THE GO PROGRAMMING LANGUAGE

With Emphasis on Concurrency

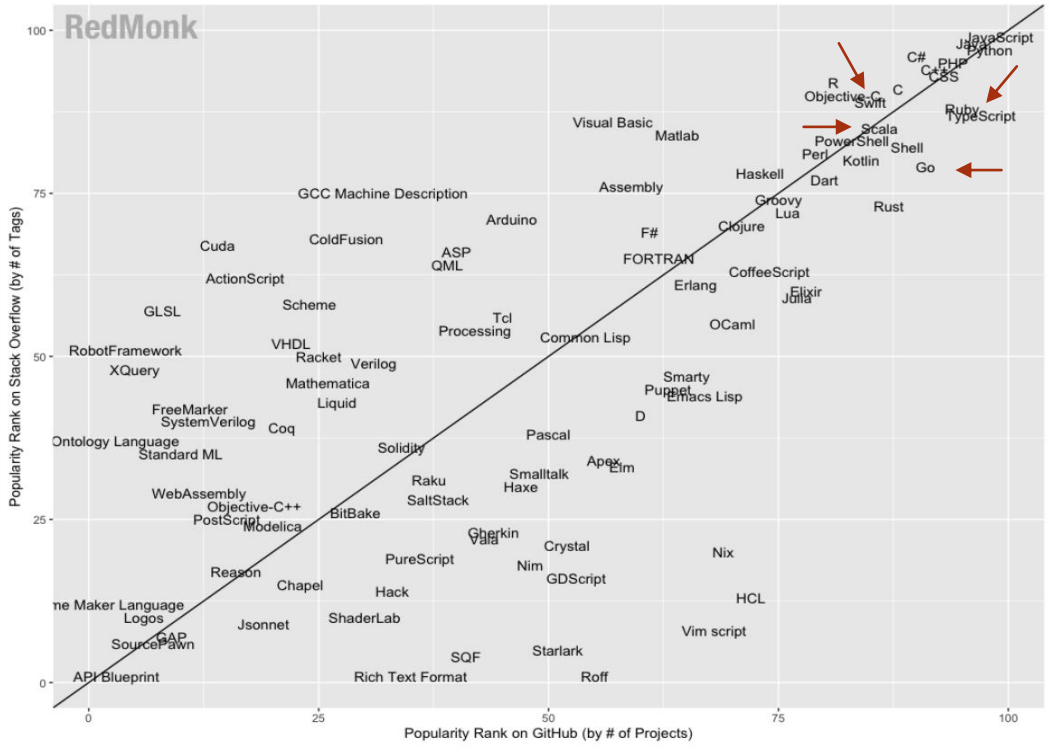
Julio Lezama Amastalli

Some reasons to learn the Go programming language

Features

- Fast compilation,
- Simplicity,
- Garbage Collection,
- **Is mainly designed for the concurrent programming.**

RedMonk Q121 Programming Language Rankings

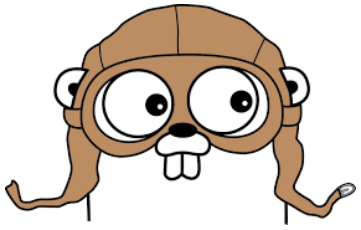


1 JavaScript
2 Python
3 Java
4 PHP
5 C#
5 C++
5 CSS
8 TypeScript
9 Ruby
10 C
11 Swift
12 R
13 Objective-C
14 Shell
14 Scala
16 Go
17 PowerShell
18 Kotlin
19 Rust

Source: The RedMonk Programming Language Rankings: Q1 2021

Agenda

- History
- Syntax Basics
- Concurrency
- The Polynomial Example
- Java Comparation



