Learning Go

Through Illustrations

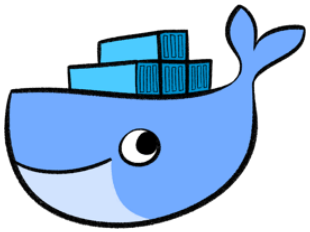
Trevor Forrey

Go What?

Developed at Google in 2009

What makes it so great?
- Opinionated
- Dependencies built-in binaries
- Concurrency Primitives
- Amazing Community

Go Where?



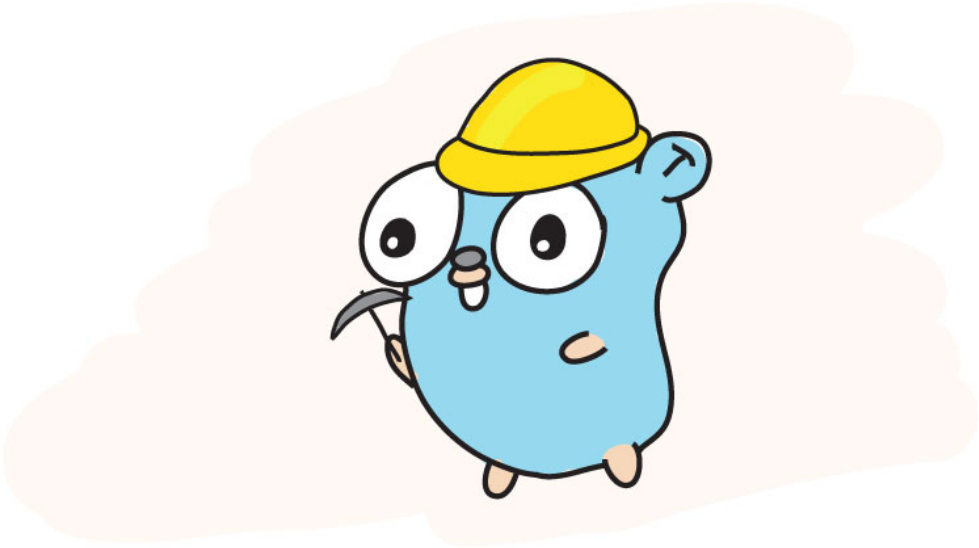
kubernetes

moz://a

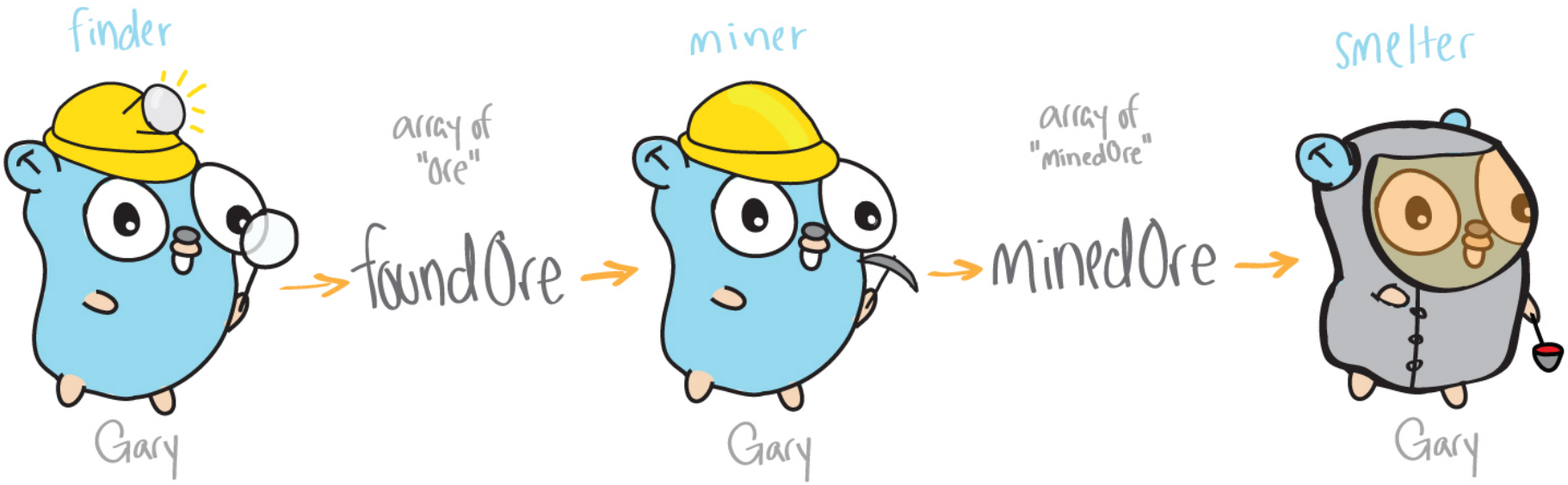
twitch

Background - Single Threaded vs. Multi Threaded

- Single Threaded: One function after another
- Multi-Threaded: independent functions share resources



