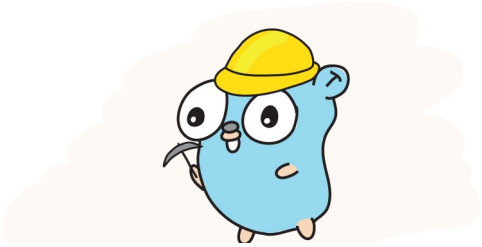
Trevor Forrey
I'm a master's software engineering student at ASU. I love learning about new tech along with combining ...
May 16

Learning Go's Concurrency Through Illustrations

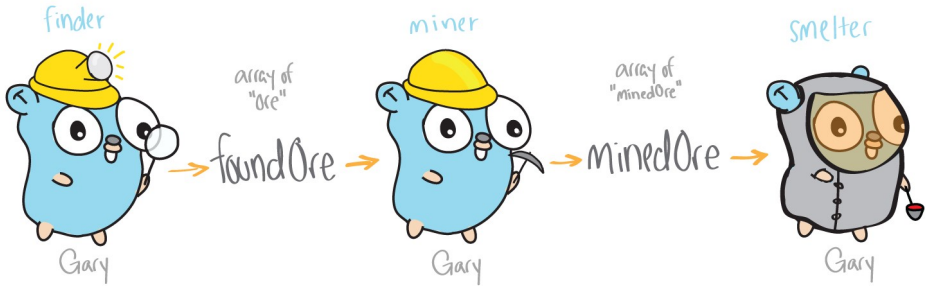
You've most likely heard of Go in one way or another. It's been increasing in popularity, and for good reason. Go is fast, simple, and has a great community behind it. One of the most exciting aspects of learning the language is its concurrency model. Go's concurrency primitives make creating concurrent, multi-threaded programs simple and fun. I'll be introducing Go's concurrency primitives through illustrations in hopes that it'll make these concepts click for future learning. This article is meant for those who're new to Go and want to start learning about Go's concurrency primitives: go routines and channels.

Single-threaded vs. Multi-threaded Programs

You've probably written multiple single-threaded programs before. A common pattern in programming is having multiple functions that perform a specific task, but they don't get called until a previous part of the program gets data ready for the next function.



This is how we'll initially set up our first example, a program that mines ore. The functions in this example perform: *finding ore*, *mining ore*, and *smelting ore*. In our example, the mine and ore are represented as an array of strings, with each function taking in and returning a "processed" array of strings. For a single-threaded application, the program would be designed as follows.



There are 3 main functions. A *finder*, a *miner*, and a *smelter*. In this version of the program, our functions run on a single thread, one right after the other—and this single thread (the gopher named Gary) would need to do all the work.

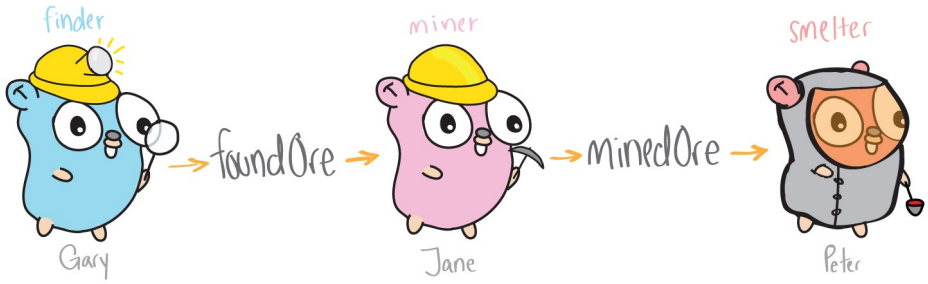
```
func main() {
 theMine := [5]string{"rock", "ore", "ore", "rock",
"ore"}
 foundOre := finder(theMine)
 minedOre := miner(foundOre)
 smelter(minedOre)
}
```

Printing out the resulting array of "ore" at the end of every function, we get the following output:

```
From Finder: [ore ore ore]
```

```
From Miner: [minedOre minedOre minedOre]

From Smelter: [smeltedOre smeltedOre smeltedOre]
```

This style of programming has the benefits of being easy to design, but what happens when you want to take advantage of multiple threads and perform functions independent of each other? This is where concurrent programming comes into play.