https://golang.org

History

Creation
Creators
Main influences

Conceived in 2007

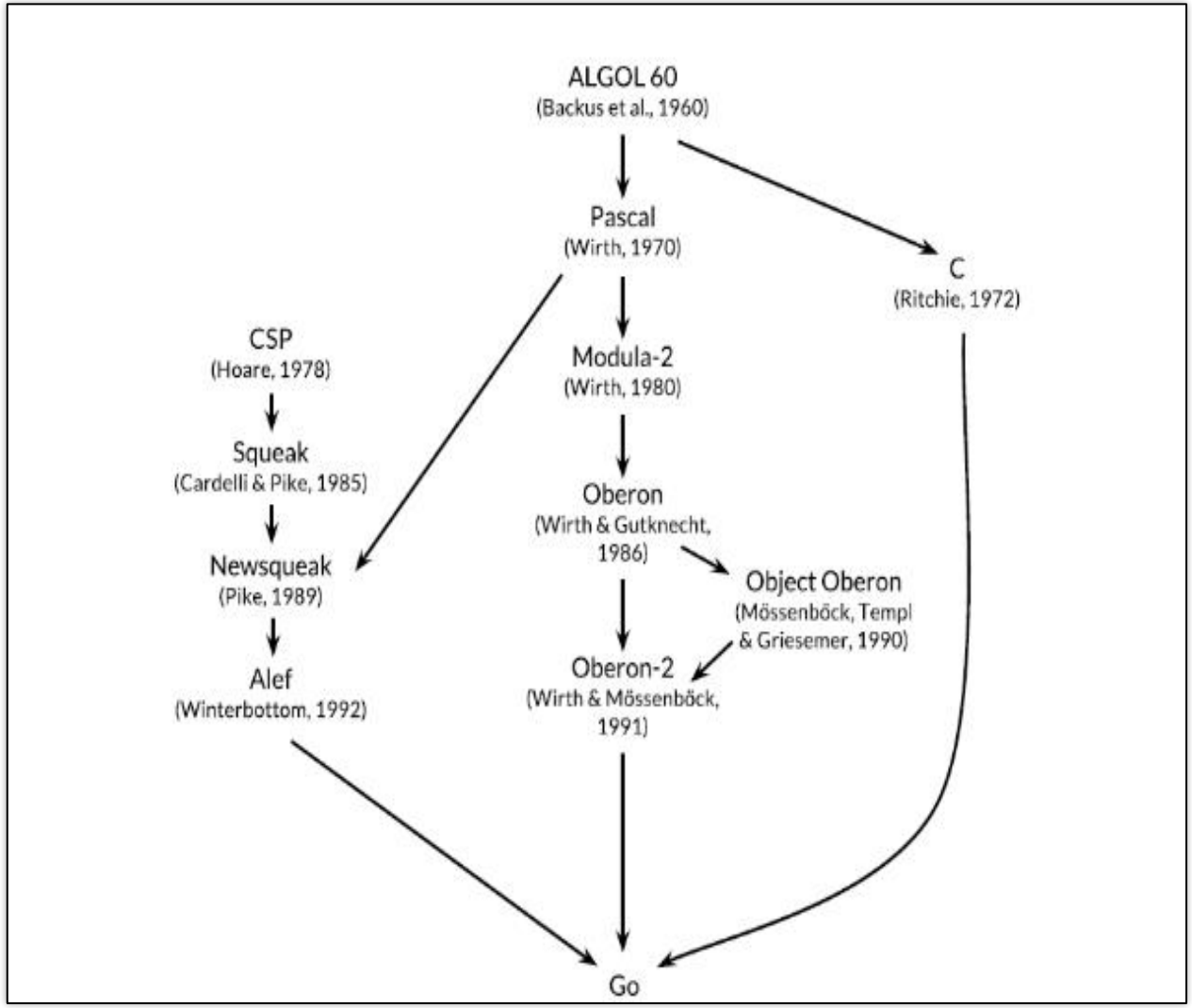
Announced 2009

i. Ken Thompson,

ii. Robert Pike and

iii. Robert Griesemer.

Most important influences of earlier programming languages on the design of Go.



Source: Donovan and Kernighan (2015)

Syntax Basics

Hello, world!

01helloWorld.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6     fmt.Println("Hello, world!")
7  }
```

Description

The typical Go file Layout

- The `package` clausule
- Any `import` statements
- The actual code

Calling the `Println` function

- `Println` Function is in the `fmt` (format) `package`.
- To be able to call `Println`, we first have to import the `package` containing it.
- `fmt.Println("Hello, world!")` specifies that we are calling a function that is part of the "`fmt`" package.

Declaring Variables and Constants

02declaringVarCons.go

```
01  package main
02
03  import "fmt"
04
05  func main()  {
06    var day string // Declaration
07    day = "Monday"
08    fmt.Println("Today is ", day)
09    var length, width int // Multiple declaration
10    length, width = 10, 20
11    fmt.Println(length, width)
12    var letter = "A" // Declaration and assignment together
13    fmt.Println(letter)
14    food := “Cake" // Short variable declaration
15    fmt.Println(food)
16  }
```

Description

- A **variable** is a piece of storage containing a value.
- We use the `var` keyword, followed by the desired name and the type of values the variable will hold.
- Declaration, Mutiple declaration, Declaration and Assignament together and Short variable declaration

- **Constant**: named values that never change.
- To declare a constant we write
- `const name myConstant type = value`
- We must assign a value at the time the constant is declared; we can't assign a value later as with variables.
- Variables have the **: =** short variable declaration syntax available, **but there is no equivalent for constants.**

Naming rules

03namingRules.go