One Gopher Architecture

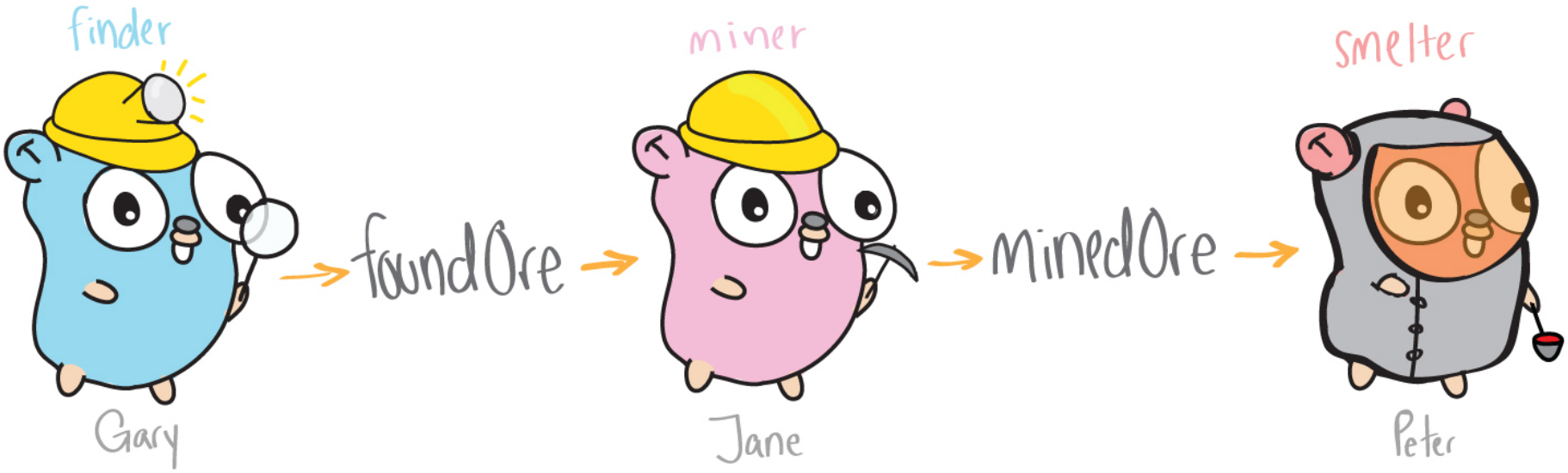


One Gopher Code

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

Run

Multi Gophered Architecture



Tools We'll Need

- A way to create Gophers
- A way to communicate between Gophers

Go Routines

go myFunc =

Two Gophers, One Mine

```
func main() {
    theMine := [5]string{"rock", "ore", "ore", "rock", "ore"}
    go finder(theMine, 1)
    go finder(theMine, 2)
    <-time.After(time.Second * 5) //you can ignore this for now
}
```

Run

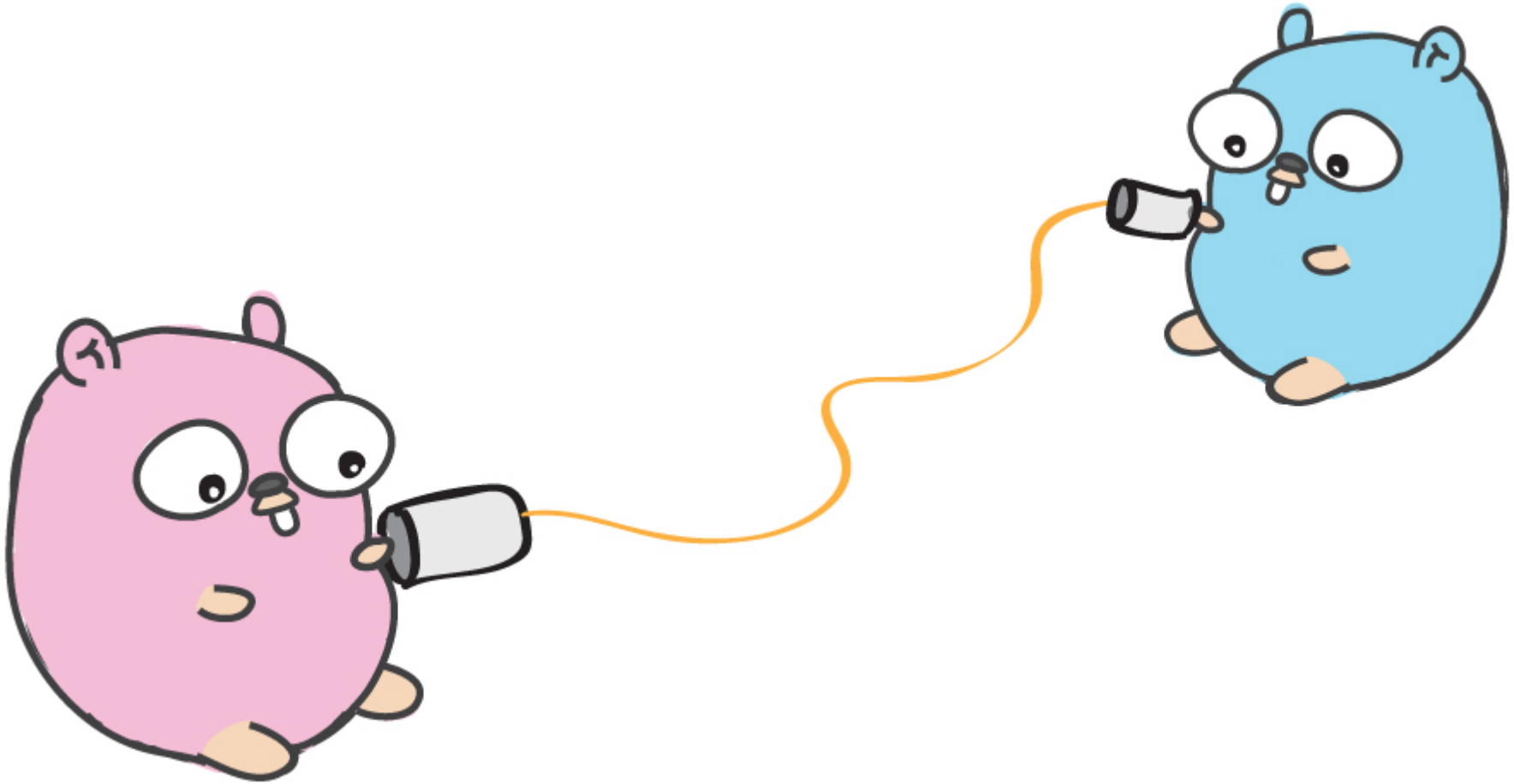
What if we want to be sneaky? (Anonymous Go Routines)