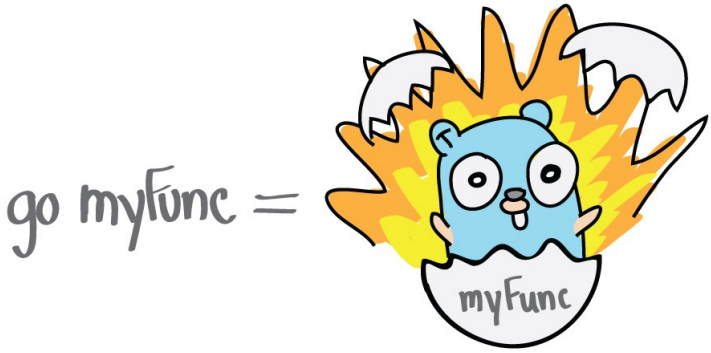
This mining design is much more efficient. Now multiple threads (gophers) are working independently; therefore, the whole operation isn't all on Gary. There's a gopher finding the ore, one mining the ore, and another smelting the ore—potentially all at the same time.

In order for us to bring this type of functionality into our code we're going to need two things: a way to create independently working gophers, and a way for gophers to communicate (*send ore)* to each other. This is where Go's concurrency primitives come in: goroutines and channels.

Go routines

Go routines can be thought of as lightweight threads. Creating a go routine is as easy as adding *go* to the start of calling a function. For an example of just how easy it is, lets create two finder functions, call them using the *go* keyword,

and have them print out every time they find “ore” in their mine.




```
func main() {
 theMine := [5]string{“rock”, “ore”, “ore”, “rock”,
“ore”}
 go finder1(theMine)
 go finder2(theMine)
 <-time.After(time.Second * 5) //you can ignore this
for now
}
```

Here’s the output from our program:

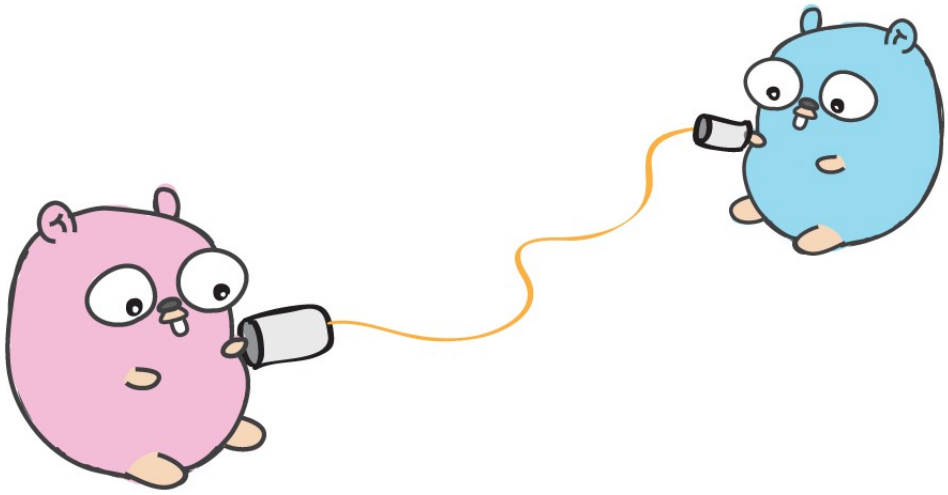
```
Finder 1 found ore!
Finder 2 found ore!
Finder 1 found ore!
Finder 1 found ore!
Finder 2 found ore!
Finder 2 found ore!
```

As you can see from the output above, the finders are running concurrently. There’s no real order in who finds ore first, and when ran multiple times, the order isn’t always the same.