```
01  package main
02
03  import "fmt"
04
05  func main() {
06    var Myvariable string
07    var myvariable string
08    fmt.Println(Myvariable)
09    fmt.Println(myvariable)
10  }
```

Description

- A name must begin with a letter, and can have any number of additional letters and numbers.
- If the name of a **variable, function, or type** begins with a capital letter, it is considered **exported** and can be accessed from packages outside the current one.
- If a **variable, function or type** name begins with a lowercase letter, it is considered **unexported** and can only be accessed within the current package.

Types of data: Strings and Runes

04stringsAndRunes.go

```
01  package main
02
03  import "fmt"
04
05  func main() {
06    fmt.Println("@")
07    fmt.Println('@')
08    fmt.Println("Hello world!")
09    //fmt.Println('Hello world!')
10  }
```

Description

- Go uses **UTF-8**, a standard that represents Unicode characters using 1 to 4 bytes each.
- Strings are a sequence of UTF-8 characters.
- Strings are value types and immutable: once created you cannot modify the contents of the string.
- We can define strings directly within our code using **string literals**: text between double quotation marks that Go will treat as a string.
- Go's runes are used to represent single characters. Rune literals are written with single quotation marks (').

Types of data: Booleans

05booleans.go

```
01  package main
02
03  import "fmt"
04
05  func main() {
06    a := true
07    b := false
08    var c bool
09    fmt.Println(a)
10    fmt.Println(b)
11    fmt.Println(c)
12  }
```

Description

- Boolean values can be one of only two predefined constants: **true** or **false**.
- Two values of the same type can be compared with each other with the relational operators **==** and **!=** Producing a boolean value: **true** or **false**.
- Boolean constants and variables can also be combined with logical operators (**not**, **and**, **or**) to produce a Boolean value.

Types of data: Integers and Floats

06integersFloats.go

```
package main

import "fmt"

func main() {
  var a int32
  var b float32
  a = 10
  b = 10
  a = b + a // compiler error
  fmt.Println(a)
}
```

Description

- Go provides four distinct signed integers, represented by the types: `int8`, `int16`, `int32`, and `int64`.
- Go also provides unsigned integers, represented by the types `uint8`, `uint16`, `uint32`, and `uint64`.
- There are also `int` and `uint` types that are the natural size or most efficient for signed and unsigned integers on a particular platform.
- The signed numbers of an n-bit number are -2^{n-1} to $2^{n-1}-1$. Unsigned integers use the full range of bits for non-negative values and therefore have a range of 0 to 2n - 1. For example, the range of int8 is −128 to 127, while the range of uint8 is 0 to 255.
- Go has two sizes of floating-point numbers: `float32` and `float64`.
- **Because Go is strongly typed, mixing of types is not allowed, as in the program 04integersFloats.**

Types of data: Complex Numbers

07complexNumbers.go

```
01  package main
02
03  import "fmt"
04
05  func main()  {
06    var myComplexNumber complex64 = 1 + 2i
07    fmt.Println(myComplexNumber)
08    fmt.Println(real(myComplexNumber))
09    fmt.Println(imag(myComplexNumber))
10  }
```

Description

- Go provides two sizes of complex numbers, `complex64` and `complex128`, whose components are `float32` and `float64`, respectively.
- A complex number is written in the form: re + im*i*, where **re** is the real part, and **im** is the imaginary part, and **i** is the square root of -1.
- The functions **`real(myComplexNumber)`** and **`imag(myComplexNumber)`** give the real and imaginary part respectively.

Logical and Arithmetic Operations

08logicalArithOper.go

