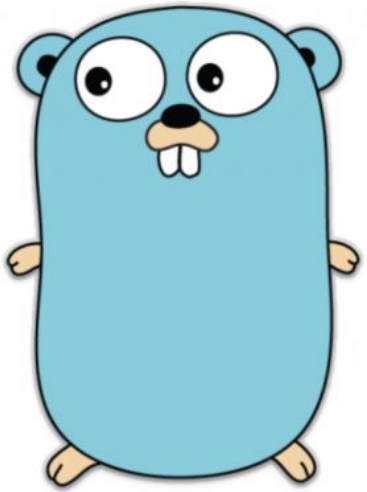
F1/10 자율주행자동차 1

golang



정보통신공학과
2016013181
조수민

패키지(Package),임포트(Import)

```
package main

import "fmt"

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

```
Hello, World!
```

Package

첫 줄을 보면 `package main` 이라고 쓰여져 있는데, 이는 이 파일이 `main` 패키지에 포함된다는 뜻이다. Golang 프로그램은 패키지로 이루어져 있는데, 그 중 `main` 패키지만이 Golang 프로그램이 실행될 수 있는 패키지이다.

Import

다른 패키지를 프로그램에서 사용하기 위해서는 **import** 를 사용하여 패키지를 포함시킨다.

함수(Function)

```
package main

import "fmt"

func add(x int, y int) int {
    return x + y
}

func main() {
    fmt.Println(add(42, 13))
}
```

함수는 특정 기능을 위해 만든 여러 문장을 묶어서 실행하는 코드 블록 단위입니다

기본적인 형태의 함수 선언은 **"func 함수이름 (매개변수이름 매개변수형) 반환형{}"**입니다.

매개변수괄호 다음부분에서 리턴 값이 없으면 괄호를 포함해 모두 생략할 수 있으며, 리턴 값이 하나일 때는 괄호를 생략할 수 있고, 리턴값이 두 개 이상일 경우, 꼭 괄호를 붙여줘야 한다.

함수(Function)

```
package main

import "fmt"

func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}

func main() {
    fmt.Println(split(17))
}
```

```
7 10
```

함수는 매개변수를 취합니다. Go에서 함수는 여러 개의 결과를 반환할 수 있습니다. 반환 값에 이름을 부여하면 변수처럼 사용할 수도 있습니다.

결과에 이름을 붙히면, 반환 값을 지정하지 않은 `return` 문장으로 결과의 현재 값을 알아서 반환합니다.

변수 (Variables)

```
1 var i int
2 var x, y, z int
3 var c, python, java bool
4
```

변수를 선언을 위해 `var` 을 사용합니다.

함수의 매개변수처럼 타입은 문장 끝에 명시합니다.

```
1 var i int = 4
2 var x, y, z int = 1, 2, 3
```

변수 선언과 함께 변수 각각을 초기화를 할 수 있습니다.

초기화를 하는 경우 타입(type)을 생략할 수 있습니다. 변수는 초기화 하고자 하는 값에 따라 타입이 결정됩니다.

```
1 i := 4
2
```

함수 내에서 `:=` 을 사용하면 `var` 과 타입을 생략할 수 있습니다.

(그러나 함수 밖에서는 `:=` 선언을 사용할 수 없습니다.)

For (반복문 for)

```go
package main

import "fmt"

func main() {
    sum := 0
    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println(sum)
}

```

```go
sum := 1
for sum < 1000 {
    sum += sum
}
```

Go 언어는 반복문이 `for` 밖에 없습니다.

기본적인 `for` 반복문은 C와 Java 언어와 거의 유사합니다. 다른점은 소괄호 ( )가 필요하지 않다는 것입니다.

하지만 실행문을 위한 중괄호 { } 는 필요합니다.

C와 Java에서 처럼 전.후 처리를 제외하고 조건문만 표현할 수도 있습니다.

조건문 (If)

