Channel Directionality

This lesson discusses different patterns for the implementation of channels for communication between goroutines.

WE'LL COVER THE FOLLOWING

A channel type may be annotated to specify that it may only send or only receive data:

```
var send_only chan<- int // data can only be sent (written) to the channel
var recv_only <-chan int // data can only be received (read) from the chan
nel
```

Receive-only channels (`<-chan T`) cannot be closed, because closing a channel is intended as a way for a sender goroutine to signal that no more values will be sent to the channel. Therefore, it has no meaning for receive-only channels. All channels are created bidirectional, but we can assign them to *directional channel variables*, like in this code snippet:

```
var c = make(chan int) // bidirectional
go source(c)
go sink(c)
func source(ch chan<- int) {
  for { ch <- 1 } // sending data to ch channel
}
func sink(ch <-chan int) {
  for { <-ch } // receiving data from ch channel
}
```

Channel iterator pattern

This pattern can be applied in the common case where we have to populate a channel with the items of a *container* type, which contains index-addressable field items. For this, we can define a method `Iter()` on the content type which returns receive-only channel items (`Iter()` is a channel factory), as follows:

```
func (c *container) Iter () <-chan items {
  ch := make(chan item)

  go func () {
    for i := 0; i < c.Len(); i++ { // or use a for-range loop
      ch <- c.items[i]
    }
  } ()
  return ch
}
```

Inside the goroutine, a for-loop iterates over the elements in the container `c` (for tree or graph algorithms, this simple for-loop could be replaced with a depth-first search). The code, which calls this method can then iterate over the container, like:

```
for x := range container.Iter() { ... }
```

which can run in its own goroutine. Then, the above iterator employs a channel and two goroutines (which may run in separate threads).

If the program terminates before the goroutine is done writing values to the channel, then that goroutine will not be garbage collected; this is by design. This seems like the wrong behavior, but channels are for thread-safe communication. In that context, a hung goroutine trying to write to a channel that nobody will ever read from is probably a bug and not something you'd like to be silently garbage-collected.