```
01  package main
02
03  import "fmt"
04
05  func main() {
06    var a  = 20
07    var b  = 10
08    var c  = 15
09    var d  = 5
10    var e int
11
12    e = a + b * c / d
13    fmt.Println("Value of a + b * c / d is :",  e )
14
15    e = (a + b * c) / d
16    fmt.Println("Value of (a + b * c) / d is  :" ,  e )
17
18    e = ((a + b) * c)  / d
19    fmt.Println("Value of ((a + b) * c) / d) is  :",  e )
20  }
```

Description

- Logical Operations
  - ==
  - !=
  - <,<=, > and >=
  - && | | !
- Arithmetic Operations
  - +, -, * and /

Precedence Operators

Precedence	Operator(s)
7	^ !
6	* / % << >> & &^
5	+ - | ^
4	== != < <= >= >
3	<-
2	&&
1	| |

Zero values

09zeroValues.go

```
01  package main
02
03  import "fmt"
04
05  func main() {
06    var myString string
07    var myInteger int
08    var myFloat   float32
09    var myBoolean  bool
10    fmt.Println(myString, myInteger, myFloat, myBoolean)
11  }
```

Description

- A variable declared without being assigned a value is automatically initialized with the zero value of its type

Zero values

Type	Zero value
complex	(0+0i)
String	“” (empty string)
Float	0
Integer	0
Bool	false

Control Structure: if else

10ifElse.go

```
01  package main
02
03  import (
04    "fmt"
05    "math/rand"
06    "time"
07  )
08
09  func main() {
10    seconds :=time.Now().Unix() // Number of seconds since Junuary 1, 1970
11    rand.Seed(seconds)
12    myNumber := rand.Intn(3)-1
13    fmt.Println(myNumber)
14    if myNumber < 0 {
15      fmt.Println("myNumber is negative")
16    } else if myNumber == 0{
17      fmt.Println("my Number is equal to cero")
18    } else {
19      fmt.Println("myNumber is positive")
20    }
21  }
```

Description

- The `if` test a conditional statement.
- In this case we have 3 exclusive branches.
- The number of `else if`—branches is in not limited.
- The **{}** are mandatory.
- The **{** after the if and else must be on the same line.
- The else if and else keywords must be on the same line as the closing **}**.

Control Structure: switch

11switch.go

```
01  package main
02
03  import (
04    "fmt"
05  )
06
07  func main() {
08
09    i := 1
10    fmt.Print("Write ", i, " as ")
11    switch i {
12    case 1:
13      fmt.Println("one")
14    case 2:
15      fmt.Println("two")
16    case 3:
17      fmt.Println("three")
18    }
19  }
```

Description

- **i** is a variable which can be of any `type`, values in lines 12, 14 and 16 must be of the same `type`, or expressions evaluating to that `type`.
- The opening **{** has to be on the same line as the `switch`.
- More than one value can be tested in a case: **case** val1, val2, val3.
- Each case-branch is exclusive; they are tried first to last.
- **fallthrough.**

Lopping Structure: for

12for.go

```
1  package main
2
3  import "fmt"
4
5  func main()  {
6    for x := 1; x <= 10; x++ {
7      fmt.Println("x is now", x)
8    }
9  }
```

Description

- **Each loop in Go is a `for` statement**.
- Loops always begin with the `for` keyword.
- In one common loop, `for` is followed by three segments of code that control the loop:
  - The initialization statement,
  - A condition that determines the end of the cycle and
  - A post statement, which is executed after each iteration of the loop.

Arrays

13array.go

```
01  package main
02
03  import "fmt"
04
05  func main() {
06    var myArray [5]int
07    fmt.Println("myArray:", myArray)
08    myArray[4] = 10
09    fmt.Println("myArray initialized on index 4:", myArray)
10    fmt.Println("The length of myArray:", len(myArray))
11    myArray2 := [5]int{1, 2, 3, 4, 5}
12    fmt.Println("myArray2 declared and initialized:", myArray2)
13    var romanNumbers [3]string = [3]string{"I", "II", "III"}
14    // romanNumbers := [3]string{"I", "II", "III"}
15    fmt.Println("romanNumbers:", romanNumbers)
16  }
```

Description

- An array is a numbered and fixed-length sequence of data *items* (elements) of the same *single type*.
- The *items* can be accessed (and changed) through their *index* (the position).
- When declaring an array, each item in it is automatically **initialized with the default zero-value** of the type.
- If we know in advance what values an array should hold, we can initialize it with those values using an **array literal**.

Slices

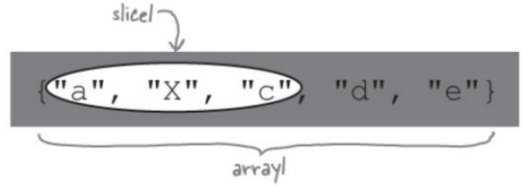
14slice.go