```
if k == 1 {
    println("One")
}
```

```
if v := 0; v < lim {
    return v
}
```

```
if v := 0; v < lim {
    return v
} else {
    return
}
```

if 문은 C와 Java와 비슷합니다. 조건 표현을 위해 ( ) 는 사용하지 않습니다. 하지만 실행문을 위한 { } 는 반드시 작성해야합니다.

for 처럼 if 에서도 조건문 앞에 짧은 문장을 실행할 수 있습니다.짧은 실행문을 통해 선언된 변수는 if *안쪽 범위* 에서 만 사용할 수 있습니다.

if 에서 짧은 명령문을 통해 선언된 변수는 else 블럭 안에서도 사용할 수 있습니다.

Struct (구조체)

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p := person{}

    p.name = "Lee"
    p.age = 10

    fmt.Println(p)
}
```

```go
var p1 person
p1 = person{"Bob", 20}
p2 := person{name: "Sean", age: 50}
```

Go의 struct는 필드들의 집합체이며 필드들의 컨테이너이다.

struct를 정의하기 위해서는 type 문을 사용한다. 예를 들어 name과 age 필드를 갖는 person 이라는 struct를 정의하기 위해서는 사진과같은 type문을 사용할 수 있다.

선언된 struct 타입으로부터 객체를 생성하는 방법은 몇 가지 방법들이 있다. 위의 예제처럼 person{} 를 사용하여 빈 person 객체를 먼저 할당하고, 나중에 그 필드값을 채워넣는 방법이 있다.

struct 객체를 생성할 때, 초기값을 함께 할당하는 방법도 있다. 즉, 아래 예제처럼, struct 필드값을 순서적으로 { } 괄호안에 넣을 수 있다

. 연산자를 이용하면 필드에 접근할 수 있다.

메서드(Method)

```
1  package main
2
3  type Rect struct {
4      width, height int
5  }
6
7  func (r Rect) area() int {
8      return r.width * r.height
9  }
10
11 func main() {
12     rect := Rect{10, 20}
13     area := rect.area()
14     println(area)
15 }
```

200

함수명 앞에 타입과 변수명이 붙어 있습니다.
이런걸 리시버(Receiver)라고 부릅니다. 리시버가 장착된 함수는 더 이상 함수가 아니라 메소드가 됩니다.

메소드는 구조체의 필드들을 이용해 특정 기능을 하는 특별한 함수입니다. 특별한 함수인 만큼 선언 방법도 특이합니다.

기본적으로 메소드는 "func (매개변수이름 구조체이름) 메소드이름() 반환형 {" 형식으로 선언합니다. 매개변수 이름은 구조체 변수명으로서 메소드 내에서 매개변수처럼 사용됩니다.

익명함수

```go
package main

import "fmt"

func main() {
    func() {
        fmt.Println("hello")
    }()

    func(a int, b int) {
        result := a + b
        fmt.Println(result)
    }(1, 3)

    result := func(a string, b string) string {
        return a + b
    }("hello", " world!")
    fmt.Println(result)

    i, j := 10.2, 20.4
    divide := func(a float64, b float64) float64 {
        return i / j
    }(i, j)
    fmt.Println(divide)
}
```

```
hello
4
hello world!
0.5
```

함수명을 갖지 않는 함수를 익명함수이라 부른다. 일반적으로 익명함수는 그 함수 전체를 변수에 할당하거나 다른 함수의 파라미터에 직접 정의되어 사용되곤 한다.

함수의 이름만 없고 그 외에 형태는 동일합니다.

함수의 블록 } 뒤에 괄호()를 사용해 함수를 바로 호출합니다. 이때, 괄호 안에 매개변수를 넣을 수 있습니다.

변수에 초기화한 익명 함수는 변수 이름을 함수의 이름처럼 사용할 수 있다는 것입니다.

일급함수

```go
package main

import "fmt"

func calc(f func(int, int) int, a int, b int) int {
    result := f(a, b)
    return result
}

func main() {
    multi := func(i int, j int) int {
        return i * j
    }

    r1 := calc(multi, 10, 20)
    fmt.Println(r1)

    r2 := calc(func(x int, y int) int { return x + y }, 10, 20
    fmt.Println(r2)
}
```

RUN

```
200
30
```

Go 프로그래밍 언어에서 함수는 일급함수로서 Go의 기본 타입과 동일하게 취급되며, 따라서 다른 함수의 파라미터로 전달하거나 다른 함수의 리턴값으로도 사용될 수 있다. 즉, 함수의 입력 파라미터나 리턴 파라미터로서 함수 자체가 사용될 수 있다.