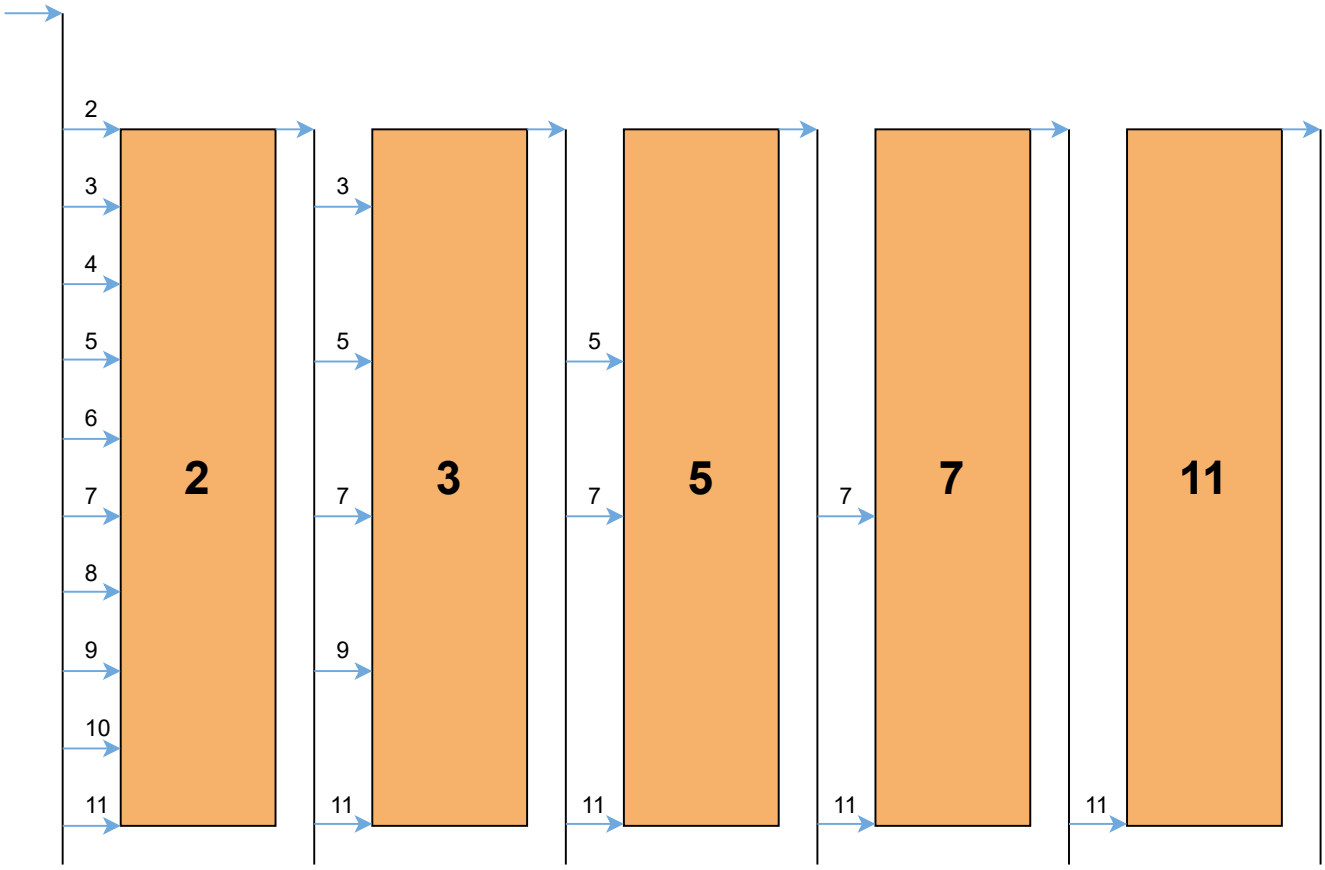
Pipe and filter pattern

A more concrete example would be a goroutine `processChannel`, which processes what it receives from an input channel and sends this to an output channel:

```
sendChan := make(chan int)
receiveChan := make(chan string)

go processChannel(sendChan, receiveChan)
func processChannel(in <-chan int, out chan<- string) {
 for inValue := range in {
   result:= ... // processing inValue
   out <- result
 }
}
```

By using the directionality notation, we make sure that the goroutine will not perform unallowed channel operations.

Here is an excellent and more concrete example taken from the Go Tutorial, which prints the prime numbers at its output, using filters (sieves) as its algorithm. Each prime gets its filter, like in this schema:

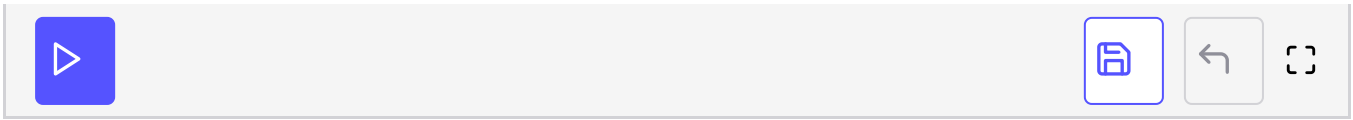


The Sieve-Prime Algorithm

Version I

```
package main
import "fmt"

// Send the sequence 2, 3, 4, ... to channel ch.
func generate(ch chan int) {
```

```
  for i := 2; ; i++ {
    ch <- i // Send i to channel ch.
  }

}

// Copy the values from channel in to channel out,
// removing those divisible by prime.
func filter(in, out chan int, prime int) {
  for {
    i := <-in // Receive value of new variable i from in.
    if i%prime != 0 {
      out <- i // Send i to channel out.
    }
  }
}

// The prime sieve
func main() {
  ch := make(chan int) // Create a new channel.
  go generate(ch) // Start generate() as a goroutine.
  for {
    prime := <-ch
    fmt.Print(prime, " ")
    ch1 := make(chan int)
    go filter(ch, ch1, prime)
    ch = ch1
  }
}
```



Sieve Algorithm-Version 1

> **Note:** The `Execution Timed Out!` is the expected behavior, so don't panic.