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

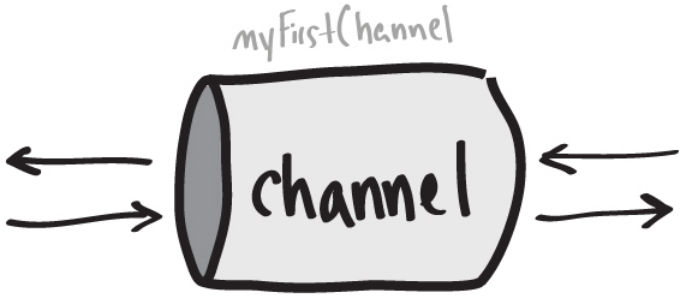
Current Status

- Start up go routines easily
- Even more lightweight than a normal 'thread'
- But how will we communicate between gophers?

Channels

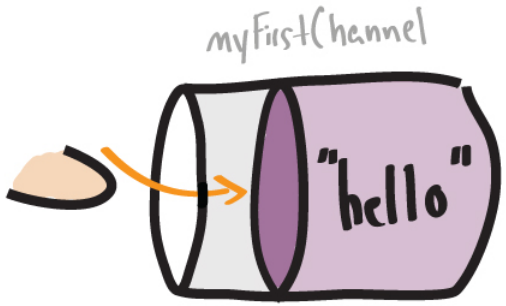


Channel Creation




```
func main() {
    myFirstChannel := make(chan string)
}
```

Sending / Receiving on Channels




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

    myFirstChannel <- "hello"      // Send
    myVariable := <-myFirstChannel // Receive
    <-myFirstChannel               // Receive and discard result
}
```

Example of sending / receiving on a channel

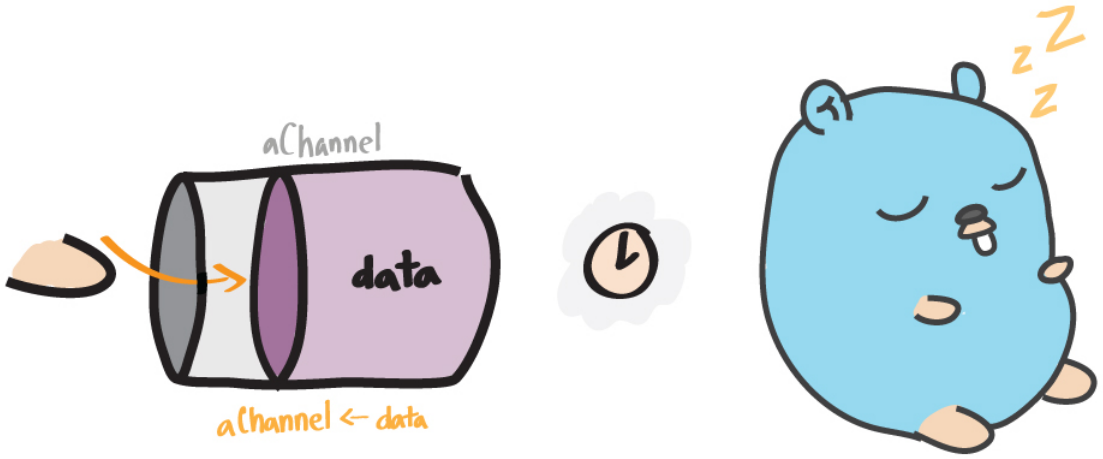
```
func main() {
    myFirstChannel := make(chan string)
    go func() {
        // send on channel
        myFirstChannel <- "hello"
    }()
    message := <-myFirstChannel
    fmt.Printf("Received: %v", message)
}
```

Run

Channel Blocking

- Allows go routines to 'sync' back up

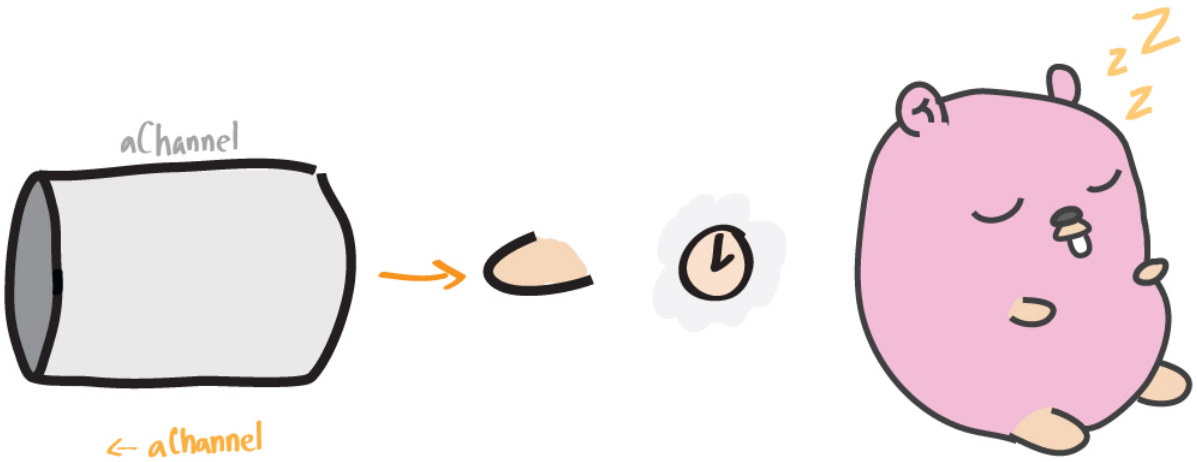
Blocking on a Send