함수를 다른 함수의 파라미터로 전달하기 위해서는 익명함수를 변수에 할당한 후 이 변수를 전달하는 방법과 직접 다른 함수 호출 파라미터에 함수를 적는 방법이 있다.

클로저 (Closure)

```go
package main

import "fmt"

func next() func() int {
    i := 0
    return func() int {
        i += 1
        return i
    }
}

func main() {
    nextInt := next()

    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    newInt := next()
    fmt.Println(newInt())
}
```

```
1
2
3
1
```

클로저는 함수 안에서 익명 함수를 정의해서 바깥쪽
함수에 선언한 변수에도 접근할 수 있는 함수를 말합니다.

defer

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      defer fmt.Println("world")
7      fmt.Println("Hello")
8  }
```

```
Hello
world
```

defer는 함수 앞에 쓰이는 키워드로써 특정 문장 혹은 함수를 감싸고 있는 함수 내에서 제일 나중에, 끝나기 직전에 실행하게 하는 용법입니다.

defer를 사용한 함수들이 여러개면 제일 나중에 지연 호출한 함수가 제일 먼저 실행되는 것입니다.

panic

```
1  package main
2
3  import  "fmt"
4
5  func main() {
6      var num int = 10.5
7      fmt.Println(num)
8  }
```

```
./prog.go:6:16: constant 10.5 truncated to integer
```

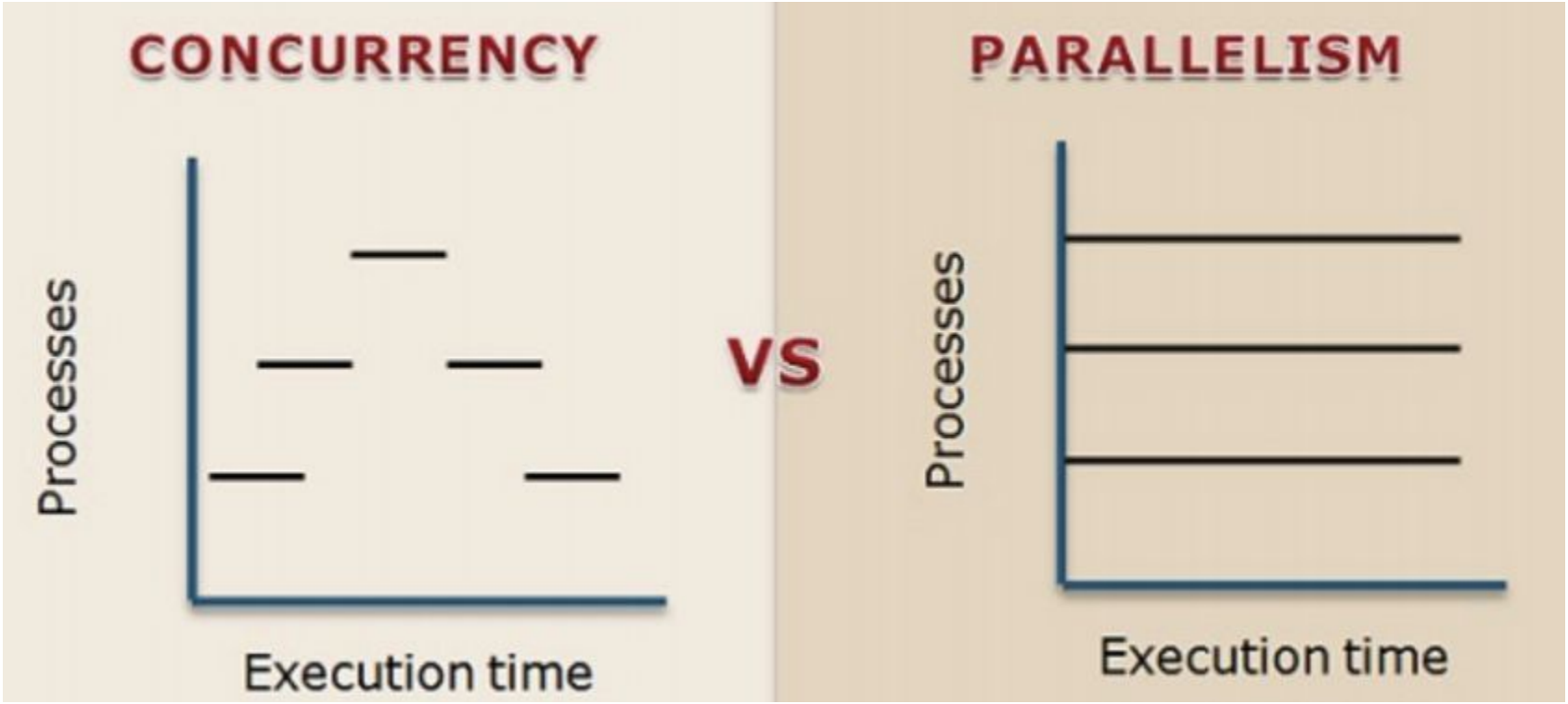
```
1  package main
2
3  import  "fmt"
4
5  func main() {
6      var num1, num2 int = 10, 0
7      fmt.Println(num1 / num2)
8  }
```

