Training 3 - Concurrency

Calvin Lei-Cramer
May 2025

What are we covering today?

Panic and recover, goroutines and concurrency, channels

Panic! Recover.

- **What are panics actually? Are they exceptions?**
- **What triggers panics?**
  - Array/slice index out of bounds
  - Divide by zero
  - Calling `panic()`
  - Nil pointer deference
  - Type assertion (without using ‘ok’ second return value) failing

```
1  func main() {
2      _ = []int{1, 2, 3}[10]
3
4      div := 0
5      _ = 5 / div
6
7      panic("Something bad happened!")
8
9      var numPtr *int = nil
10     _ = *numPtr
11
12     var something any = struct{ A int }{A: 1000}
13     _ = something.(string)
14 }
```

Panic! Recover.

- **How to catch panics?**
- **`recover()` global**
  - Returns value passed to panic() if panicking
  - Otherwise returns nil
- **Named return values, defer, recover()**

```
$ go run .
25 / 2 =  12
43 / 0 =  9223372036854775807
-96 / 0 =  -9223372036854775808
$
```

```go
func SafeDivide(num, div int) (result int) {
    defer func() {
        if recover() != nil {
            // Set to near infinity
            if num < 0 {
                result = math.MinInt
            } else {
                result = math.MaxInt
            }
        }
    }()
    result = num / div // may cause panic
    return
}
```

Panic! Recover.

- **Pattern to recover normal execution flow like try/catch**

Python

```python
def main():
    print("Before")
    try:
        badFunction()
    except Exception as e:
        pass
    print("After")
```

```
$ go run .
Before
Log panic: something bad happened in badFunc()
After
```

Golang

```go
func handlePanic(action func()) {
    defer func() {
        if val := recover(); val != nil {
            fmt.Println("Log panic:", val)
        }
    }()
    action()
}

func main() {
    fmt.Println("Before")
    handlePanic(badFunc)
    fmt.Println("After")
}
```

Panic! Recover.

- **What happens if panic, then panic again in defer function?**

```go
func main() {
    defer func() {
        panic("panic #2")
    }()
    panic("panic #1")
}
```

```
$ go run .
panic: panic #1
        panic: panic #2

goroutine 1 [running]:
main.main.func1()
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:5 +0x25
panic({0x46f5a0?, 0x499a78?})
        /usr/local/go/src/runtime/panic.go:792 +0x132
main.main()
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:7 +0x3e
exit status 2
$ _
```

Panic! Recover.

```
3 func doStuff() {
4     defer func() {
5         if val := recover(); val != nil {
6             fmt.Println("Recovered panic:", val)
7         }
8     }()
9     defer func() {
10        panic("panic #2")
11    }()
12    panic("panic #1")
13 }
14
15 func main() {
16    fmt.Println("Before")
17    doStuff()
18    fmt.Println("After")
19 }
```

- **Panic, panic, recover?**

```
$ go run .
Before
Recovered panic: panic #2
After
$
```

Panic overrides any current panic value.

Keep panic recover simple.

Golang Idioms

- Idiomatic Go, mantras, patterns, group think
- “Programming in a language vs programming into a language” – Code Complete by Steve McConnell
- Error handling → errors are values, don’t code by exception
- Defer
- Goroutines and concurrency models

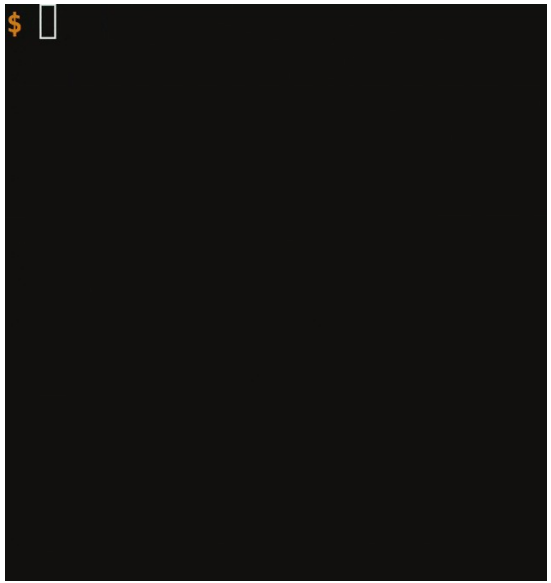
```
3  func fetch(url string) (body string, err error) {
4      resp, err := http.Get(url)
5      if err != nil {
6          return
7      }
8      defer resp.Body.Close()
9      bodyBytes, err := io.ReadAll(resp.Body)
10     if err != nil {
11         return
12     }
13     body = string(bodyBytes)
14     return
15 }
16
17 func main() {
18     resp, err := fetch("https://www.google.com")
19     if err != nil {
20         fmt.Println("error fetching URL -", err)
21         return
22     }
23     fmt.Println(resp)
24 }
```