```
func main() {
    myFirstChannel := make(chan string)
    myFirstChannel <- "hello"
}
```

Run

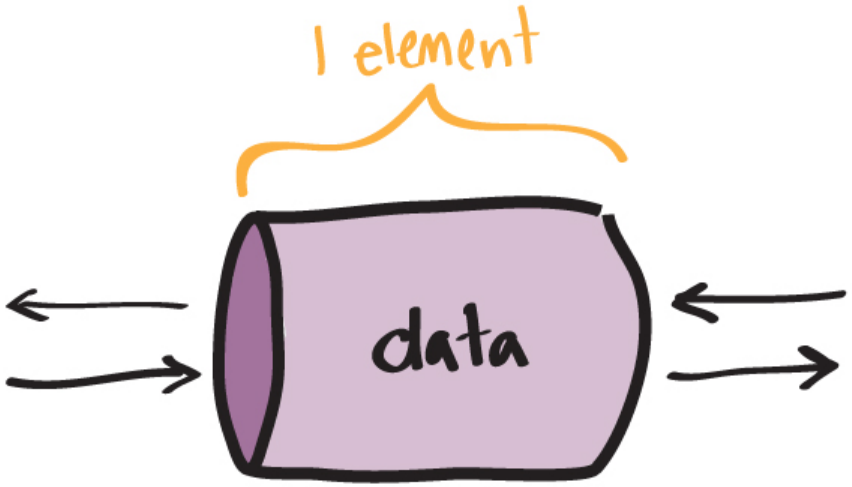
Blocking on a Receive




```
func main() {
    myFirstChannel := make(chan string)
    message := <-myFirstChannel
    fmt.Printf("Message: %v", message)
}
```

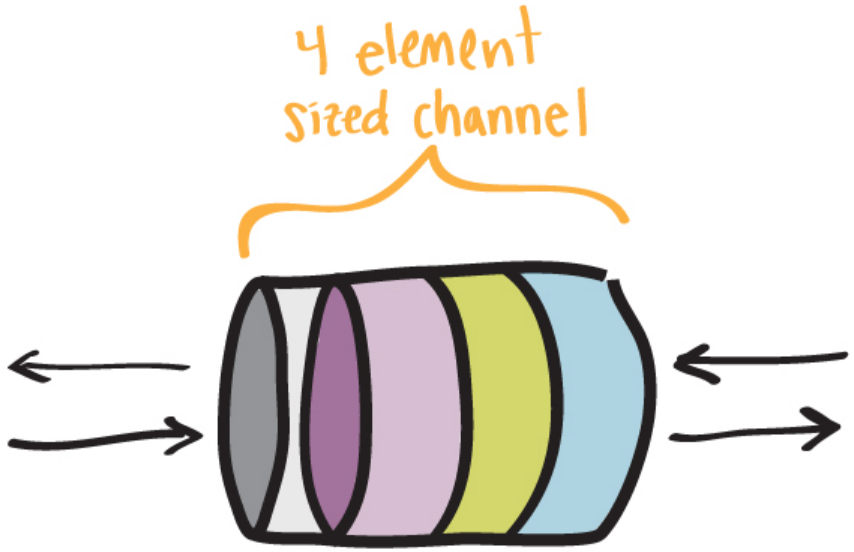
Run

Unbuffered Channels



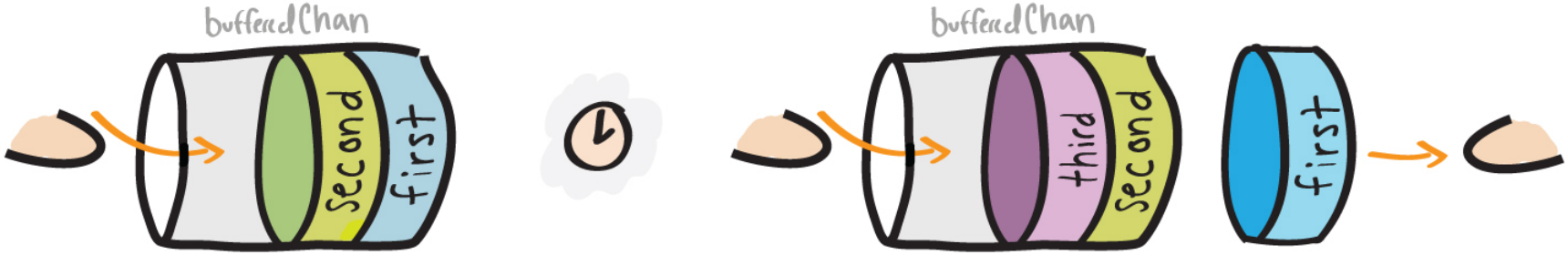