```
panic: runtime error: integer divide by zero

goroutine 1 [running]:
main.main()
        /tmp/sandbox425149133/prog.go:7 +0x20
```

panic은 겉으로 보이게 아무런 문제가 없는데 실행해보니 에러가 발생해서 프로그램을 종료하는 기능을 합니다. 반대로 말하자면 문법 자체를 잘못 입력했을 때 발생하는 에러는 **panic**이 아닙니다.

그리고 만약에 **panic**이 발생한 함수 안에 **defer** 구문이 있다면 프로그램을 종료하기 전에 **defer** 구문을 실행하고 종료합니다.

동시성(Concurrency)



동시성(Concurrency)

Process - 공식적인 설명은 운영체제에서 사용하는 task의 단위이나 **쉽게 말하면 main함수**라고도 할 수 있다. 프로그램을 실행시키면 가장 먼저 켜진다. 프로세스를 켜기 위해서는 **os에게 요청**을 해야한다. . 생성과 삭제, 컨텍스트 스위칭시에 **비용이 크다.**

Thread - **모든 프로세스는 반드시 적어도 한개는 쓰레드를 가진다.** 쓰레드는 **프로세스를 다시 단위로 쪼갠 것**이다. 쓰레드를 생성하기 위해서는 **os에게 요청**을 해야한다. 생성과 삭제, 컨텍스트 스위칭시의 **비용이 프로세스보다는 작다.**

Goroutine - os에게 요청하지 않고 사용하는 경량 쓰레드.. os에서 실행하지 않는다는 말은 바꿔말하면 go언어 내부에서 자체적으로 관리하고 실행한다는 뜻이다. 당연히 os에 대한 요청이 없으므로 비용이 프로세스보다도 작다.

Goroutine

```
1  package main
2
3  import "fmt"
4
5  func testGo() {
6      fmt.Println("Hello World!")
7  }
8
9  func main() {
10     go testGo()
11 }
```

```
9  func main() {
10     go testGo()
11
12     fmt.Scanln()
13 }
```

```
Hello, World!
```

고루틴은 다른 함수를 동시에 실행할 수 있는 함수를 일컫는다. 고루틴을 생성하려면 `go`라는 키워드 다음에 함수 호출을 지정하면 된다.

고루틴은 **go** 키워드를 사용해 생성할 수 있는데 두 가지 방법으로 생성할 수 있습니다. 하나는 일반 함수를 사용하는 것이며 다른 하나는 익명 함수를 사용하는 것입니다.

고루틴은 **main** 함수와는 독립적으로 실행되지만 **main** 함수가 종료되면 모든 고루틴들이 종료됩니다. 따라서 고루틴보다 **main**이 먼저 종료되는걸 방지하기위한 여러 방법들이 있습니다.

Goroutine-time

```go
package main

import ("fmt"
        "time"
)

func testGo() {
    fmt.Println("Hello World!")
}

func main() {
    go testGo()
    time.Sleep(2 * time.Second)
}
```

```
Hello, World!
```

시간 출력을 위해 `"time"` 패키지를 **import**합니다.

`time.Sleep()`은 프로그램에 대기 시간을 주는 함수입니다.

Goroutine-sync.WaitGroup

```
2nd goroutine sleeping
1st groutine sleeping
All goroutine complete
```

```go
package main

import ("fmt"
        "sync"
)


func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    go func (){
        defer wg.Done()
        fmt.Println("1st groutine sleeping")

