Concurrency

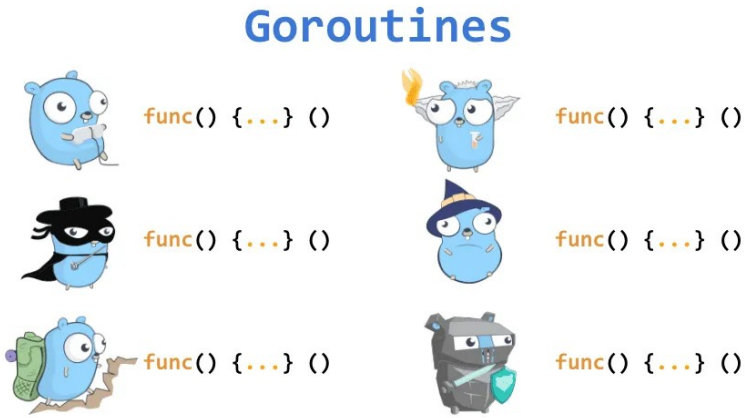
Współbieżność w języku Go odnosi się do zdolności programu do wykonywania wielu zadań równolegle

```
go func(msg string) {
    fmt.Println(msg)
}("going")
```

```
go foo(“going”)
```

Gorutyna pozwala na wykonywanie funkcji w nowym wątku [playground]

```
 func say(s string) {
    for i := 0; i < 3; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}
```



Concurrency

```
func main() {
    say("pjatk")
}

OUTPUT:

pjatk
pjatk
pjatk
```

```
func main() {
    say("pjatk")
    say("wp")
}

OUTPUT:

pjatk
pjatk
pjatk
wp
wp
wp
```

```
func main() {
    go say("pjatk")
    say("wp")
}

OUTPUT:

pjatk
wp
pjatk
pjatk
wp
wp
pjatk
```

Concurrency

gorutyny działają w tej samej przestrzeni adresowej, więc dostęp do pamięci współdzielonej musi być synchronizowany

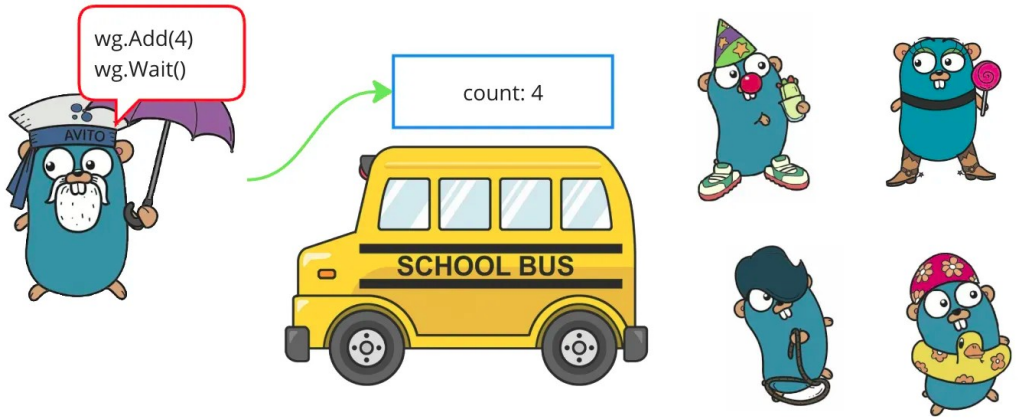
playground

```
var who string

func main() {
    for i := 0; i < 100; i++ {
        go func() { who = "pjatk" }()

        go func() { who = "wp" }()
    }

    time.Sleep(time.Second)
    fmt.Println("Hello " + who)
}
```

Wait Groups

Wait groupy w języku Go to narzędzie, które pomaga programowi poczekać na zakończenie pracy wielu gorutyn.

```
wg := sync.WaitGroup{}
//Add delta to the WaitGroup counter
wg.Add(int)
//Decrements WaitGroup counter by one
wg.Done()
//Wait until WaitGroup counter is zero
wg.Wait()
```