```
01  package main
02
03  import "fmt"
04
05  func main() {
06    var mySlice []int
07    fmt.Println(mySlice)
08    var mySlice2 []int
09    mySlice2 = make([]int, 4)
10    fmt.Println(mySlice2)
11    var mySlice3 = []string{"a","b","c"}
12    fmt.Println(mySlice3)
13    mySlice4 := []string{"I","II","III"}
14    fmt.Println(mySlice4)
15    array1 :=[]string{"a","b","c","d","e"}
16    mySlice5 := array1[2:4]
17    fmt.Println(mySlice5)
18  }
```

Description

- Slices are a collection type that can grow to hold additional items.
- To declare the type for a variable that holds a slice, we use an empty pair of square brackets, followed by the type of elements the slice will hold.
- Unlike with array variables, declaring a slice variable doesn't automatically create a slice. For that, we can call the built-in `make` function.
- We pass make the type of the slice we want to create (which should be the same as the type of the variable you're going to assign it to), and the length of slice it should create.
- If we know in advance what values a slice will start with, we can initialize the slice with those values using a **slice literal**.

Underlying Array



Source: McGabren (2019)

Maps

15maps.go

```
01  package main
02
03  import "fmt"
04
05  func main() {
06    var myMap map[string]int
07    myMap = make(map[string]int)
08    myMap["Ruby"] = 2
09    myMap["Adda"] = 1
10    myMap["Cobol"] = 3
11    myMap["Go"] = 4
12    fmt.Println(myMap)
13    ranks := map[string]int{"First":1, "Second":2, "Third":3}
14    fmt.Println(ranks)
15  }
```

Description

- A **map** is a collection where each value is accessed via a *key*. A map can use any type for keys.
- If we declare a map variable, but don't assign it a value, its value will be `nil`.
- How to tell zero values apart from assigned **values**

Functions

16functionDeclaring.go

```
01  package main
02
03  import "fmt"
04
05  func plusInt(a int, b int) int {
06    return a + b
07  }
08
09  func main() {
10    res := plusInt(1, 2)
11    fmt.Println("1+2 =", res)
12  }
```

Description

- A function is a ***block*** of code that takes an input, processes it, and returns it an output.
- The input is a *tuple* (can be empty) of parameters and returns *tuple* (can be empty) of values.
- A function is the basic ***block*** of code in a program.
- There are 3 types of functions in Go:
  i. Normal functions with identifier,
  ii. Anonymous functions or lambda functions and
  iii. Methods.

Structs

17structs.go