```
func main() {
    myFirstChannel := make(chan string)
}
```

Buffered Channels




```
func main() {
    bufferedChan := make(chan string, 4)
}
```

Buffered Channels cont



Now we don't block on every channel write/read!

Buffered Channels Example

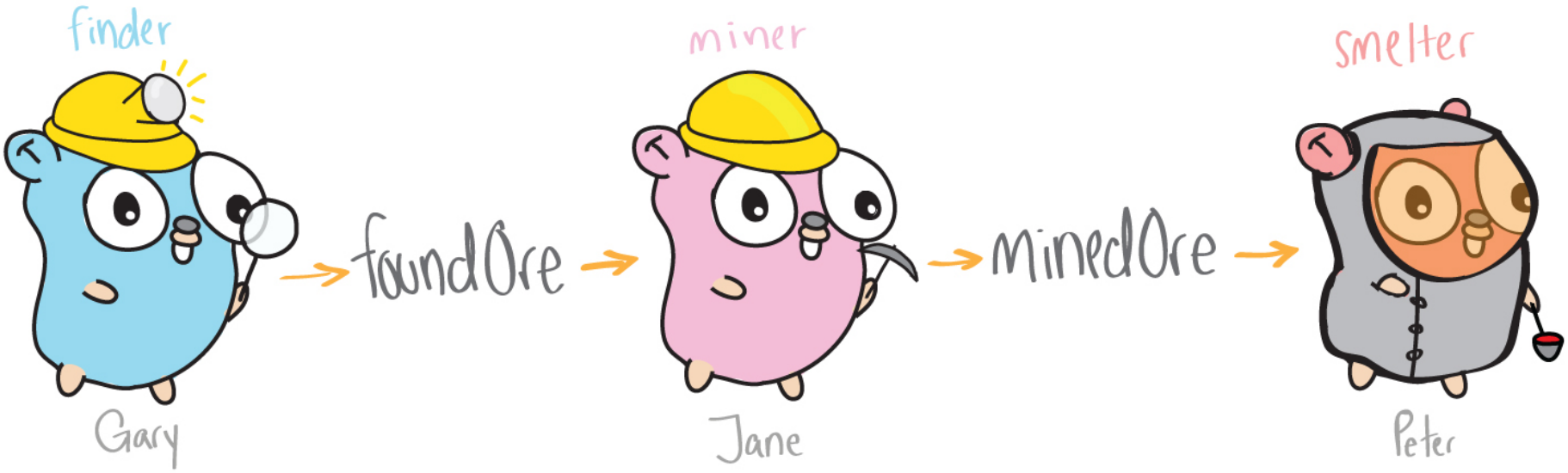
```
func main() {
    dataArray := [7]string{"1", "2", "3", "4", "5", "6", "7"}
    myFirstChannel := make(chan string, 3)
    go func() {
        for _, item := range dataArray {
            myFirstChannel <- item
            fmt.Printf("Sent: %v\n", item)
        }
        close(myFirstChannel)
    }()
    for item := range myFirstChannel {
        <-time.After(1 * time.Second)
        fmt.Printf("Received: %v\n", item)
    }
}
```

Run

Current Status

- Create Go Routines (Gophers)
- Create Channels
- You are all Masters of Concurrency!

Desired Pipeline



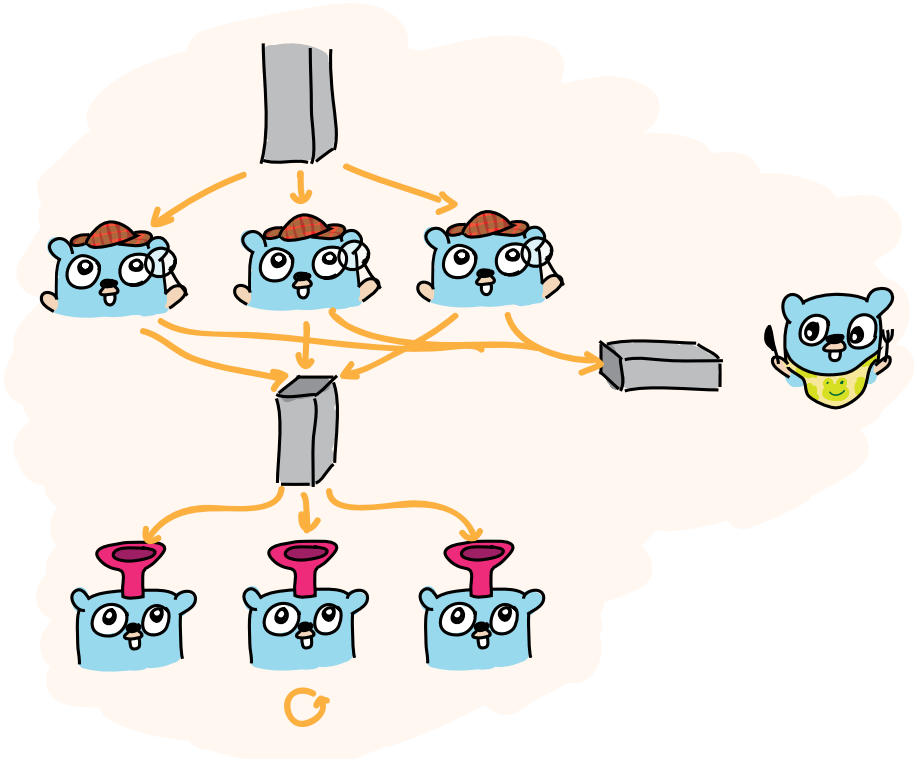
Final Pipeline

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

    go func() {
        for _, item := range theMine {
            if item == "ore" {
                foundOre <- item
            }
        }
        close(foundOre)
    }()
    go func() {
        for ore := range foundOre {
            fmt.Printf("Got: %v\n", ore)
            minedOre <- "minedOre"
        }
        close(minedOre)
    }()

    for processedOre := range minedOre {
        fmt.Printf("Got: %v!\n", processedOre)
    }
}
```

