Concurrency

Go provides concurrency constructions as part of the core language. This lesson presents them and gives some examples on how they can be used.

The Go Authors

https://golang.org

*** Goroutines**

A _goroutine_ is a lightweight thread managed by the Go runtime.

```
go f(x, y, z)
```

starts a new goroutine running

```
f(x, y, z)
```

The evaluation of `f`, `x`, `y`, and `z` happens in the current goroutine and the execution of `f` happens in the new goroutine.

Goroutines run in the same address space, so access to shared memory must be synchronized. The [[/pkg/sync/][`sync`]] package provides useful primitives, although you won't need them much in Go as there are other primitives. (See the next slide.)

*** Channels**

Channels are a typed conduit through which you can send and receive values with the channel operator, `<-`.

```
ch <- v    // Send v to channel ch.

v := <-ch  // Receive from ch, and

           // assign value to v.
```

(The data flows in the direction of the arrow.)

Like maps and slices, channels must be created before use:

```
ch := make(chan int)
```

By default, sends and receives block until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

The example code sums the numbers in a slice, distributing the work between two goroutines.
Once both goroutines have completed their computation, it calculates the final result.

*** Buffered Channels**

Channels can be _buffered_. Provide the buffer length as the second argument to `make` to initialize a buffered channel:

```
ch := make(chan int, 100)
```

Sends to a buffered channel block only when the buffer is full. Receives block when the buffer is empty.

Modify the example to overfill the buffer and see what happens.

*** Range and Close**

A sender can `close` a channel to indicate that no more values will be sent. Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression: after

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

`ok` is `false` if there are no more values to receive and the channel is closed.

The loop `for`i`:=`range`c` receives values from the channel repeatedly until it is closed.

***Note:* Only the sender should close a channel, never the receiver. Sending on a closed channel will cause a panic.**

***Another*note:* Channels aren't like files; you don't usually need to close them. Closing is only necessary when the receiver must be told there are no more values coming, such as to terminate a `range` loop.**

*** Select**

The `select` statement lets a goroutine wait on multiple communication operations.

A `select` blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

*** Default Selection**

The `default` case in a `select` is run if no other case is ready.

Use a `default` case to try a send or receive without blocking:

```
select {
case i := <-c:
 // use i
default:
 // receiving from c would block
}
```

*** Exercise: Equivalent Binary Trees**

There can be many different binary trees with the same sequence of values stored in it. For example, here are two binary trees storing the sequence 1, 1, 2, 3, 5, 8, 13.

.image /tour/static/img/tree.png

A function to check whether two binary trees store the same sequence is quite complex in most languages. We'll use Go's concurrency and channels to write a simple solution.

This example uses the `tree` package, which defines the type:

```
type Tree struct {
  Left  *Tree
  Value int
  Right *Tree
}
```

Continue description on [[javascript:click('.next-page')][next page]].

*** Exercise: Equivalent Binary Trees**

***1.* Implement the `Walk` function.**

***2.* Test the `Walk` function.**

The function `tree.New(k)` constructs a randomly-structured (but always sorted) binary tree holding the values `k`, `2k`, `3k`, ..., `10k`.

Create a new channel `ch` and kick off the walker:

```
go Walk(tree.New(1), ch)
```

Then read and print 10 values from the channel. It should be the numbers 1, 2, 3, ..., 10.

***3.* Implement the `Same` function using `Walk` to determine whether `t1` and `t2` store the same values.**

***4.* Test the `Same` function.**

`Same(tree.New(1),`tree.New(1))` should return true, and
`Same(tree.New(1),`tree.New(2))` should return false.

The documentation for `Tree` can be found
[[https://godoc.org/golang.org/x/tour/tree#Tree][here]].

*** sync.Mutex**

We've seen how channels are great for communication among goroutines.

But what if we don't need communication? What if we just want to make sure only

one goroutine can access a variable at a time to avoid conflicts?

This concept is called _mutual_exclusion_, and the conventional name for the data structure that provides it is _mutex_.

Go's standard library provides mutual exclusion with

[[/pkg/sync/#Mutex][`sync.Mutex`]] and its two methods:

- `Lock`

- `Unlock`

We can define a block of code to be executed in mutual exclusion by surrounding it

with a call to `Lock` and `Unlock` as shown on the `Inc` method.

We can also use `defer` to ensure the mutex will be unlocked as in the `Value` method.

*** Exercise: Web Crawler**

In this exercise you'll use Go's concurrency features to parallelize a web crawler.

Modify the `Crawl` function to fetch URLs in parallel without fetching the same URL twice.

Hint: you can keep a cache of the URLs that have been fetched on a map, but maps alone are not
safe for concurrent use!

*** Where to Go from here...**

#appengine: You can get started by

#appengine: [[/doc/install/][installing Go]].

#appengine: Once you have Go installed, the

The

[[/doc/][Go Documentation]] is a great place to

#appengine: continue.

start.

It contains references, tutorials, videos, and more.

To learn how to organize and work with Go code, read [[/doc/code][How to Write Go Code]].

If you need help with the standard library, see the [[/pkg/][package reference]]. For help with the language itself, you might be surprised to find the [[/ref/spec][Language Spec]] is quite readable.

To further explore Go's concurrency model, watch

[[https://www.youtube.com/watch?v=f6kdp27TYZs][Go Concurrency Patterns]]

([[/talks/2012/concurrency.slide][slides]])

and

[[https://www.youtube.com/watch?v=QDDwwePbDtw][Advanced Go Concurrency Patterns]]
([[/talks/2013/advconc.slide][slides]])

and read the

[[/doc/codewalk/sharemem/][Share Memory by Communicating]]

codewalk.

To get started writing web applications, watch

[[https://vimeo.com/53221558][A simple programming environment]]

([[/talks/2012/simple.slide][slides]])

and read the

[[/doc/articles/wiki/][Writing Web Applications]] tutorial.

The [[/doc/codewalk/functions/][First Class Functions in Go]] codewalk gives an interesting perspective on Go's function types.

The [[/blog/][Go Blog]] has a large archive of informative Go articles.

Visit [[/][the Go home page]] for more.