Goroutines

- **What are goroutines?**
  - Lightweight application-level threads managed by Go runtime
  - Multiple goroutines may run on a single OS-level thread (concurrency) or multiple OS-level threads (parallelism). Go runtime handles this.
  - OS-level threads may be scheduled to run on arbitrary CPU core by OS.
  - All goroutines share the same address space
  - Synchronization → no IPC required. Can use shared memory (mutex, locks, barriers) for synchronization.
- **Spawn goroutine? → `go funcCall(args...)`**

Goroutines Example

```
1  func child() {
2      for {
3          var msSleep int = rand.IntN(2000) + 500
4          time.Sleep(time.Duration(msSleep) * time.Millisecond)
5          fmt.Println("Child: are we there yet?")
6      }
7  }
8
9  func parent() {
10     for {
11         var msSleep int = rand.IntN(3000) + 1000
12         time.Sleep(time.Duration(msSleep) * time.Millisecond)
13         fmt.Println("Parent: 10 more minutes")
14     }
15 }
16
17 func main() {
18     go child()
19     go parent()
20     for { time.Sleep(time.Second) }
21 }
```



Goroutines

- Join gorountine? pthread_join?
- Nope! Will need to use other tools
- Sync.Mutex


```
1 var childLock sync.Mutex
2
3 func child() {
4     fmt.Println("Getting lunch...")
5     time.Sleep(time.Second)
6     fmt.Println("Getting homework...")
7     time.Sleep(time.Second)
8     fmt.Println("Getting socks...")
9     time.Sleep(time.Second)
10    fmt.Println("Getting jacket...")
11    time.Sleep(time.Second)
12    childLock.Unlock()
13 }
14
15 func main() {
16    childLock.Lock()
17    go child()
18    // Wait for child
19    childLock.Lock()
20    fmt.Println("Walking to school!")
21 }
```

Goroutines

- **Sync.WaitGroup → wait for a group of goroutines**
- `wg.Add(n)` → add n goroutines to wait for (usually 1 or all)
- `wg.Done()` → call by goroutine when done (subtracts 1)
- `wg.Wait()` → block until no goroutines running (wait for 0)

```go
func main() {
    var wg sync.WaitGroup
    const numChildren = 5

    childDoHomework := func(num int) {
        for {
            fmt.Printf("Child %d working...\n", num)
            wait := time.Duration(rand.IntN(2000)+500)
            time.Sleep(time.Millisecond * wait)
            if rand.Float64() <= 0.25 {
                break
            }
        }
        fmt.Printf("Child %d done\n", num)
        wg.Done()
    }

    wg.Add(numChildren)
    for i := range numChildren {
        go childDoHomework(i)
    }
    wg.Wait()
    fmt.Println("Everyone is done!")
}

```

Goroutines

- **Handling synchronization of global data via sync primitives**

```go
func main() {
    var total int = 0
    var wg sync.WaitGroup
    wg.Add(1000000)
    for range 1000000 {
        go func() {
            defer wg.Done()
            total += 1
        }()
    }
    wg.Wait()
    fmt.Println(total)
}
```

```
$ for i in {1..100}; do go run _; done
922539
927714
929655
925084
925332
923226
921020
924350
919063
918562
917953
911826
915570
923962
915425
913478
```

Goroutines

- **Handling synchronization of global data via sync primitives**

```go
func main() {
    var total int = 0
    var lock sync.Mutex
    var wg sync.WaitGroup
    wg.Add(1000000)
    for range 1000000 {
        go func() {
            defer wg.Done()
            lock.Lock()
            total += 1
            lock.Unlock()
        }()
    }
    wg.Wait()
    fmt.Println(total)
}
```

```
$ for i in {1..100}; do go run .; done
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
```