```
01  package main
02
03  import "fmt"
04  func main() {
05    var myStruct struct{
06      figure string
07      area float32
08    }
09    fmt.Println(myStruct)
10    myStruct.figure = "Circle"
11    myStruct.area = 5
12    fmt.Println(myStruct)
13  }
```

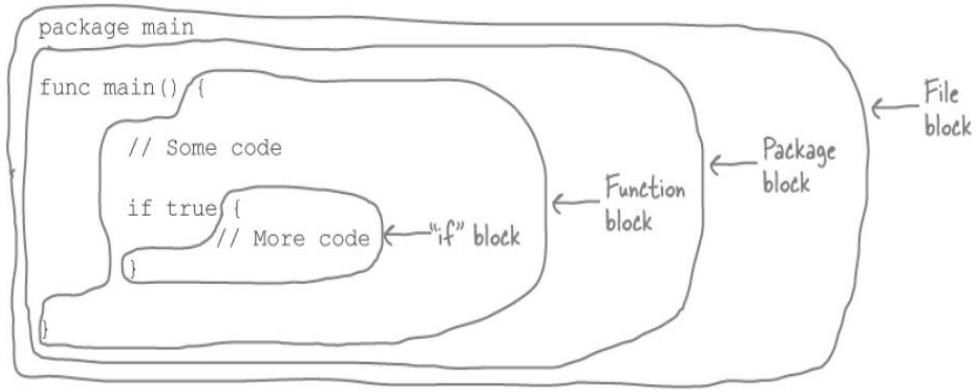
Description

- A **struct** is a value that is constructed out of other values of many different types.
- Within the braces, you can define one or more **fields**: values that the struct groups together.
- You can use a struct type as the type of a variable you're declaring.
- We can use a dot operator to indicate fields that "belong to" a struct.

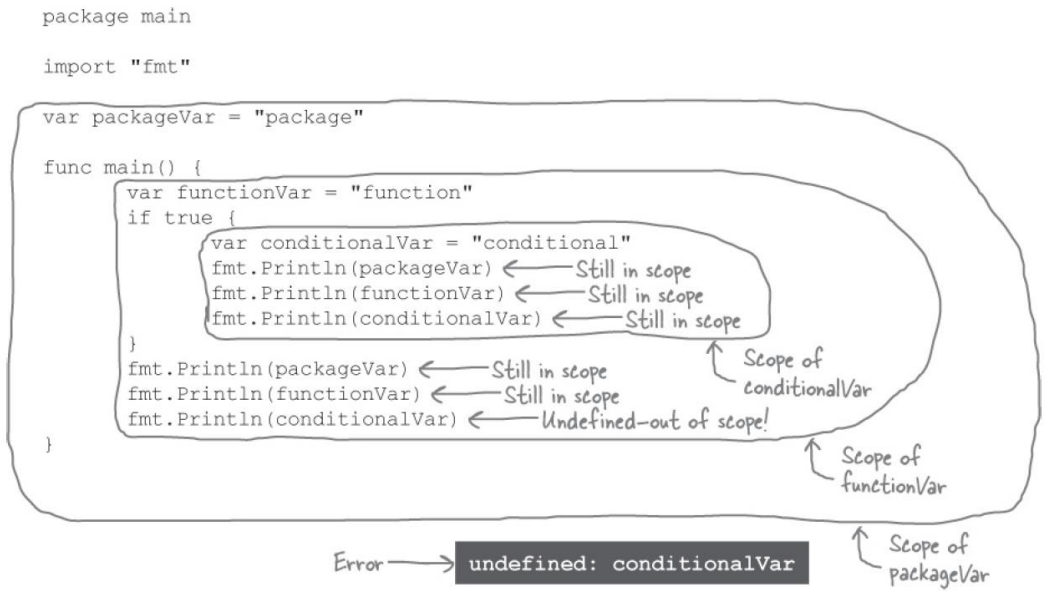
Blocks and Scope

- Go code can be divided up into **blocks**, segments of code.
- Blocks are usually surrounded by curly braces ({}), although there are also blocks at the source code file and package levels.
- Blocks can be nested inside one another.
- Each variable you declare has a **scope**: a portion of your code that it's "visible" within.
- A variable's scope consists of the block it's declared in and any blocks nested within that block.
- See 16scopeBlocks.go

Block



Scope



Source: McGabren (2019)

The Go Programming Language Specification

Extended Backus-Naur Form

For statements Example

ForStmt = "for" [Condition | ForClause | RangeClause] Block .

Condition = Expression .

Description

- https://golang.org/ref/spec
- https://github.com/JulioLezamaAmastalli/golang

See 19aGame.go

The Polynomial Example

Similarities and Differences

Java Implementation

- Defining a package called polynomial
- Defining a class called Polynomial within the polynomials package
- Definition of the methods
  1. degree
  2. plus
  3. minus
  4. times
  5. compose
  6. evaluate
  7. differentiate
  8. toString

- See 19Polynomials.java

Go Implementation

- Defining a package called polynomial
- There are no classes
- Definition of the methods
  1. ~~degree~~
  2. Plus
  3. Minus
  4. Times
  5. Compose
  6. Evaluate
  7. Differentiate
  8. ToString
  9. ToPol

- See 19Polynomials.go

Concurrency

Introduction

Concurrency

- There are lots of situation where programs are just sitting around waiting.
- **Concurrency** allows a program to pause one task and work on other tasks. A program **waiting** for user input might do other processing in the background.
- If a program is written to support concurrency, then it may also support **parallelism**: running tasks simultaneously.
- A computer with only one processor can only run one task at a time.
- Goroutines.

Figure 1. Sequential.

Figure 2. Parallelism.

Figure 3. Concurrent.

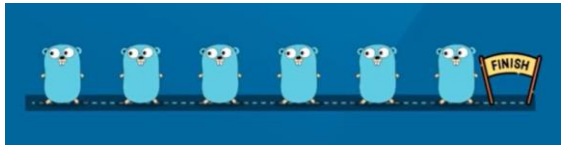
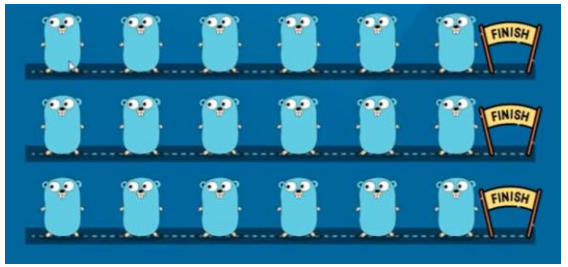


Figure 4. Concurrent and Parallel.



Source: Edteam (2020)

Goroutines

20goroutines.go

