Channel Factory and Producer-Consumer Pattern

This lesson provides detailed concepts on the channel factory and producer-consumer pattern.

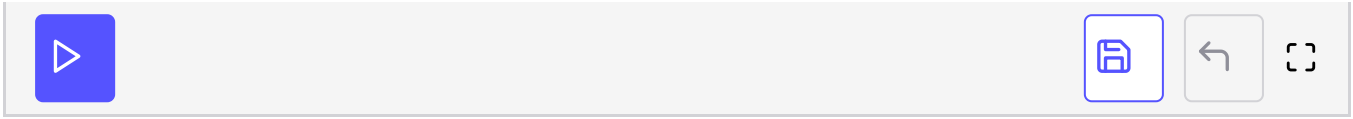
WE'LL COVER THE FOLLOWING

Channel factory pattern

Another common pattern in this style of programming is that, instead of passing a channel as a parameter to a goroutine, the function makes the channel and returns it (so it plays the role of a factory). Inside the function, a lambda function is called a goroutine. The following code is an implementation of this pattern:

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

func main() {
  stream := pump()
  go suck(stream)
  // the above 2 lines can be shortened to: go suck( pump() )
  time.Sleep(1e9)
}

func pump() chan int {
  ch := make(chan int)
  go func() {
    for i := 0; ; i++ {
      ch <- i
    }
  }()
  return ch
}

func suck(ch chan int) {
```

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

  }
}
```



Channel Factory Pattern

At **line 8**, the `main()` goroutine starts the function `pump()`. As we see from **line 14**, `pump()` returns a channel of *ints*, which is received in the `stream` variable.

Look at the header of `pump()` at **line 14**. It makes a local channel `ch` at **line 15** and then starts a goroutine in an anonymous function at **line 16**. This function executes an infinite for-loop at **line 17**, putting successive integers onto the channel. While this has started, `ch` is returned at **line 21**, and received in variable `stream`.

At **line 9**, a *second* goroutine is started, executing the `suck()` function. Look at the header of `suck()` at **line 24**. It takes `ch` as a parameter. This gets a value from the channel and prints it out. At **line 11**, `main()` waits 1 second to allow the display of the initial output. Then, the program exits, stopping all goroutines.

For-range applied to channels

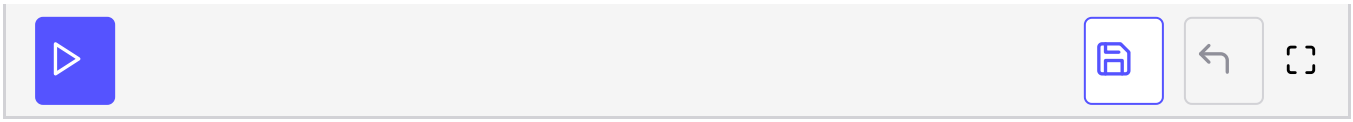
The `range` clause on for loops accepts a channel `ch` as an operand, in which case the for loops over the values received from the channel, like this:

```
for v := range ch {
 fmt.Printf("The value is %v\n",v)
}
```

It reads from the given channel `ch` until the channel is closed, and then the code following the for continues to execute. Obviously, another goroutine must be writing to `ch` (otherwise the execution blocks in the for-loop) and must close `ch` when it is done writing. The function `suck()` can apply this and also launch this action in a goroutine. Then, the former program becomes:

```
package main
import (
```

```
  "fmt"
  "time"
)

func main() {
  suck(pump())
  time.Sleep(1e9)
}

func pump() chan int {
  ch := make(chan int)
  go func() {
    for i := 0; ; i++ {
      ch <- i
    }
  }()
  return ch
}

func suck(ch chan int) {
  go func() {
    for v := range ch {
      fmt.Println(v)
    }
  }()
}
```



For range on channels

The logic of this program is nearly the same as the previous code. The `suck()` function calls `pump()` at **line 8**. This is possible because `suck` expects a channel of *ints* as a parameter, and `pump` returns a channel of *ints*. Now, we can make the design much more symmetrical: both `pump()` and `suck()` start a goroutine. The `pump()` (see implementation from **line 12** to **line 20**) is identical to the previous version. Now, the `suck()` starts an anonymous function in a goroutine. This function iterates over the channel `ch` (**line 24**), getting, reading, and printing out each successive value (**line 25**).

Producer-consumer pattern

Suppose we have a `Produce()` function, which delivers the values needed by a `Consume()` function. Both functions could be run as a separate goroutine, `Produce` putting the values on a channel which is read by `Consume`. The whole process could take place in an infinite loop:

```
for {
  Consume(Produce())
```

```
    consume(produce())
}
```

Now that you're familiar with the different patterns, the next lesson brings you a challenge to solve.