Channels

- **What are channels?**
  - bidirectional communication mechanism between goroutines
- **`make(chan TYPE)`**
- **Can send/receive things on channels**
  - Send: `myChannel <- value`
  - Receive: `var := <-myChannel`
- **Channels include the type of data transferred → chan int, chan MyStruct**

Channels

```
5 func main() {
6     postOffice := make(chan string)
7     go penPal(postOffice)
8     message := "Hi there pen pal!"
9     fmt.Println("main sending message")
10    postOffice <- message
11    reply := <-postOffice
12    fmt.Println("main received a reply!", reply)
13 }
```

Waits for message

```
15 func penPal(messageChan chan string) {
16     message := <-messageChan
17     fmt.Println("Pen pal received a message!")
18     newMessage := "Cool message! " + message
19     messageChan <- newMessage
20 }
```

```
$ go run .
main sending message
Pen pal received a message!
main received a reply! Cool message! Hi there pen pal!
$
```

Channels

- **What if both try to send messages at same time?**

```
1 func main() {
2     postOffice := make(chan string)
```

```
$ go run .
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:6 +0x70

goroutine 5 [chan send]:
main.penPal(...)
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:10
created by main.main in goroutine 1
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:5 +0x5f
exit status 2
$
```

Channels

- **What if both wait to receive messages at same time?**

```
1 func main() {
```

```
$ go run .
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:8 +0x7b

goroutine 18 [chan receive]:
main.penPal(0x0?)
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:13 +0x1e
created by main.main in goroutine 1
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:7 +0x66
exit status 2
$
```

Channels

- Channels are preferred over synchronization primitives
- “Don’t communicate by sharing memory, share memory by communicating”
- Synchronization primitives are shared data and sync mechanism
- Channels have no shared data, send/receive state with other goroutines
- Channels are similar to distributed computing, for example communicating between multiple hosts over HTTP/REST

Channels

- **Unbuffered vs buffered channels**
- **Unbuffered = channel has no capacity**
  - Think of unbuffered as writer literally handing message to reader
  - Aka **synchronous** communication
  - `make(chan int)`
  - `make(chan int, 0)`
- **Buffered = channel has capacity → `make(chan int, 100)`**
  - Aka **asynchronous** communication
  - Think of buffered channel as a queue