Look at the `main` function at **line 25**. It starts `generate` as its own goroutine. It is implemented at the top. Look at its header at **line 5**. It receives a channel `ch` and starts sending integers on it in an infinite loop (see **line** 7). Let's see the flow of the program in detail:

- In `main()` starts an infinite for-loop (from **line 26** to **line 32**). It takes a value from the channel named `prime` (see **line** 27), and prints it at **line 28**.
- A new channel `ch1` of integers is made at **line 29**. The `filter` function is started at **line 30** as its own goroutine, passing the *two* channels (becoming resp. `in` and `out`), and `prime` as a parameter.
- Then, in an infinite for-loop (from **line 14** to **line 19**) in `filter` function

- Then, in an infinite for loop (from **line 11** to **line 15**) in `filter` function, an integer `i` is taken from `in (ch)`, only when it is not divisible by the `prime` (**line 16**) and is copied to `out (ch1)`.
- Then, channel `ch` is replaced by `ch1` at **line 31**.

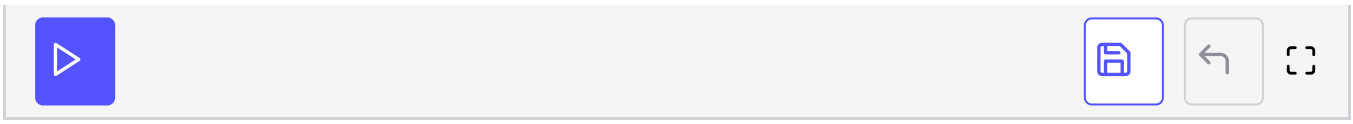
Thus, following the schema outlined above, the *net* result is that only prime numbers remain in `ch`.

Version II

In the second version, the *pipe and filter* pattern described above is applied. The functions `sieve`, `generate`, and `filter` are factories. They make a channel and return it, and they use lambda functions as goroutines. The main routine is now very short and clear. It calls `sieve()`, which returns a channel containing the primes, and then the channel is printed out via `fmt.Println(<-primes)`.

```
package main
import "fmt"

// Send the sequence 2, 3, 4, ... to returned channel
func generate() chan int {
  ch := make(chan int)
  go func() {
    for i := 2; ; i++ {
     ch <- i
    }
  }()
  return ch
}

// Filter out input values divisible by prime, send rest to returned channel
func filter(in chan int, prime int) chan int {
  out := make(chan int)
  go func() {
    for {
      if i := <-in; i%prime != 0 {
       out <- i
      }
    }
  }()
  return out
}

func sieve() chan int {
  out := make(chan int)
  go func() {
    ch := generate()
    for {
      prime := <-ch
      ch = filter(ch, prime)
```

```
      out <- prime
    }
  }()

  return out
}

func main() {
  primes := sieve()
  for {
    fmt.Println(<-primes)
  }
}
```



Sieve Algorithm-Version 2

> **Note:** The `Execution Timed Out!` is the expected behavior, so don't panic.

Here, we refactor the code of **version I**. The `main()` calls function `sieve()` at **line 42**, which produces the prime numbers as a channel of *ints* called `primes`. Then in an infinite for loop (from **line 43** to **line 45**), the values are read from the channel and printed out.

Now, look at the header of sieve() at **line 28**. It constructs its return channel at **line 29** and starts at **line 30** a goroutine, which executes an *anonymous* function. In it, we call the function `generate()` at **line 31**. This function is defined (from **line 5** to **line 13**) as in *Version I*; this puts all integers onto its output channel, which is captured in `ch`.

Then, in an infinite for loop, we take a value `prime` from `ch`, and pass it together with `ch` to function `filter()` at **line 34**. The `filter()` works like in *Version I*, making an `out` channel at **line 17**, then starting at **line 18** a goroutine, which executes an *anonymous* function. In it, enclosed in an infinite for loop, we take a value from the `in` channel (which was `ch`). Only when it is a prime, we pass it to the `out` channel at **line 21**, which is returned at **line 25**.

Now that you're familiar with channel directionality, the next lesson brings the information to synchronize the channels between goroutines.