Run

But what if I need to have more gophers (for speed of course)?!

Let's get fancy....by passing and returning channels



Passing in / returning a channel

```
func gen(mine [5]string) <-chan string {
    out := make(chan string)
    go func() {
        // Write to output channel
    }()
    return out
}
```

Passing in / returning a channel

```
func gen(mine [5]string) <-chan string {
    out := make(chan string)
    go func() {
        for _, item := range mine {
            out <- item
        }
        close(out)
    }()
    return out
}
```

Passing in / returning a channel contd

```
func finder(mineChan <-chan string) <-chan string {
    foundOreChan := make(chan string)
    go func() {
        for item := range mineChan {
            if item == "ore" {
                foundOreChan <- item
                fmt.Println("Finder found ore")
            }
        }
        close(foundOreChan)
    }()
    return foundOreChan
}
```

Passing in / returning a channel example

```
func main() {
    baseData := [5]string{"rock", "ore", "ore", "rock", "ore"}
    generatedChannel := gen(baseData)
    outputChannel := finder(generatedChannel)
    for ore := range outputChannel {
        fmt.Printf("%v found!\n", ore)
    }
}
```

Run

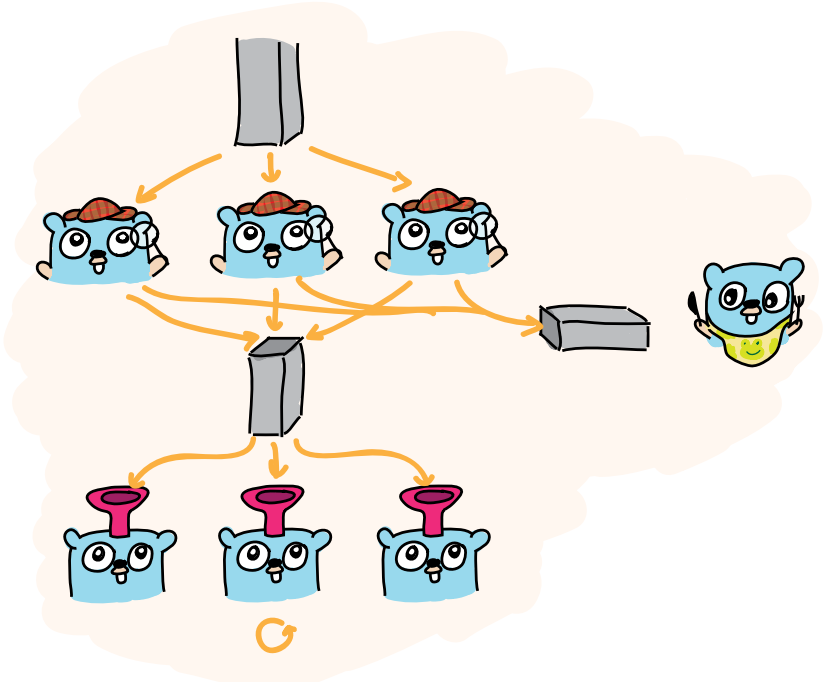
Full Program

```
func main() {
    theMine := [5]string{"rock", "ore", "ore", "rock", "ore"}
    mineChan := gen(theMine)
    foundOreChan := finder(mineChan)
    minedOreChan := miner(foundOreChan)
    smeltedOreChan := smelter(minedOreChan)
    for smeltedOre := range smeltedOreChan {
        fmt.Printf("%v processed\n", smeltedOre)
    }
}
```