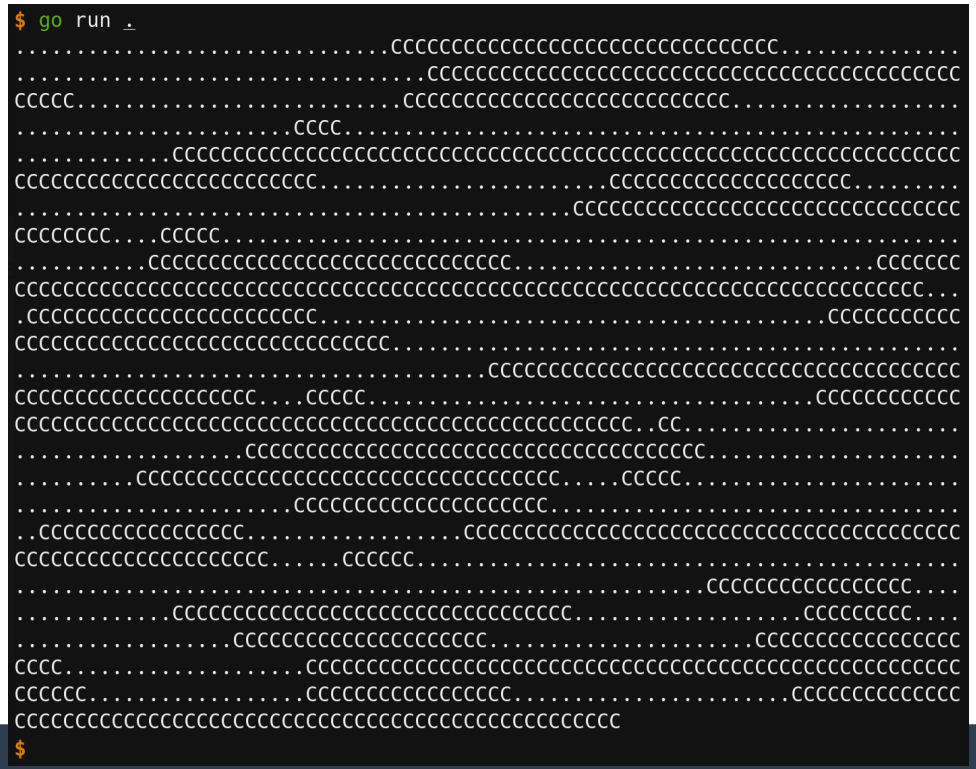
Channels

- **When do channels block?**

	Read	Write
Unbuffered	Blocks until value available	Blocks until value is read
Buffered	Blocks if channel empty	Blocks if channel is full

Channels

- **Buffered channel example**




```go
var wg sync.WaitGroup

func main() {
    ch := make(chan bool, 100)
    wg.Add(2)
    go producer(ch)
    go consumer(ch)
    wg.Wait()
    fmt.Print("\n")
}

func producer(ch chan bool) {
    defer wg.Done()
    for range 1000 {
        ch <- false
        fmt.Print(".")
    }
    ch <- true
}

func consumer(ch chan bool) {
    defer wg.Done()
    for {
        val := <-ch
        if val == true {
            break
        }
        fmt.Print("C")
    }
}
```

Channels

- **Same example but UNBUFFERED**

```
$ go run .
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
.CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..C
C..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC..CC.
$
```

```go
var wg sync.WaitGroup

func main() {
    ch := make(chan bool, 0)
    wg.Add(2)
    go producer(ch)
    go consumer(ch)
    wg.Wait()
    fmt.Print("\n")
}

func producer(ch chan bool) {
    defer wg.Done()
    for range 1000 {
        ch <- false
        fmt.Print(".")
    }
    ch <- true
}

func consumer(ch chan bool) {
    defer wg.Done()
    for {
        val := <-ch
        if val == true {
            break
        }
        fmt.Print("C")
    }
}
```

Channels

- Really should treat channels as unidirectional most of the time.
- If need bidirectional then just use two channels. Especially for buffered channels.

```
1 func main() {
2     fruitSalad := make(chan bool, 10)
3     pancake := make(chan bool, 10)
4     wg.Add(2)
5     go chef1(fruitSalad, pancake)
6     go chef2(fruitSalad, pancake)
7     wg.Wait()
8 }
```

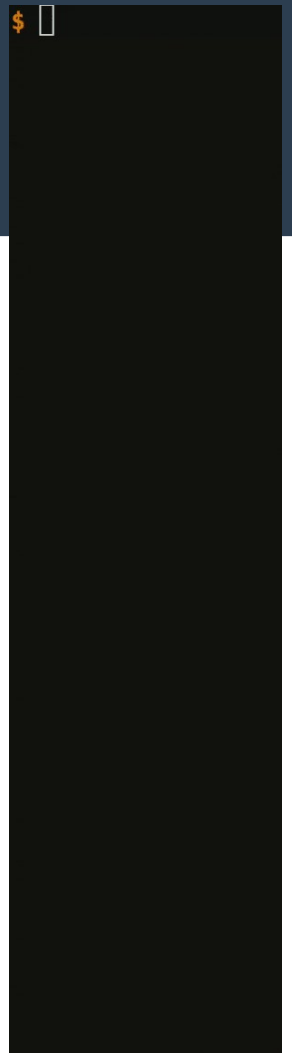
```
1 func chef1(fruitSalad chan bool, pancake chan bool) {
2     defer wg.Done()
3     for range 1000 {
4         <-fruitSalad    // chef1 likes fruit salad
5         pancake <- true // chef1 makes pancaked
6     }
7 }
```

```
1 func chef2(fruitSalad chan bool, pancake chan bool) {
2     defer wg.Done()
3     for range 1000 {
4         fruitSalad <- true // chef2 makes fruit salad
5         <-pancake          // chef2 likes pancakes
6     }
7 }
```