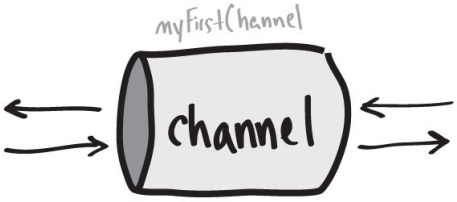
This is great progress! Now we have an easy way to set up a multi-threaded (multi-gopher) program, but what happens

when we need our independent go routines to communicate to each other? Welcome to the magical world of *channels*.

Channels

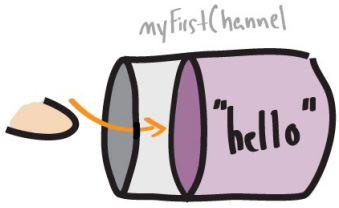


Channels allow go routines to communicate with each other. You can think of a channel as a pipe, from which go routines can send and receive information from other go routines.



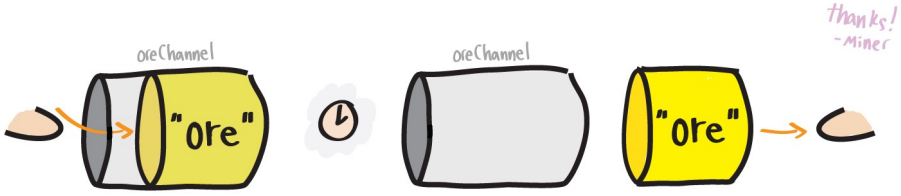

```
myFirstChannel := make(chan string)
```

Go routines can *send* and *receive* on a channel. This is done through using an arrow (<-) that points in the direction that the data is going.




```
myFirstChannel <- "hello" // Send
myVariable := <- myFirstChannel // Receive
```

Now by using a channel, we can have our ore finding gopher send what they discover to our ore breaking gopher right away, without waiting to discover everything.



I've updated the example so the finder code and miner functions are set up as unnamed functions. If you've never seen lambda functions don't focus too much on that part of the program, just know that each of the functions are being called with the *go* keyword so they're being ran on their own go routine. What's important is to notice how the go routines are passing data between each other using the channel, *oreChan*. *Don't worry, I'll explain unnamed functions at the end.*

```
func main() {
 theMine := [5]string{“ore1”, “ore2”, “ore3”}
```

```
 oreChan := make(chan string)

 // Finder
 go func(mine [5]string) {
  for _, item := range mine {
   oreChan <- item //send
  }
 }(theMine)

 // Ore Breaker
 go func() {
  for i := 0; i < 3; i++ {
   foundOre := <-oreChan //receive
   fmt.Println("Miner: Received " + foundOre + " from
finder")
  }
 }()
 <-time.After(time.Second * 5) // Again, ignore this
for now
}
```

In the output below, you can see that our Miner receives the pieces of "ore" one at a time from reading off the ore channel three times.

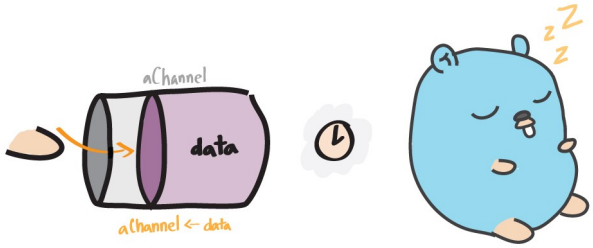
```
Miner: Received ore1 from finder

Miner: Received ore2 from finder

Miner: Received ore3 from finder
```

Great, now we can send data between different go routines (gophers) in our program. Before we start writing complex programs with channels, lets first cover some crucial to understand channel properties.

