Learning Objectives

Learners will be able to…

- **Use Goroutines and channels**
- **Use structures like `for...range` and `select` with channels**
- **Synchronize goroutines with Mutexes and WaitGroup**

info

Make Sure You Know

Python or other modern programming language.

Goroutines

A goroutine is a lightweight thread implemented by the Go runtime. Because Goroutines are controlled by language runtime and not by the operating system, they consume less memory and the cost of context switching is much lower.

To start a separate thread in Go, use the `go` keyword:

```
go myFunction()

// or

go func() {
  // your code here
}()
```

If you run the program several times, you will notice the output will be in random order.

Since the execution process occurs in parallel, the order the threads finish is unknown. This behavior can sometimes emerge in the form of a **race condition** if you assume that threads are going to execute in a particular sequence.

Channels, Close, and Range

Channels are used for communication between goroutines.

Channels are typed communication objects through which values can be sent and received.

To define a channel, the keyword `chan` is used, followed by the type of data that is transmitted using it.

```
var sendMessages chan string
```

In this case, the channel is used to transmit data of type `string`.

Before creating channels, you need to create it. The `make` keyword is used for this.

```
sendMessages := make(chan string)
```

The `<-` operator is used to send and receive data to a channel. Data is transmitted in the direction of the arrow.

For example:
* `ch <- data` - send data to the channel.
* `data := <-ch` - receive data from the channel and assign the value to `data`.

Paste the following code in the `main` method and try it

```
numbers := []float64{3, 7, -2, 9, 0, 12, 7}
ch := make(chan float64, 1)
go sumOfNumbersRaisedToPower(numbers, 2, ch)
go sumOfNumbersRaisedToPower(numbers, 3, ch)
part1, part2 := <-ch, <-ch
result := part1 + part2
fmt.Println(result)
```

Note that channels can be passed as function arguments.

close and range

You can also receive data from a channel repeatedly.

Use a for loop with `range` to iterate until the channel is closed:

```
for item := range sendMessages {
    // ...
}
```

To close a channel, call `close`:

```
close(sendMessages)
```

The sender must close the channel.

Paste the following code in the `main` method and try it

```
ch := make(chan int, 3)
go printStep(ch)
for step := range ch {
  fmt.Println(step)
}
```

Sending through a closed channel will cause `panic`. To check whether the channel is closed, you need to check the second parameter in the expression for receiving data from the channel:

```
value, ok := <-sendMessages
```

If the value `ok` is `true` this means that the channel is not closed and you can continue receiving data. Otherwise the channel was closed.

Unlike files, *it is not necessary to close channels.* Close channels to inform the recipient that the channel is closed – for example, in order to complete a `range` loop.

Blocking, Buffered Channels, and Select

Operations for writing and receiving data in channels are **blocking**. This means when data is written to a channel, the thread who wrote the data is blocked until the data is read from it – other goroutines will continue.

Try out the following code:

```
func main() {
    ch := make(chan string)
    ch <- "one"
    fmt.Println(<-ch)
}
```

After executing this code, we will receive the error `fatal error: all goroutines are asleep - deadlock!`.

▼ **Why the error?**

> After placing the data in the channel, the thread was blocked, and the reading that would unblock occurs only further in the same, blocked, goroutine.

Buffered Channels

Channels in Go can also be buffered. That is, the channel has a certain capacity.

To initialize a buffered channel, you must specify its length as an argument:

```
channel := make(chan string, 2)
```

With a buffered channel, the goroutine is blocked only when the buffer is full or, in the case of reading, when the buffer is empty.

Try out the following code:

```
func main() {
    ch := make(chan string, 2)
    ch <- "one"
    fmt.Println(<-ch)
    ch <- "two"
    ch <- "three"
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

In the following example, in the output we will get

```
one
two
three
```

## Try it out

- Try changing the buffer size to `1`:

  ```
  ch := make(chan string, 1)
  ```

  After the first output of `one` we will again receive the error `fatal error: all goroutines are asleep - deadlock!`.

  Blocking will occur due to buffer overflow.

- You can also try adding:

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

  Blocking will occur due to the fact that there will be an attempt to read from an empty channel.

Select

The `select` statement is used for operations with channels and is similar to `switch` without arguments.

The `select` operator is blocking - it waits until one of the goroutines sends data to the channel. Once one of the cases can be executed, it will be executed. If several channels are ready, the case to be executed will be selected randomly.

Try out the following code:

