In main(), x is : %d\n", x)
24 }
25
```

Pointeurs (opérateurs * et &)

- En Go, le pointeur est représenté par * (astérisque) suivi du type de la variable stockée
- Dans la fonction setToOne, xPtr est un pointeur vers un `int8`
- * est aussi utilisé pour déréférencer les variables de type pointeur
- Le déréférencement nous donne accès à la valeur référencée par le pointeur
- Lorsque l'on écrit `*xPtr = 0`, nous disons "Stocker l'`int8` de valeur 0 dans l'emplacement mémoire indiqué par `xPtr`"
- Si nous essayons `xPtr = 0`, nous aurons une erreur de compilation car xPtr n'est pas un entier (`int8`) mais un pointeur vers un entier (`*int8`) qui ne peut recevoir qu'un autre pointeur vers un entier
- Enfin, nous utilisons l'opérateur & pour trouver l'adresse de stockage d'une variable
- `&x` retourne un `*int8` (pointeur vers un `int8`) car x est un entier (`int8`)
- `&x` et `xPtr` font référence au même emplacement mémoire

Structures

- Considérons le programme ci-contre
- Garder une trace de toutes les coordonnées des formes rend le programme difficile à lire et à comprendre ce qu'il fait
- Il finira, au fil de ses extensions de fonctionnalités, par mener à des erreurs

```
1 package main
2
3 import (
4     "math"
5     "fmt"
6 )
7
8 func distance(x1, y1, x2, y2 float64) float64 {
9     a := x2 - x1
10    b := y2 - y1
11    return math.Sqrt(a*a + b*b)
12 }
13
14 func rectangleArea(x1, y1, x2, y2 float64) float64 {
15     l := distance(x1, y1, x1, y2)
16     w := distance(x1, y1, x2, y1)
17     return l * w
18 }
19
20 func circleArea(x, y, r float64) float64 {
21     return math.Pi * r * r
22 }
23
24 func main() {
25     var rx1, ry1 float64 = 0, 0
26     var rx2, ry2 float64 = 10, 10
27     var cx, cy, cr float64 = 0, 0, 5
28     fmt.Println(rectangleArea(rx1, ry1, rx2, ry2))
29     fmt.Println(circleArea(cx, cy, cr))
30 }
```

Structures

- Le mot-clé **type** introduit un nouveau type de donnée
- Il est suivi du nom (**Circle** ou **Rectangle**)
- Le mot-clé **struct** et le bloc (entre accolades) contenant la liste des champs de la nouvelle structure
- Chaque champ à un nom et un type
- **c := Circle{x:0, y:0, r:1}** déclare et initialise la structure
- **x := new(Rectangle)** aurait déclaré et initialisé une variable de type **Rectangle** mais aurait retourné un pointeur (***Rectangle**) et non pas une variable de type **Rectangle**

```
Circle area (x: 1, y: 0, r: 1) is 3.141592653589793
Rectangle area (x: 0, y: 0, l: 2, w: 2) is 4
```

```
1 package main
2
3 import (
4     "math"
5     "fmt"
6 )
7
8 type Circle struct {
9     x float64
10    y float64
11    r float64
12 }
13
14 type Rectangle struct {
15     x float64
16     y float64
17     l float64
18     w float64
19 }
20
21 func circleArea(c Circle) float64 {
22     return math.Pi * c.r * c.r
23 }
24
25 func rectangleArea(r Rectangle) float64 {
26     return r.l * r.w
27 }
28
29 func main() {
30     c := Circle{x: 0, y: 0, r: 1}
31     fmt.Printf(
32         "Circle area (x: %v, y: %v, r: %v) is %v\n",
33         c.x, c.y, c.r, circleArea(c))
34     r := Rectangle{0,0,2,10}
35     r.x = 0
36     r.y = 0
37     r.l = 2
38     r.w = 2
39     fmt.Printf(
40         "Rectangle area (x: %v, y: %v, l: %v, w: %v) is %v\n",
41         r.x, r.y, r.l, r.w, rectangleArea(r))
42 }
43
```


Méthodes

- Le programme est devenu plus lisible mais on peut encore l'améliorer en utilisant un type de fonction particulier : une méthode
- Entre le mot-clé **func** et le nom de la fonction, on insère un récepteur
- Le récepteur est semblable à un paramètre (il a un nom et un type) mais en créant la fonction de cette manière, cela nous permet de pouvoir l'appeler en utilisant l'opérateur **.** (point)

```
Area of Rectangle(10,100) is 1000
Area of Circle(10) is 314.1592653589793
```

```
1 package main
2
3 import (
4     "math"
5     "fmt"
6 )
7
8 type Rectangle struct {
9     width float64
10    length float64
11 }
12
13 type Circle struct {
14     radius float64
15 }
16
17 func (r Rectangle) area() float64 {
18     return r.length * r.width
19 }
20
21 func (r Circle) area() float64 {
22     return math.Pi * r.radius * r.radius
23 }
24
25 func main() {
26     var myRectangle Rectangle
27     myRectangle.width = 10
28     myRectangle.length = 100
29     fmt.Printf(
30         "Area of Rectangle(%v,%v) is %v\n",
31         myRectangle.width, myRectangle.length, myRectangle.area())
32
33     myCircle := new(Circle)
34     myCircle.radius = 10
35     fmt.Printf(
36         "Area of Circle(%v) is %v\n",
37         myCircle.radius, myCircle.area())
38 }
39
```


Types incorporés

- Les champs d'une structure matérialisent une relation <objet> possède <attribut>
- Un parallélogramme **ne possède pas** de rectangle mais **est une** forme rectangulaire **possédant** une hauteur
- Go supporte ce type de relation : les types incorporés ou les champs anonymes

```
Parallelogram(1,1,1) volume is 1  
Parallelogram width is 1 and length is 1
```

```
embedded-types.go  
1 package main  
2  
3 import (  
4     "fmt"  
5 )  
6  
7 type Rectangle struct {  
8     width float64  
9     length float64  
10 }  
11  
12 type Parallelogram struct {  
13     Rectangle  
14     height float64  
15 }  
16  
17 func (r Rectangle) area() float64 {  
18     return r.width * r.length  
19 }  
20  
21 func (p Parallelogram) volume() float64 {  
22     return p.area() * p.height  
23 }  
24  
25 func main() {  
26     width := 1.  
27     length := 1.  
28     height := 1.  
29     myParallelogram := Parallelogram{Rectangle{width, length}, height}  
30     fmt.Printf(  
31         "Parallelogram(%v,%v,%v) volume is %v\n",  
32         width, length, height, myParallelogram.volume()  
33     )  
34     fmt.Printf(  
35         "Parallelogram width is %v and length is %v\n",  
36         myParallelogram.Rectangle.width, myParallelogram.Rectangle.length  
37     )  
38 }
```

Références

- Golang Project: <https://golang.org/>
- Goland-book.com: <https://www.golang-book.com/books/intro>
- Didier Gérard: <https://lemag.sfeir.com/pourquoi-golang/>
- Wikipédia: [https://fr.wikipedia.org/wiki/Go_\(langage\)](https://fr.wikipedia.org/wiki/Go_(langage))
- The Go Programming Language: Donovan, Kernigan
- Goland IDE: <https://www.jetbrains.com/go/>

- Gitlab course link : <https://gitlab.com/ThierryDecker/learning-go>
- Slice Tricks: <https://github.com/golang/go/wiki/SliceTricks>