Channel Blocking

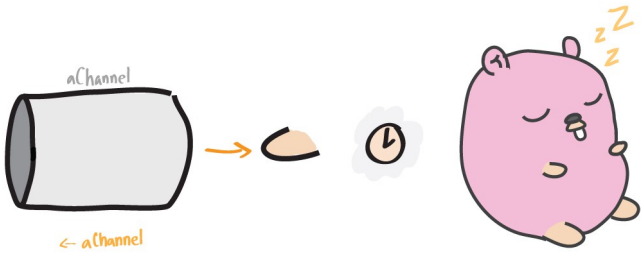
Channels block go routines in various situations. This allows our go routines to sync up with each other for a moment, before going on their independently merry way.

Blocking on a Send



Once a go routine (gopher) sends on a channel, the sending go routine blocks until another go routine receives what was sent on the channel.

Blocking on a Receive

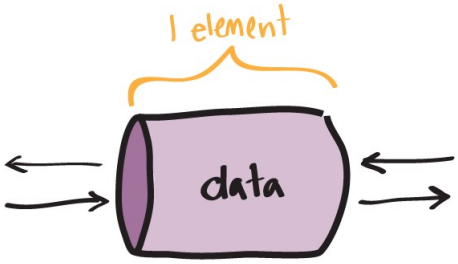


Similar to blocking after sending on a channel, a go routine can block waiting to get a value from a channel, with nothing sent to it yet.

Blocking can be a bit confusing at first, but you can think of it like a transaction between two go routines (gophers). Whether a gopher is waiting for money or sending money, it will wait until the other partner in the transaction shows up.

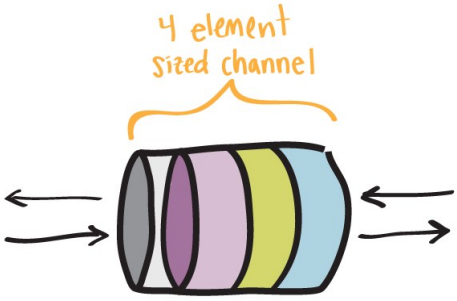
Now that we have an idea on the different ways a go routine can block while communicating through a channel, lets discuss the two different types of channels: *unbuffered,* and *buffered*. Choosing what type of channel you use can change how your program behaves.

Unbuffered Channels



We've been using unbuffered channels in all previous examples. What makes them unique is that only one piece of data fits through the channel at a time.

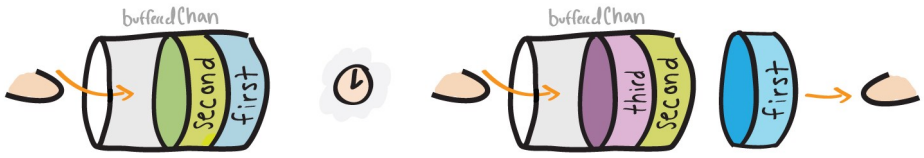
Buffered Channels



In concurrent programs, timing isn't always perfect. In our mining example, we could run into a situation where our finding gopher can find 3 pieces of ore in the time it takes the breaking gopher to process one piece of ore. In order to not let the surveying gopher spend most of its time waiting to send the breaking gopher some ore until it finishes, we can use a *buffered* channel. Lets start by making a buffered channel with a capacity of 3.

```
bufferedChan := make(chan string, 3)
```

Buffered channels work similar to unbuffered channels, but with one catch—we can send multiple pieces of data to the channel before needing another go routine to read from it.




```
bufferedChan := make(chan string, 3)

go func() {
 bufferedChan <- "first"
 fmt.Println("Sent 1st")
 bufferedChan <- "second"
 fmt.Println("Sent 2nd")
 bufferedChan <- "third"
 fmt.Println("Sent 3rd")
}()

<-time.After(time.Second * 1)

go func() {
 firstRead := <- bufferedChan
 fmt.Println("Receiving..")
 fmt.Println(firstRead)
 secondRead := <- bufferedChan
 fmt.Println(secondRead)
 thirdRead := <- bufferedChan
 fmt.Println(thirdRead)
}()
```