Channels

- Select control flow
- wait for multiple channels
- Useful for timeout and stopping conditions

```
1 func main() {
2     chNumber := time.Tick(time.Second)
3     chFizz := time.Tick(time.Second * 3)
4     chBuzz := time.Tick(time.Second * 5)
5     chFizzBuzz := time.Tick(time.Second * 3 * 5)
6     chTimeout := time.After(time.Second * 20)
7     num := 1
8
9     for {
10        select {
11        case <-chNumber:
12            fmt.Println(num)
13            num += 1
14        case <-chFizz:
15            fmt.Println("Fizz")
16        case <-chBuzz:
17            fmt.Println("Buzz")
18        case <-chFizzBuzz:
19            fmt.Println("Fizz Buzz")
20        case <-chTimeout:
21            return
22        }
23    }
24 }
```



Channels

- **Channel direction “send-only” and “read-only”**
- **Type of channel changes**
- **Compile time error if read a send-only channel, and vice-versa**

```go
var wg sync.WaitGroup

func main() {
    wg.Add(2)
    var chNum chan int = make(chan int, 100)
    go producer(chNum)
    go consumer(chNum)
    wg.Wait()
}

func producer(ch chan<- int) {
    defer wg.Done()
    for n := range 1000 {
        ch <- n
    }
    close(ch)
}

func consumer(ch <-chan int) {
    defer wg.Done()
    for {
        num, ok := <-ch
        if !ok {
            return
        }
        fmt.Println(num)
    }
}
```

Channels

- **Multi-reader multi-writer example**

```go
var wg sync.WaitGroup

func main() {
    wg.Add(10)
    ch := make(chan string, 100)
    go writer(ch, "A")
    go writer(ch, "B")
    go writer(ch, "C")
    go writer(ch, "D")
    go writer(ch, "E")
    go reader(ch)
    go reader(ch)
    go reader(ch)
    go reader(ch)
    go reader(ch)
    wg.Wait()
    fmt.Print("\n")
}
```

```go
func writer(ch chan string, letter string) {
    defer wg.Done()
    for range 100 {
        ch <- letter
    }
    ch <- "DONE"
}

func reader(ch chan string) {
    defer wg.Done()
    for {
        letter := <-ch
        if letter == "DONE" {
            return
        }
        fmt.Print(letter)
    }
}
```

```
$ go run .
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAEBCDDDDDDDBCAEBABBBBDCA
BBBBBBBCEDCCCCCDCCCCCCEDBCDCDD
DBCEBBBBBBBBBBEBCEBBBBBCEDCCCC
CBCEDCCCCCCCCBCDEBCDEBBBBBBBBB
BCDEDBCEDBCEBBBDBBBBDBBBBBDDCE
BBBBBBCEDBECDBCCCCCCCCCCEDEBCD
EBCDEEEEBEEEEEEBCDEBCDEBCDEBBB
BBBBBBBBDCEBDCBEEEEEEEEEEDCBDE
CCDEBDCEBCDEBCDEBCDEBCBEEEEEEE
EEBCDEBCDEDDDDDDDDDDBCEEEEDEEE
EEEEEBCDEBCDBBEEDDDDDDDDDDECED
CEDCCCCCDCCCCCECDECDECCCCCCCCC
CCCDECDEDEEEEEEECEEEEDEDEDEEED
DDDDDDDDDDDDDDDDDDDD
```

Channels

- **Closing channels → close(). Closed = no more values sent.**
  - Reading a closed channel → OK while channel has values, after gives zero value
  - Writing a closed channel → panic
  - Closing a closed channel → panic
  - Reader detecting a closed channel → `val, ok := <-myChannel`
- **Not necessary to close channel, but good way to indicate end-of-transmission EOT**

Extra: Runtime Type Comparison

```go
func main() {
    var bird interface{} = ""
    if bird == "" {
        fmt.Println("true branch") // taken
    } else {
        fmt.Println("false branch")
    }
}
```

```go
func main() {
    var bird interface{} = ""
    switch birdVal := bird.(type) {
    case string:
        if birdVal == "" {
            fmt.Println("true branch") // taken
        } else {
            // unexpected value
        }
    default:
        panic("bird really should be a string.")
    }
}
```

- A ***runtime*** type check is performed first, if types not equal then always false, otherwise do regular comparison

Practice!