    }()

    wg.Add(1)
    go func(){
        defer wg.Done()
        fmt.Println("2nd goroutine sleeping")

    }()

    wg.Wait()
    fmt.Println("All goroutine complete")
}
```

sync패키지에는 저수준의 메모리 접근 동기화에 가장 유용한 동시성 기본 요소 포함

다음은 **sync.WaitGroup**이 제공하는 메서드다.

Add(): WaitGroup에 대기 중인 고루틴 개수 추가

Done(): 대기 중인 고루틴의 수행이 종료되는 것을 알려줌

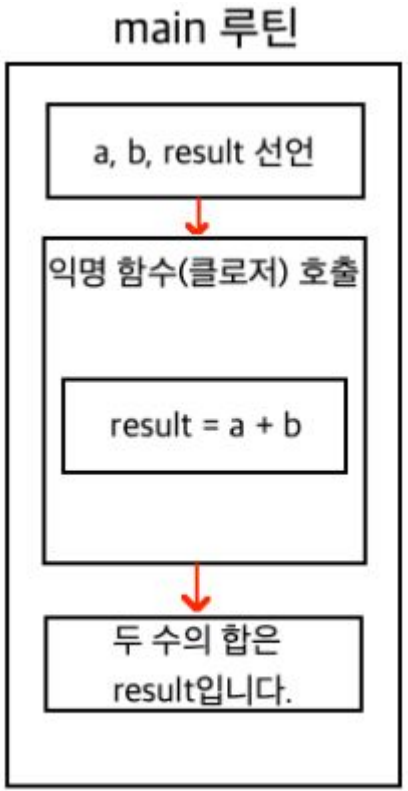
Wait(): 모든 고루틴이 종료될 때까지 대기

Channel

example1

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var a, b = 10, 5
7      var result int
8
9      func() {
10         result = a + b
11     }()
12
13     fmt.Printf("두 수의 합은 %d입니다.", result)
14 }
```

두 수의 합은 15입니다.