```
import  (
  "fmt"
  "time"
)

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

    go tick(ch)
    go initiateQuit(quit)

    for {
        select {
        case step := <-ch:
            fmt.Println(step)
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

// sends a step number to the channel every 200 ms
func tick(ch chan int) {
    for i := 0; i < 100; i++ {
        ch <- i
        time.Sleep(200 * time.Millisecond)
    }
}

// sends data to the channel one second after it is called
func initiateQuit(quit chan int) {
    time.Sleep(1 * time.Second)
    quit <- 0
}
```

Otherwise, after the first case is executed, the program will continue execution and will be terminated.

A `switch` can have a `default` case which will be executed if none of the other cases are ready.

Try adding a `default` case to the previous example:

```
select {
case step := <-ch:
    fmt.Println(step)
case <-quit:
    fmt.Println("quit")
    return
default:
    time.Sleep(50 * time.Millisecond)
    fmt.Println("default")
}
```

Mutexes

Goroutines run in the same address space, meaning shared memory access should be synchronized.

Try out the example on the left to print 1 - 15

If we run this program, we will see that the output of numbers will not be ordered. Additionally, not all numbers from 1 to 15 will be printed while some may be repeated.

This happens because our goroutines work in parallel and they all have access to the `counter` variable.

Mutexes ensures that only one goroutine have access to a variable at a time, avoiding conflicts.

Go provides `sync.Mutex` in the standard library. Add `sync` to the `imports`:

```
import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)
```

`sync.Mutex` has `Lock` and `Unlock` methods:
* `Lock` is used to block access to a resource
* `Unlock` is used to unlock it

Add a mutex to our `Counter` type:

```
type Counter struct {
    mutex sync.Mutex
    value int
}
```

In the `increment5` function we will call `Lock` and `Unlock` to lock and unlock `counter`:

```
func increment5(counter *Counter) {
    counter.mutex.Lock() // added
    for i := 0; i < 5; i++  {
        counter.value++

time.Sleep(time.Duration(rand.Int63n(50))*time.Millisecond)
        fmt.Println(counter.value)
    }
    counter.mutex.Unlock() // added
}
```

A better solution would be to call `Unlock` using `defer` to ensure we don't forget to call it and create a deadlock:

```
func increment5(counter *Counter) {
    counter.mutex.Lock() //added
    defer counter.mutex.Unlock() //added
    for i := 0; i < 5; i++  {
        counter.value++

time.Sleep(time.Duration(rand.Int63n(50))*time.Millisecond)
        fmt.Println(counter.value)
    }
}
```

▼ **Got lost? Click to view entire updated code**

```
package main

import (
  "fmt"
  "math/rand"
  "sync"
  "time"
)

type Counter struct {
  mutex sync.Mutex
  value int
}

func main() {
  counter := Counter{value: 0}
  for i := 0; i < 3; i++  {
    go increment5(&counter)
  }
  time.Sleep(1*time.Second)
  fmt.Printf("counter: %d\n", counter.value)
}

func increment5(counter *Counter) {
  counter.mutex.Lock()
  defer counter.mutex.Unlock()
  for i := 0; i < 5; i++  {
    counter.value++

time.Sleep(time.Duration(rand.Int63n(50))*time.Millisecond)
    fmt.Println(counter.value)
  }
}
```

Try the updated code with `mutex`

We see that the output is now ordered!

The block of code enclosed between `Lock` and `Unlock` will work in mutual exclusion – only one goroutine will access it at a time. Because goroutines can perform many different actions, only those parts of the code that are enclosed between `Lock` and `Unlock` are blocked – goroutines doing other tasks will be unaffected.

WaitGroup

Another convenient way to control routines is to use `sync.WaitGroup`. `sync.WaitGroup` allows you to wait for goroutines to complete and only then continue executing the code.

has three methods:

- The method call must be executed before the goroutine is called. It is also possible to call the method again after all previous calls to `Wait` have returned.
- Called in the goroutine to signal the thread has completed its execution.
-

Try out the example on the left

In it we launch two goroutines, and only after they are completed we launch the next two.

WaitGroups are useful if the execution of our code needs to continue after some tasks running in goroutines have been completed:

```
var wg sync.WaitGroup
wg.Add(2)
go taskOne(&wg)
go TaskTwo(&wg)
wg.Wait()
// further actions
```