Run

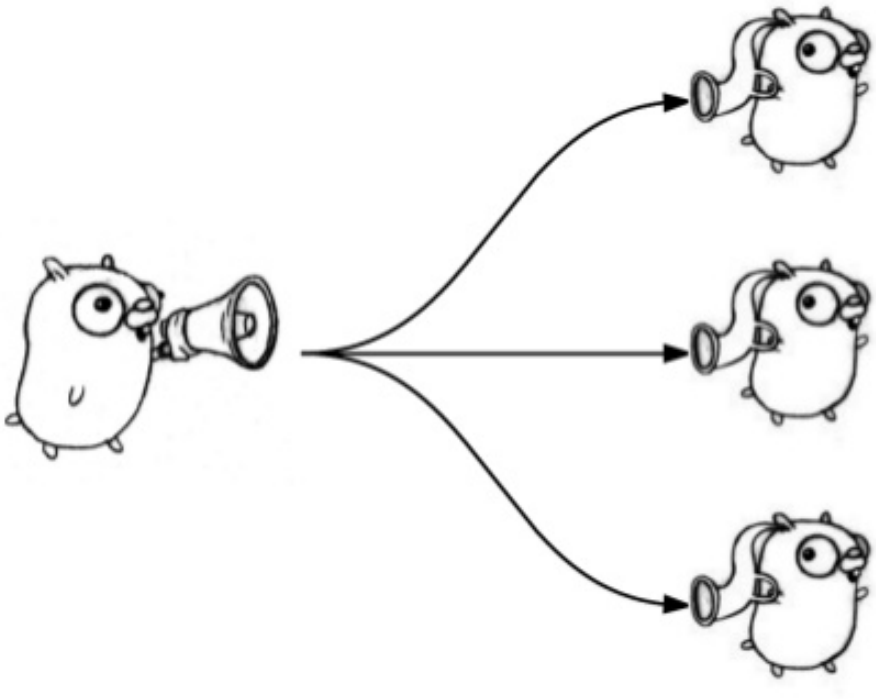
What's the Point though?

If we can pass in and return channels, we can share channels between multiple Go routines (Gophers).