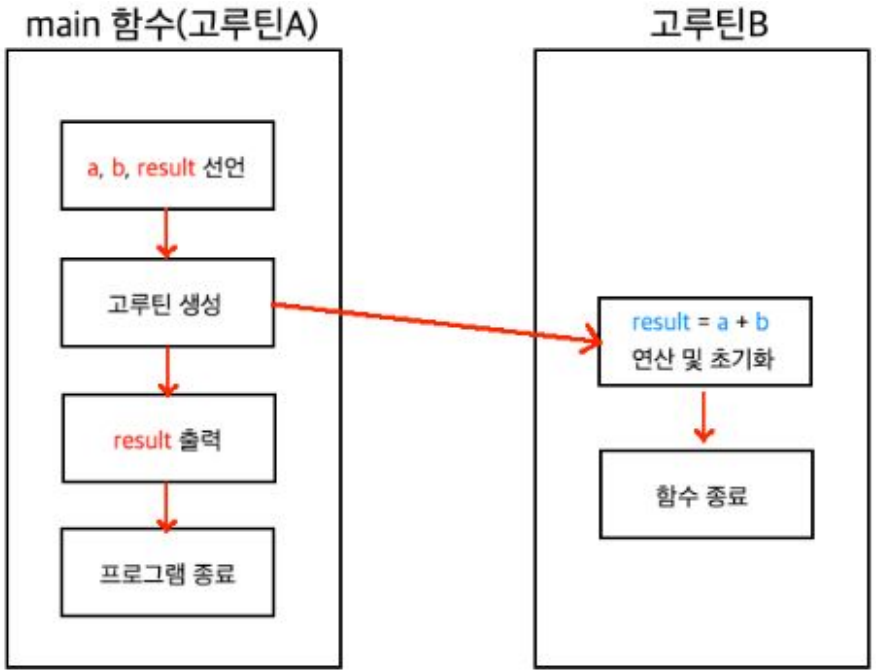


Channel

example2

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var a, b = 10, 5
7      var result int
8
9      go func() {
10         result = a + b
11     }()
12
13     fmt.Printf("두 수의 합은 %d입니다.", result)
14 }
```

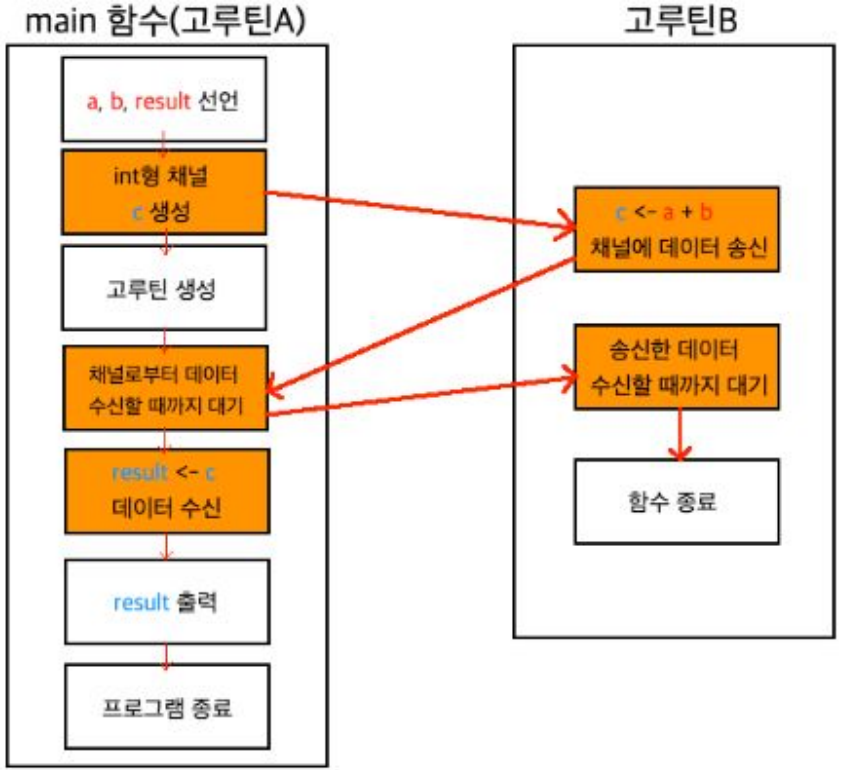
두 수의 합은 0입니다.



Channel

```go
package main

import "fmt"

func main() {
    var a, b = 10, 5
    var result int

    c := make(chan int)

    go func() {
        c <- a + b
    }()

    result = <-c
    fmt.Printf("두 수의 합은 %d입니다.", result)
}
```

두 수의 합은 15입니다.



채널은 고루틴 사이에서 값을 주고받는 통로 역할을 한다

Channel

채널은 고루틴 사이에서 값을 주고받는 통로 역할을 한다

- "make(chan 데이터타입)" 형식으로 생성합니다.
- 채널의 데이터 송/수신은 '<-' 연산자를 이용합니다.
- 채널에 값을 보낼 때는 (채널 <- 데이터), 채널에서 값을 받을 때는 (<- 채널) 입니다. 값을 받을 때는 :=이나 =을 이용해 변수에 바로 값을 대입할 수 있습니다.

Deadlock

```go
package main

import "fmt"

func main() {
    ch := make(chan int)

    ch <- 101

    fmt.Println(<-ch)
}
```

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
        /tmp/sandbox301793553/prog.go:8 +0x60
```

데드락은 둘 이상의 프로세스(함수)가 서로 가진 한정된 자원을 요청하는 경우 발생하는 것으로, 프로세스가 전진되지 못하고 모든 프로세스가 대기 상태가 되는 것을 말합니다.

수신자가 없어 무한 대기로 데드락이 발생하는 코드

Buffer

```go
package main

import "fmt"

func main() {
    ch := make(chan int, 1)

    ch <- 101

    fmt.Println(<-ch)
}
```

```
101
```

Go 채널은 Unbuffered Channel로서 이 채널에서는 하나의 수신자가 데이타를 받을 때까지 송신자가 데이터를 보내는 채널에 묶여 있게 된다. 하지만, Buffered Channel을 사용하면 비록 수신자가 받을 준비가 되어 있지 않을 지라도 지정된 버퍼만큼 데이타를 보내고 계속 다른 일을 수행할 수 있다.

버퍼 채널은 make(chan type, N) 함수를 통해 생성된다.