The order of printing between our two go routines would be:

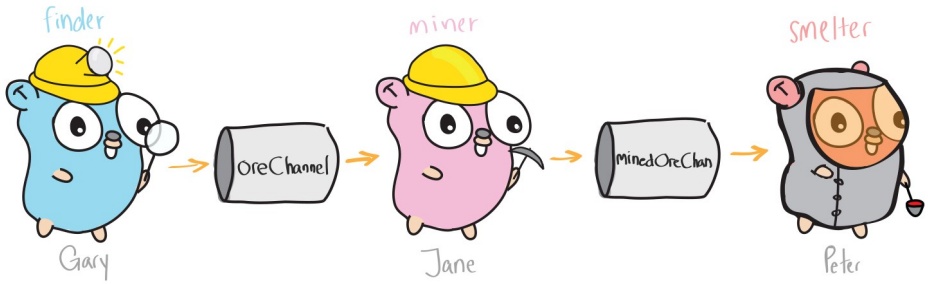
```
Sent 1st
Sent 2nd
Sent 3rd
Receiving..
first
second
third
```

To keep things simple, we won't be using buffered channels in our final program, but it's important to know what types of channels are available in your concurrency tool belt.

> *Note: Using buffered channels doesn't prevent blocking from happening. For example, if the finding gopher is 10 times faster than the breaker, and they communicate through a buffered channel of size 2, the finding gopher will still block multiple times in the program.*

Putting it all Together

Now with the power of go routines and channels, we can write a program that takes full advantage of multiple threads using Go's concurrency primitives.




```
theMine := [5]string{"rock", "ore", "ore", "rock",
"ore"}
oreChannel := make(chan string)
minedOreChan := make(chan string)

// Finder
go func(mine [5]string) {
 for _, item := range mine {
  if item == "ore" {
   oreChannel <- item //send item on oreChannel
  }
 }
}(theMine)

// Ore Breaker
go func() {
```

```
 for i := 0; i < 3; i++ {
  foundOre := <-oreChannel //read from oreChannel
  fmt.Println("From Finder: ", foundOre)
  minedOreChan <- "minedOre" //send to minedOreChan
 }
}()

// Smelter
go func() {
 for i := 0; i < 3; i++ {
  minedOre := <-minedOreChan //read from minedOreChan
  fmt.Println("From Miner: ", minedOre)
  fmt.Println("From Smelter: Ore is smelted")
 }
}()

<-time.After(time.Second * 5) // Again, you can ignore
this
```

The output of this program is the following:

```
From Finder:  ore

From Finder:  ore

From Miner:  minedOre

From Smelter: Ore is smelted

From Miner:  minedOre

From Smelter: Ore is smelted

From Finder:  ore

From Miner:  minedOre

From Smelter: Ore is smelted
```

This has been a great improvement from our original example! Now each of our functions are running independently on their own go routines. Also, every time

there's a piece of ore processed, it gets carried on to the next stage of our mining line.

For the sake of keeping the focus on understanding the basics of channels and go routines, there was some important information I didn't mention above- which, if you don't know, could cause some trouble when you start programming. Now that you have an understanding of how go routines and channels work, let's go over some information you should know before you start coding with go routines and channels.

Before you go, you should know..

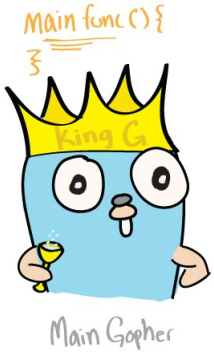
Anonymous Go Routines

Similar to how we can set up a function to run on its own go routine using the *go* keyword, we can create an anonymous function to run on it's own go routine using the following format:

```
// Anonymous go routine
go func() {
 fmt.Println("I'm running in my own go routine")
}()
```

This way, if we only need to call a function once, we can place it on its own go routine to run, without worrying about creating an official function declaration.