Wait Groups

playground

```
func service(id int) {
    fmt.Printf("Getting data from service %d starting\n", id)
    time.Sleep(time.Second)
    fmt.Printf("Service %d done\n", id)
}

func main() {
    var wg sync.WaitGroup

    for i := 1; i <= 5; i++ {
        wg.Add(1)

        go func() {
            defer wg.Done()
            service(i)
        }()
    }
    wg.Wait()
}
```

Channels (Kanały)

Kanały umożliwiają komunikację pomiędzy gorutynami. Możesz przesyłać wartości do kanałów z jednej gorutyny i odbierać te wartości w innej gorutynie

podobnie jak mapy i tablice, kanały muszą być stworzone przed użyciem

```
ch := make(chan int)
ch <- v    // Send v to channel ch.
v := <-ch  // Receive from ch, and assign value to v.
```

dane są przekazywane w kierunku strzałki <-

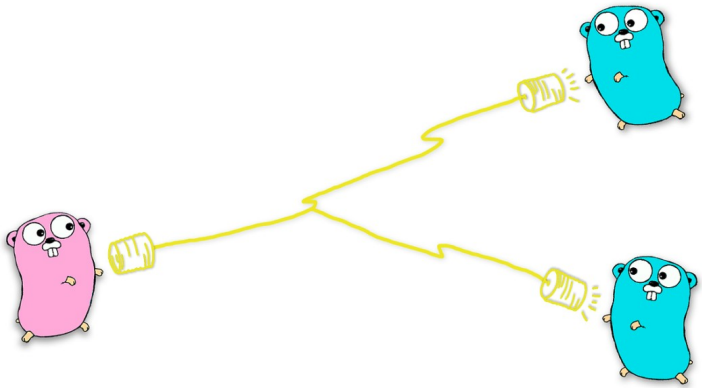
channele można zamykać

```
close(ch)
```

próba zamknięcia zamkniętego channelu = panic

sprawdzenie, czy channel jest zamknięty

```
v, ok := <-ch
```



Channels

playground

```
func main() {
    ch := make(chan string)
    wg := sync.WaitGroup{}
    go func() {
        for v := range ch {
            fmt.Println("Hello " + v)
            wg.Done()
        }
    }()
    wg.Add(2)
    go func() {
        ch <- "pjatk"

    }()
    go func() {
        ch <- "wp"
    }()
    wg.Wait()
}
```

Channels

channnele mogą być buforowane, długość bufora jako drugi argument do funkcji make
playground

```
func main() {
    ch := make(chan string, 2)

    ch <- "pjatk"
    ch <- "wp"

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

Domyślnie wysyłanie i odbieranie jest blokowane do momentu, gdy zarówno nadawca, jak i odbiorca są gotowi.

sync.Mutex

kanały są do świetne komunikacji między gorutynami.
Ale co jeśli nie potrzebujemy komunikacji? Co jeśli po prostu chcemy upewnić się, że tylko jedna gorutyna może jednocześnie uzyskać dostęp do danej zmiennej, aby uniknąć konfliktów?
taki koncept nazywamy mutual exclusion (wzajemne wykluczenie) stąd nazwa pakietu mutex
syntax

```
mu := sync.Mutex{}
mu.Lock()
mu.Unlock()

rwmu := sync.RWMutex{}
rwmu.RLock()
rwmu.RUnlock()
```

sync.Mutex

playground

```
func main() {
    m := make(map[string]int)

    var wg sync.WaitGroup

    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {

            m["pjatk"]++

            wg.Done()
        }()
    }

    wg.Wait()

    fmt.Println(m)
}
```

sync.Mutex

playground

```
var mu sync.Mutex

func main() {
    m := make(map[string]int)
    var wg sync.WaitGroup

    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            mu.Lock()

            m["a"]++

            mu.Unlock()
            wg.Done()
        }()
    }
    wg.Wait()
    fmt.Println(m)
}
```