Select

```go
package main

import (
    "fmt"
    "time"
)
func main() {
    c1 := make(chan int)
    c2 := make(chan string)

    go func() {
        for {
            c1 <- 10
            time.Sleep(100 * time.Millisecond)
        }
    }()
    go func() {
        for {
            c2 <- "Hello, world!"
            time.Sleep(500 * time.Millisecond)
        }
    }()
    go func() {
        for {
            select {
            case i := <-c1:
                fmt.Println("c1 :", i)
            case s := <-c2:
                fmt.Println("c2 :", s)
            }
        }
    }()
    time.Sleep(10 * time.Second)
}
```

```
c1 : 10
c2 : Hello, world!
c1 : 10
c1 : 10
c1 : 10
c1 : 10
c1 : 10
c2 : Hello, world!
```

switch문과 거의 흡사한 용법

select 분기문은 **switch** 분기문과 비슷하지만 **select** 키워드 뒤에 검사할 변수를 따로 지정하지 않으며 각 채널에 값이 들어오면 해당 **case**가 실행됩니다.

Channel parameter

```
1  package main
2
3  import "fmt"
4
5  func producer(c chan<- int){
6      for i :=0; i<5; i++{
7          c<-i
8      }
9      c<-100
10 }
11
12 func consumer(c <-chan int){
13     for i :=range c{
14         fmt.Println(i)
15     }
16     fmt.Println(<-c)
17 }
18 func main(){
19     c := make(chan int)
20
21     go producer(c)
22     go consumer(c)
23
24     fmt.Scanln()
25 }
```

```
0
1
2
3
4
100
```

보내기 전용: (**chan<-자료형**) 형식. (c chan<-int)는 int형 보내기 전용 채널c를 뜻함

보내기 전용 채널은 값을 보낼수만 있으며 값을 가져오려고 하면 에러 발생

받기 전용: (**<-chan 자료형**) 형식. (c<-chan int)는 int형 받기 전용 채널c를 뜻함.

Channel parameter

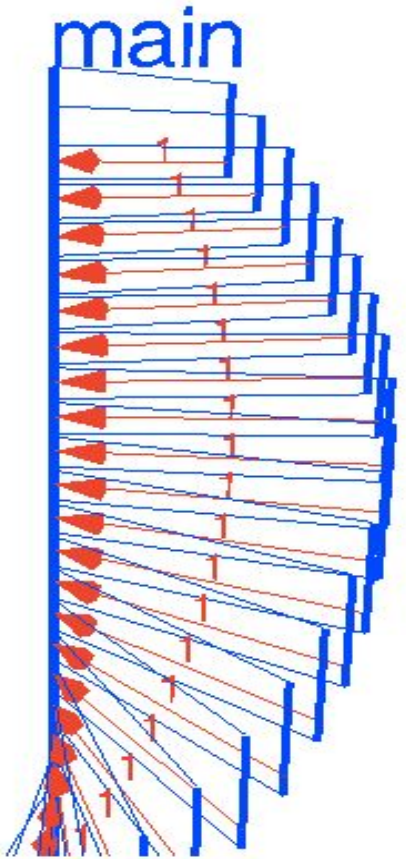
```
1  package main
2
3  import "fmt"
4
5  func sum(a,b int)<- chan int {
6      out := make(chan int)
7      go func(){
8          out<-a+b
9      }()
10     return out
11 }
12 func main(){
13      c:=sum(1,2)
14      fmt.Println(<-c)
15 }
16
```