```
func a() {
  for i := 0; i < 100; i++{
    fmt.Print("a")
  }
}

func b() {
  for i := 0; i < 100; i++{
    fmt.Print("b")
  }
}

func c() {
  for i := 0; i < 100; i++{
    fmt.Print("c")
  }
}

func main() {
  go a()
  go b()
  go c()
  time.Sleep(time.Second)
  fmt.Println("End main() function")
}
```

Description

- In Go, concurrent tasks are called **goroutines**. Other programming languages have a similar concept called *threads*, but goroutines require less computer memory than threads, and less time to start up and stop, meaning you can run more goroutines at once.
- They're also easier to use. To start **another** goroutine, we use a **go** statement, which is just an ordinary function or method call with the **go** keyword in front of it.
- The main function of every Go program is started using a goroutine, so every Go program runs at least one goroutine.
- We don't directly control when goroutines run.

Channels

21channels.go

```
package main

import "fmt"

func greeting(myChannel chan string)  {
  myChannel <- "Hi!"
}

func main() {
  myChannel := make(chan string)
  go greeting(myChannel)
  fmt.Println(<-myChannel)
}
```

Description

- Go won't let you use the return value from a function called with a go statement, because there's no guarantee the return value will be ready before we attempt to use it.
- There *is* a way to communicate between goroutines: **channels**.
- Channels allow us to send values from one goroutine to another, **they ensure the sending goroutine has sent the value before the receiving goroutine attempts to use it.**

- `ch := make(chan int)`
- `ch <- int1`
- `int2 = <- ch`
- `<- ch`

Synchronizing goroutines with channels

22synchronize.go

```
01  func abc(channel chan string) {
02    channel <- "a"
03    channel <- "b"
04    channel <- "c"
05  }
06
07  func def(channel chan string){
08    channel <- "d"
09    channel <- "e"
10    channel <- "f"
11  }
12
13  func main() {
14    channel1 := make(chan string)
15    channel2 := make(chan string)
16    go abc (channel1)
17    go def (channel2)
18    fmt.Println(<-channel1)
19    fmt.Println(<-channel2)
20    fmt.Println(<-channel1)
21    fmt.Println(<-channel2)
22    fmt.Println(<-channel1)
23    fmt.Println(<-channel2)
24    fmt.Println()
25  }
```

Description

- Channels also ensure the sending goroutine has sent the value before the receiving channel attempts to use it. Channels do this by:
  - **Blocking**—by pausing all further operations in the current goroutine. A send operation blocks the sending goroutine until another goroutine executes a receive operation on the same channel. And vice versa: a receive operation blocks the receiving goroutine until another goroutine executes a send operation on the same channel.
- This behavior allows goroutines to **synchronize** their actions—that is, to coordinate their timing.

Select

23select.go

```
01  func main() {
02    runtime.GOMAXPROCS(2) // in goroutine_select2.go
03    ch1 := make(chan int)
04    ch2 := make(chan int)
05    go pump3(ch1)
06    go pump2(ch2)
07    go suck1(ch1, ch2)
08    time.Sleep(1e9)
09  }
10  func pump3(ch chan int) {
11    for i := 0; ; i++ {
12      ch <- i
13    }
14  }
15  func pump2(ch chan int) {
16    for i := 0; ; i++ {
17      ch <- i
18    }
19  }
20  func suck1(ch1 chan int, ch2 chan int) {
21    for {
22      select {
23      case v := <-ch1:
24        fmt.Printf("Received on channel 1: %d\n", v)
25      case v := <-ch2:
26        fmt.Printf("Received on channel 2: %d\n", v)
27      }
28    }
29  }
```

Description

- Getting the values out of different concurrently executing goroutines can be accomplished with the **select keyword**, which closely resembles the switch control statement and is sometimes called the *communications switch*.
- It acts like an are you ready polling mechanism; select listens for incoming data on channels, but there could also be cases where a value is sent on a channel.
- A select is terminated when a **break** or **return** is executed in one of its cases.

Wait Group

21swaitGroup.go