The main function is a go routine



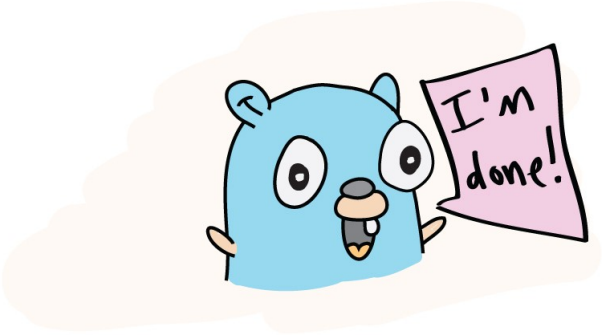
The main function indeed runs in its own go routine! Even more important to know is that once the main function returns, it closes all other go routines that are currently running. This is why we had a timer at the bottom of our main function—which created a channel and sent a value on it after 5 seconds.

```
<-time.After(time.Second * 5) //Receiving from channel
after 5 sec
```

Remember how a go routine will block on a read until something is sent? That's exactly what is happening to the main routine by adding this code above. The main routine will block, giving our other go routines 5 seconds of additional life to run.

Now there are much better ways to handle blocking the main function until all other go routines are complete. A common practice is to create a *done channel* which the main function

blocks on waiting to read. Once you finish your work, write to this channel, and the program will end.




```
func main() {
 doneChan := make(chan string)

 go func() {
  // Do some work…
  doneChan <- "I'm all done!"
 }()

 <-doneChan // block until go routine signals work is
done
}
```

You can range over a channel

In a previous example we had our miner reading from a channel in a for loop that went through 3 iterations. What would happen if we didn't know exactly how many pieces of ore would come from the finder? Well, similar to doing ranges over collections, you can *range over a channel*.

Updating our previous miner function, we could write:

```
 // Ore Breaker
 go func() {
  for foundOre := range oreChan {
   fmt.Println("Miner: Received " + foundOre + " from
finder")
```

```
  }
 }()
```

Since the miner needs to read everything that the finder sends him, ranging over the channel here makes sure we receive everything that gets sent.

> *Note: Ranging over a channel will block until another item is sent on the channel. The only way to stop the go routine from blocking after all sends have occurred is by closing the channel with 'close(channel)'*

You can make a non-blocking read on a channel

But you just told us all about how channels block go routines?! True, but there is a technique where you can make a non-blocking read on a channel, using Go's *select case* structure. By using the structure below, your go routine will read from the channel if there's something there, or run the default case.

```
myChan := make(chan string)

go func(){
 myChan <- "Message!"
}()

select {
 case msg := <- myChan:
  fmt.Println(msg)
 default:
  fmt.Println("No Msg")
}

<-time.After(time.Second * 1)

select {
 case msg := <- myChan:
  fmt.Println(msg)
 default:
```

```
  fmt.Println("No Msg")
}
```

When ran, this example has the following output:

```
No Msg
Message!
```

You can also do non-blocking sends on a channel

Non-blocking sends use the same *select case* structure to perform their non-blocking operations, the only difference is our case would look like a send rather than a receive.

```
select {
 case myChan <- "message":
  fmt.Println("sent the message")
 default:
  fmt.Println("no message sent")
}
```

Where to learn next

There are numerous talks and blog posts that cover channels & go routines in much more detail. Now that you have a solid understanding of the purpose and application of these tools, you should be able to get the most out of the following articles and talks.

> *Google I/O 2012—Go Concurrency Patterns*
>
> *Rob Pike—'Concurrency Is Not Parallelism'*
>
> *GopherCon 2017: Edward Muller—Go Anti-Patterns*

Thanks for taking the time to read this. I hope you were able to learn about go routines, channels, and the benefits they bring to writing concurrent programs.

Golang Go Programming Learning To Code

Golang Tutorial