채널을 리턴 값으로 사용한 형태

```
3
```

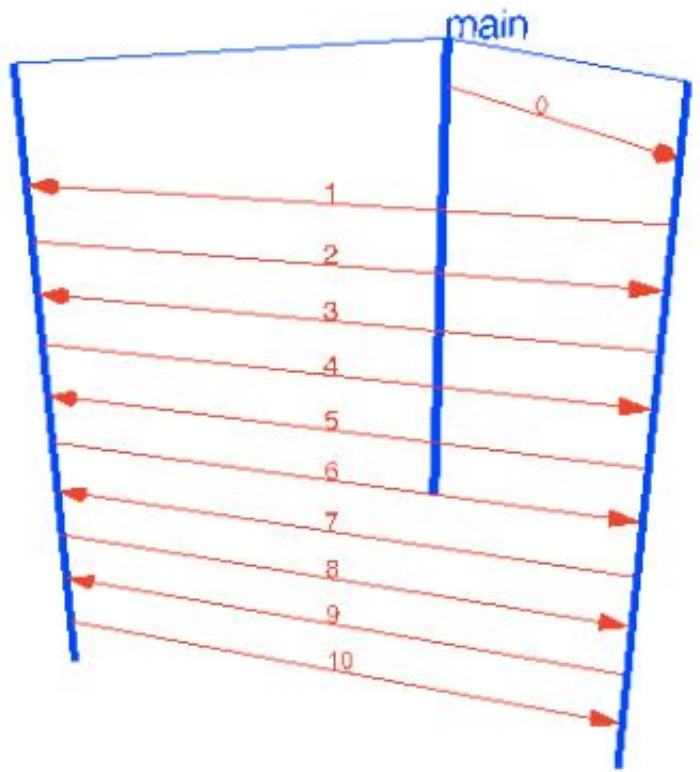
Channel-timers

```go
package main

import "time"

func timer(d time.Duration) <-chan int {
    c := make(chan int)
    go func() {
        time.Sleep(d)
        c <- 1
    }()
    return c
}

func main() {
    for i := 0; i < 24; i++ {
        c := timer(1 * time.Second)
        <-c
    }
}
```



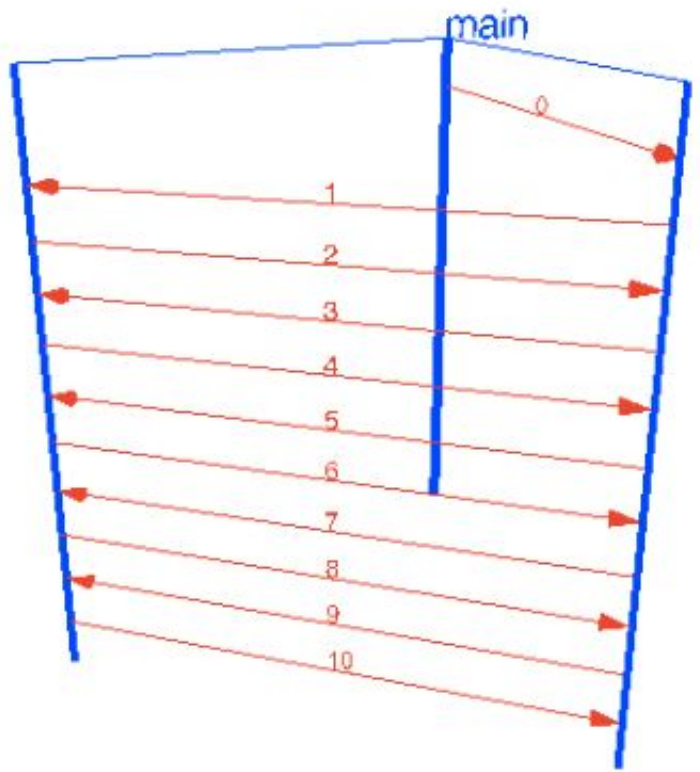
Channel-Ping pong

```go
package main

import "time"

func main() {
    var Ball int
    table := make(chan int)
    go player(table)
    go player(table)

    table <- Ball
    time.Sleep(1 * time.Second)
    <-table
}

func player(table chan int) {
    for {
        ball := <-table
        ball++
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```



Channel-Ping pong

```go
package main

import "time"

func main() {
    var Ball int
    table := make(chan int)
    go player(table)
    go player(table)

    table <- Ball
    time.Sleep(1 * time.Second)
    <-table
}

func player(table chan int) {
    for {
        ball := <-table
        ball++
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```



Channel-Ping pong

```
1  package main
2
3  import "time"
4
5  func main() {
6      var Ball int
7      table := make(chan int)
8      go player(table)
9      go player(table)
10     go player(table)
11
12     table <- Ball
13     time.Sleep(1 * time.Second)
14     <-table
15 }
16
17 func player(table chan int) {
18     for {
19         ball := <-table
20         ball++
21         time.Sleep(100 * time.Millisecond)
22         table <- ball
23     }
24 }
```