- https://github.com/calvincramer/golang-training-3

Training 3 - Concurrency

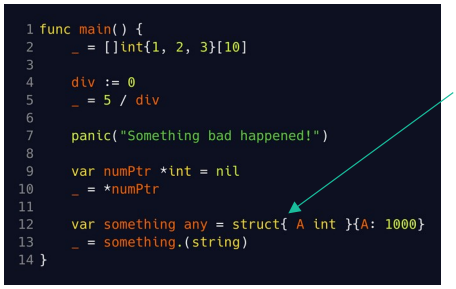
Calvin Lei-Cramer
May 2025

What are we covering today?

Panic and recover, goroutines and concurrency, channels

Panic! Recover.

- **What are panics actually? Are they exceptions?**
- **What triggers panics?**
  - Array/slice index out of bounds
  - Divide by zero
  - Calling `panic()`
  - Nil pointer deference
  - Type assertion (without using 'ok' second return value) failing



```
1  func main() {
2      _ = []int{1, 2, 3}[10]
3
4      div := 0
5      _ = 5 / div
6
7      panic("Something bad happened!")
8
9      var numPtr *int = nil
10     _ = *numPtr
11
12     var something any = struct{ A int }{A: 1000}
13     _ = something.(string)
14 }
```

3

On a panic, all DEFERRED functions in current function executed. Very similar to exception.

Then the caller of the current function has its DEFERRED functions called.

If panic is not handled in deferred functions then it will cause program to terminate. Same as exceptions.

Panics and exceptions are SIMILAR in how they are INVOKED, but Golang and other language are different in how you are allowed to HANDLE exceptions, and where normal execution can RECOVER from.

Anonymous struct

Panic! Recover.

- **How to catch panics?**
- **`recover()` global**
  - Returns value passed to panic() if panicking
  - Otherwise returns nil
- **Named return values, defer, recover()**

```
$ go run .
25 / 2 =  12
43 / 0 =  9223372036854775807
-96 / 0 =  -9223372036854775808
$
```

```
1 func SafeDivide(num, div int) (result int) {
2     defer func() {
3         if recover() != nil {
4             // Set to near infinity
5             if num < 0 {
6                 result = math.MinInt
7             } else {
8                 result = math.MaxInt
9             }
10        }
11    }()
12    result = num / div // may cause panic
13    return
14 }
```

4

Deferred function may change named return values in order to return an appropriate value that indicates an error, or any adjustment that's necessary because of the panic.

Keep in mind, any deferred function call may be ran in the context of a panic happening. SHARED USE OF NORMAL EXECUTION AND PANIC EXECUTION.

Panic! Recover.

- **Pattern to recover normal execution flow like try/catch**

Python

```python
def main():
    print("Before")
    try:
        badFunction()
    except Exception as e:
        pass
    print("After")
```

```
$ go run .
Before
Log panic: something bad happened in badFunc()
After
```

Golang

```go
func handlePanic(action func()) {
    defer func() {
        if val := recover(); val != nil {
            fmt.Println("Log panic:", val)
        }
    }()
    action()
}

func main() {
    fmt.Println("Before")
    handlePanic(badFunc)
    fmt.Println("After")
}
```

In Golang the try/catch forces the function that recovers from the panic to return.

Panic! Recover.

- **What happens if panic, then panic again in defer function?**

```
3 func main() {
4     defer func() {
5         panic("panic #2")
6     }()
7     panic("panic #1")
8 }
```

```
$ go run .
panic: panic #1
        panic: panic #2

goroutine 1 [running]:
main.main.func1()
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:5 +0x25
panic({0x46f5a0?, 0x499a78?})
        /usr/local/go/src/runtime/panic.go:792 +0x132
main.main()
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:7 +0x3e
exit status 2
$ _
```

6

Both panics printed

Execution trace shows line numbers of both panics

Can panic while panicking.

Panic! Recover.

```
3 func doStuff() {
4     defer func() {
5         if val := recover(); val != nil {
6             fmt.Println("Recovered panic:", val)
7         }
8     }()
9     defer func() {
10        panic("panic #2")
11    }()
12    panic("panic #1")
13 }
14
15 func main() {
16    fmt.Println("Before")
17    doStuff()
18    fmt.Println("After")
19 }
```

- **Panic, panic, recover?**

```
$ go run .
Before
Recovered panic: panic #2
After
$
```

Panic overrides any current panic value.

Keep panic recover simple.