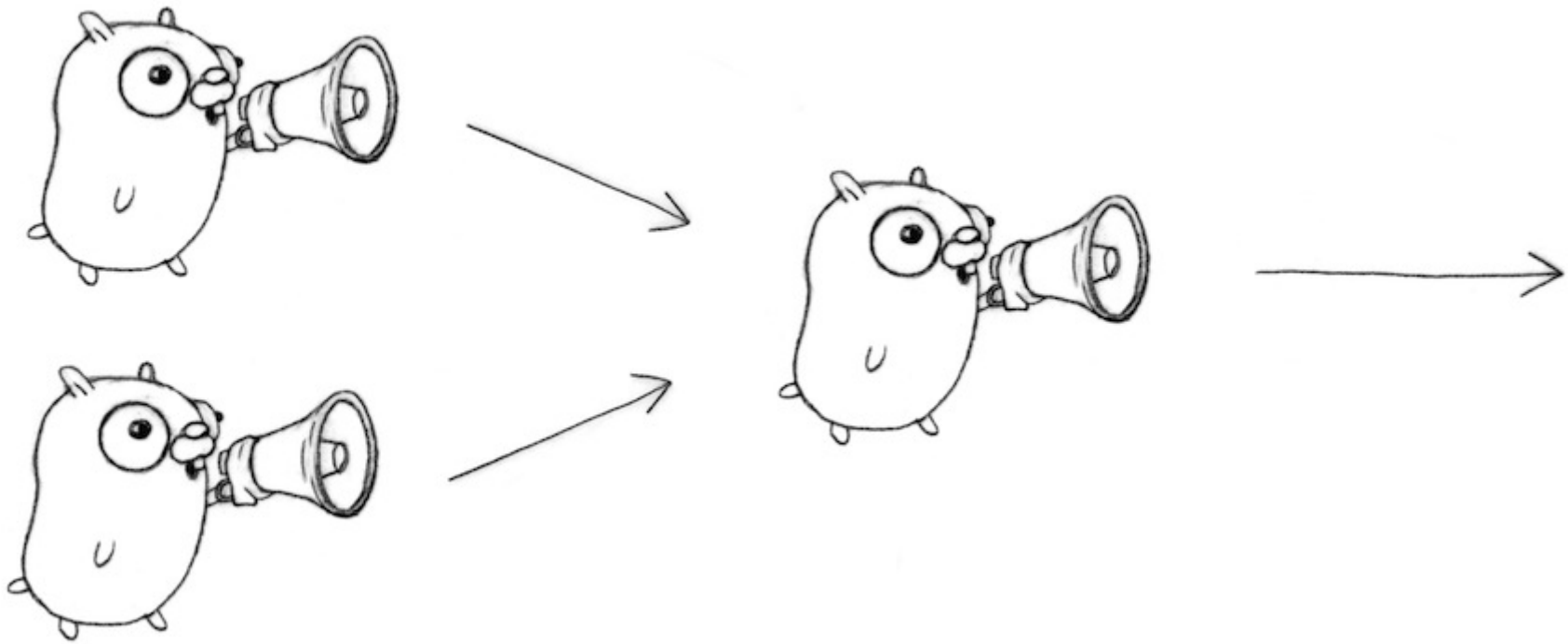


- Concurrency Pattern: Fan In / Fan Out

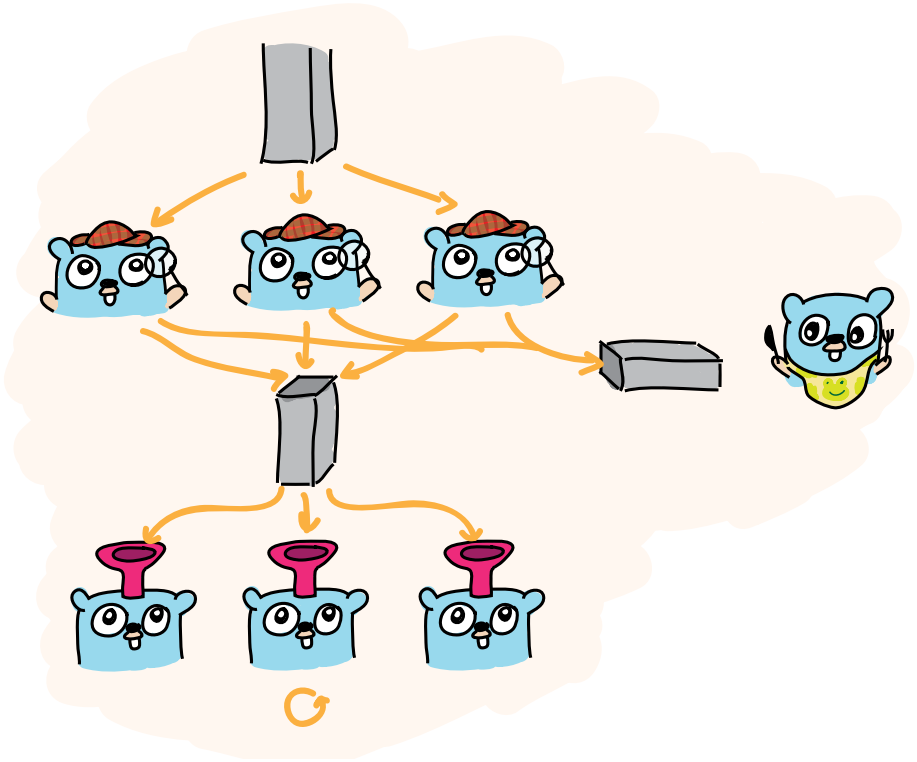
Fan Out



Fan In



Now we can do stuff like this



Before You Go...

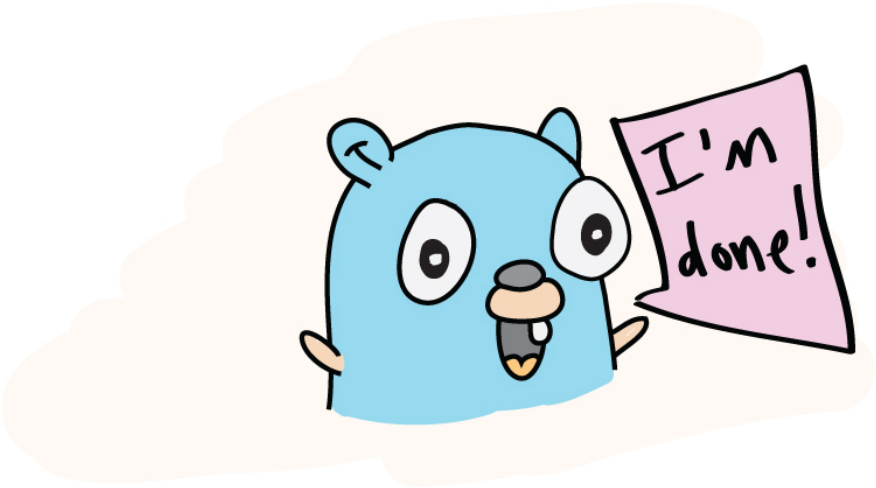
You should know!

Main Function is a Go Routine


```
func main() {
    go func() {
        fmt.Printf("Hi there!")
    }()
}
```

Run

Better ways than time out




```
func main() {
    doneChan := make(chan string)
    go func() {
        fmt.Printf("Hi there!\n")
        fmt.Printf("Hello?!")
        doneChan <- "I'm all done!"
    }()

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

Run

Non-blocking reads

```
func main() {
    myChannel := make(chan string)
    go func() {
        myChannel <- "message received!\n"
    }()

    select {
    case msg := <-myChannel:
        fmt.Printf("Received: %v", msg)
    default:
        fmt.Printf("no message\n")
    }

    select {
    case msg := <-myChannel:
        fmt.Printf("Received: %v", msg)
    default:
        fmt.Printf("no message")
    }
}
```

Run

Non-blocking writes

```
func main() {
    myChannel := make(chan string)
    go func() {
        msg := "message"
        select {
        case myChannel <- msg:
            fmt.Printf("sent: %v\n", msg)
        default:
            fmt.Printf("no one home to receive\n")
        }
        close(myChannel)
    }()

    <-time.After(time.Second * 2)
}
```

Run

Where to learn next

Go by Example (https://gobyexample.com/)

Rob Pike - 'Concurrency Is Not Parallelism' (https://www.youtube.com/watch?v=cN_DpYBzKso)

Google I/O 2012 — Go Concurrency Patterns (https://www.youtube.com/watch?v=f6kdp27TYZs&t=938s)

Thank you

Trevor Forrey
http://trevorforrey.com (http://trevorforrey.com)
@tforrey (http://twitter.com/tforrey)