```go
var wg sync.WaitGroup // Wait for a collection of goroutines to run

func sayHi(names []string) {
  defer wg.Done() // Decrease the number of goroutines to wait by one
  for _, name := range names {
    fmt.Printf("Hi %s\n\r", name)
    time.Sleep(1 * time.Second)
  }
}

func sayBye(names []string) {
  defer wg.Done() // Decrease the number of goroutines to wait by one
  for _, name := range names {
    fmt.Printf("Bye %s\n\r", name)
    time.Sleep(1 * time.Second)
  }
}

func main() {
  wg.Add(2) // Add the number of goroutines to wait

  nombres := []string{"Andres", "Brandon", "Luis",
    "Braulio", "Salvador", "Diego"}

  go sayHi(nombres)
  go sayBye(nombres)

  wg.Wait() // Wait until the goroutines run
  fmt.Println("End")
}
```

Description

- With `WaitGroup` from the `sync` package we can define a collection of goroutines whose execution must be waited before continuing with the process.
- With the `Add` method we indicate the number of goroutines to wait.
- The `Done` method indicates that a goroutine has concluded and is subtracted from the waiting group.
- Finally, the `Wait` method forces us to wait for the execution of the number of indicated goroutines.
- For this reason, the statement fmt.Println ("End") will be executed after both goroutines have finished their execution.

Once

21once.go

```
01  package main
02
03  import (
04    "fmt"
05    "sync"
06  )
07
08  var doOnce sync.Once
09
10
11  func main() {
12    DoSomething()
13    DoSomething()
14    //fmt.Println(reflect.TypeOf(doOnce))
15  }
16
17  func DoSomething() {
18    doOnce.Do(func() {
19      fmt.Println("Run once - first time, loading...")
20    })
21    fmt.Println("Run this every time")
22  }
```

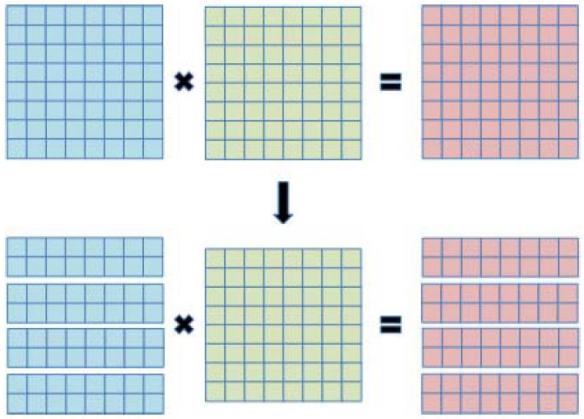
Description

- Sometimes we want to load a resource **only once** and when it's first needed.
- The best way to achieve this is using the sync.Once to efficiently run code once even across goroutines.

Matrix Multiplication

Togashi and Klyuev (2014)

Method of Matrix Multiplication



Source: Togashi and Klyuev (2014)

Implementation

1) Java Implementation: In Java code of this experiment, they simply extend the Thread class, and override the run method:

```
class ParallelMatrix extends Thread{
        public void run(){
                /* implementation */
        }
}
```

Before the calculation, they get available CPUs on local machine by using availableProcessors() method to specify the number of processes.

```
NumOfThreads=Runtime.getRuntime()
        .availableProcessors();
```

After that, they use the start() method to start created threads, and they use the join() method to wait for finish of threads:

```
for(int i=0; i<NumOfThreads; i++){
        threads[i].start();
}
for(int i=0; i<NumOfThreads; i++){
        threads[i].join();
}
```

References

Non-exhaustive list of references

1. McGabren J., "Head First GO A Brain-Friendly Guide", O'Reilly Media, Inc, 2019.
2. Balbaert I., "The Way to Go A Thorough Introduction to the Programming Language", iUniverse Inc, 2012.
3. Bach J. and Aronowitz A., "Go Programming Languaje The Ultimate Beginner's Guide to Learn Go Programming Step by Step", mEm Inc, 2021.
4. Donovan A. and Kernighan B., "The GO Programming Languaje", Addison-Wesley Progessional Computing Series, 2015.
5. Chisnall D., "The Go Programming Language", Addison - Wesley, 2012.
6. Prasertsang A. and Pradubsuwun D., "Formal Verification of Concurrency in Go", International Joint Conference on Computer Science and Software Engineering (JCSSE), 2016.
7. Prabhakar R. and Kumar R., "Concurrent Programming in Go", http://www.golang.org, 2011.