Introduction to Channels

This lesson explains how goroutines can interact with one another through channels.

WE'LL COVER THE FOLLOWING

Concept

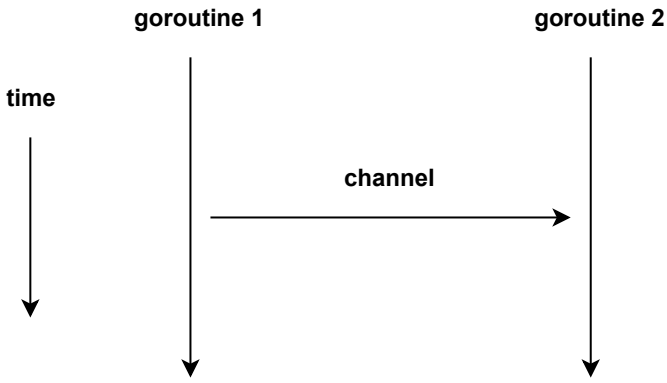
Previously, the goroutines executed independently; they did not communicate. Of course, to be more useful, they have to communicate by *sending and receiving information between them and, thereby, coordinating (synchronizing) their efforts*. Goroutines could communicate by using shared variables, but this is highly discouraged because this way of working introduces all the difficulties with shared memory in multi-threading.

Instead, Go has a special type, the *channel*, which is like a conduit (a pipe) through which you can send typed values and which takes care of communication between goroutines, avoiding all the pitfalls of shared memory. The very act of communication through a channel guarantees synchronization; there is no need for explicit synchronization through locking anything.

Data is passed around on channels: *only one goroutine has access to a data item at any given time: so data races cannot occur, by design*. The ownership of the data (that is the ability to read and write it) is passed around. A useful analogy is to compare a channel with a conveyor belt in a factory. One machine (the producer goroutine) puts items onto the belt, and another

machine (the consumer goroutine) takes them off for packaging.

Channels serve the dual purpose of *communication*, i.e., the exchange of value with *synchronization*, guaranteeing that two calculations (goroutines) are in a known state at any time.



Channels and goroutines

In the general format, the declaration of a channel is:

```
var identifier chan datatype
```

The value of an uninitialized channel is *nil*.

A channel can only transmit data-items of one datatype, e.g. `chan int` or `chan string`, but a channel can be made for any type, also custom types and for the empty interface{}. It is even possible (and sometimes useful) to create a channel of channels.

A channel is, in fact, a *typed message queue*: data can be transmitted through it. It is a First In First Out (FIFO) structure, and so it preserves the order of the items that are sent into it (for those who are familiar with it, a channel can be compared to a two-way pipe in Unix shells).

A channel is also a *reference type*, so we have to use the `make()` function to allocate memory for it. Here is a declaration of a channel ch1 of strings, followed by its creation (instantiation):

```
var ch1 chan string
ch1 = make(chan string)
```

But of course, this can be shortened to:

```
ch1 := make(chan string)
```

Here, we construct a channel of channels of int:

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

Or a channel of functions:

```
funcChan := make(chan func())
```

Channels are first-class objects. They can be stored in variables, passed as arguments to functions, returned from functions, and sent themselves over channels. Moreover, they are typed, allowing the type system to catch programming errors like trying to send a pointer over a channel of integers.

Closing a channel

Once we are done using a channel, it's a good practice to close it. Let's suppose we have a channel `ch`, to close it we can do:

```
close(ch)
```

Communication operator `<-`

This operator represents the transmission of data very intuitively, i.e., information flows in the direction of the arrow.

To a channel (sending)

The operation `ch <- int1` means that the variable `int1` is sent through the channel `ch` (binary operator, infix means send).

From a channel (receiving)

The operation `int2 = <- ch` means that variable `int2` receives data (gets a new value) from the channel `ch` (unary prefix operator, prefix means receive). This supposes `int2` is already declared; if not, this can be written as: `int2 := <- ch`. Another form `<- ch` can on itself be used to take the (next) value from the channel. This value is effectively discarded, but it can be tested upon. That's why the following is legal code:

```
if <-ch != 1000 {
   ...

}
```

The same operator `<-` is used for sending and receiving, but Go figures out what to do depending on the operands. Although not necessary, for readability the channel name usually starts with `ch` or contains `chan`.

The channel `send` and `receive` operations are atomic which means they always complete without interruption.

The use of the communication operator is illustrated in the following example:

```
package main
import (
  "fmt"
  "time"
)

func main() {
  ch := make(chan string)
  go sendData(ch) // calling goroutine to send the data
  go getData(ch)  // calling goroutine to receive the data
  time.Sleep(1e9)
}

func sendData(ch chan string) { // sending data to ch channel
  ch <- "Washington"
  ch <- "Tripoli"
  ch <- "London"
  ch <- "Beijing"
  ch <- "Tokyo"
}

func getData(ch chan string) {
  var input string
  for {
    input = <-ch // receiving data sent to ch channel
    fmt.Printf("%s ", input)
  }
  close(ch) // closed the channel
}
```

Goroutines' communication

In main() at **line 8**, a channel `ch` of strings is made. Then, *two* goroutines are

started: `sendData()` (at **line 9**) sends **5** strings over channel `ch` (see implementation from **line 14** to **line 20**), and `getData()` (at **line 10**) receives them one by one in order in the string input (**line 25**) and prints what is received (**line 26**). If two goroutines have to communicate, you must give them both the same channel as a parameter for doing that. Experiment what happens when you comment out `time.Sleep(1e9)`. Also, remember to comment out the import of the `time` package or your code won't compile.

Here, we see that synchronization between the goroutines becomes important:

- The `main()` waits for 1 second so that both goroutines can come to completion. If this is not allowed, `sendData()` doesn't have the chance to produce its output.
- getData() works with an infinite for-loop. This comes to an end when `sendData()` has finished, and `ch` is empty.
- If we remove one or all `go` keywords, the program doesn't work anymore, and the Go runtime throws a panic:

```
---- Error run <path> with code Crashed Fatal error: all goroutines are asleep - deadlock!
```

Why does this occur? The runtime can detect that all goroutines (or perhaps only one in this case) are waiting for something (to write to a channel or be able to read from a channel), which means the program can't possibly proceed. It is a form of a deadlock, and the runtime can detect it for us.

> **Remark**: Don't use print statements to indicate the order of sending to and receiving from a channel. This could be out of order with what happens due to the time lag between the print statement and the actual channel sending and receiving.

Channels make communication possible between goroutines. To communicate in the best possible manner, we should know the type of connection is required. The next lesson focuses on types of communication between goroutines.