Golang Idioms

- Idiomatic Go, mantras, patterns, group think
- “Programming in a language vs programming into a language” – Code Complete by Steve McConnell
- Error handling → errors are values, don’t code by exception
- Defer
- Goroutines and concurrency models

```
3 func fetch(url string) (body string, err error) {
4     resp, err := http.Get(url)
5     if err != nil {
6         return
7     }
8     defer resp.Body.Close()
9     bodyBytes, err := io.ReadAll(resp.Body)
10    if err != nil {
11        return
12    }
13    body = string(bodyBytes)
14    return
15 }
16
17 func main() {
18    resp, err := fetch("https://www.google.com")
19    if err != nil {
20        fmt.Println("error fetching URL -", err)
21        return
22    }
23    fmt.Println(resp)
24 }
```

8

Pythonic python

Programming IN a language means thinking in terms of what the language offers. This may limit how you think and solve problems, for example working in C.

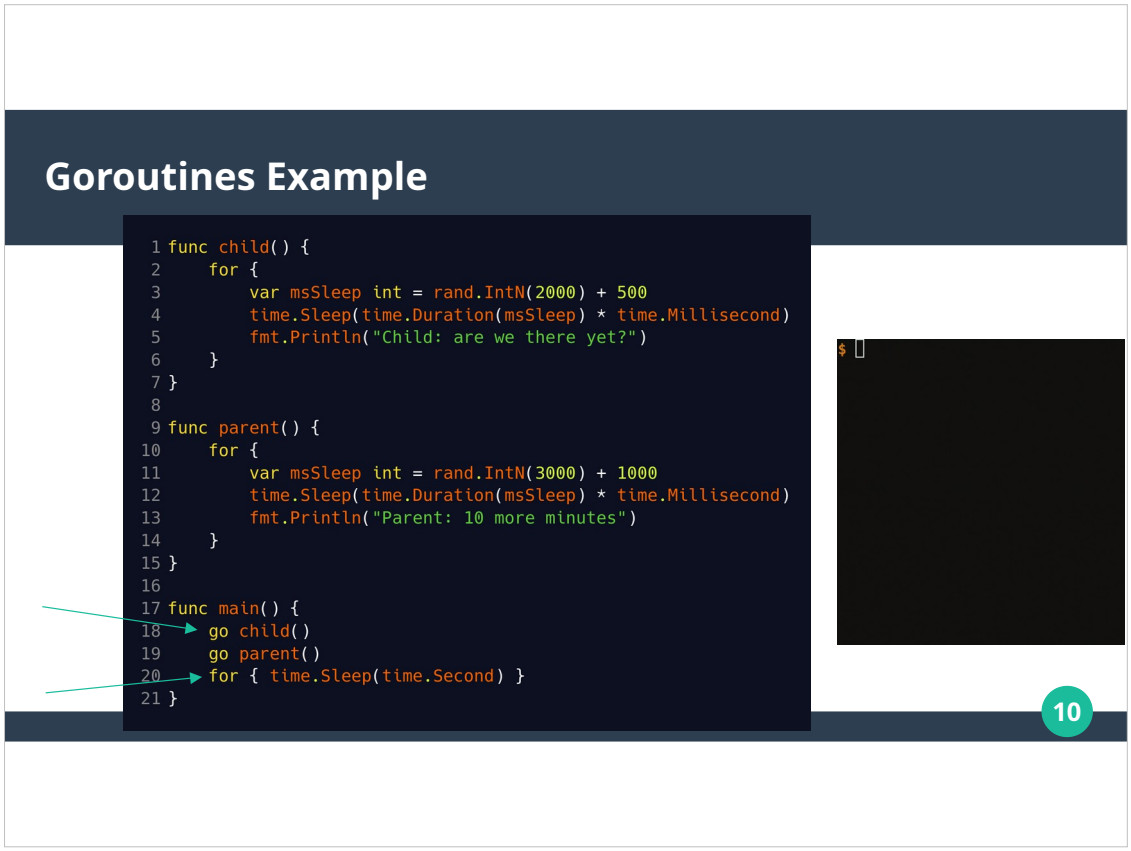
Programming INTO a language means thinking what you want to accomplish first, then accomplishing that with the tools in the language.

Goroutines

- **What are goroutines?**
  - Lightweight application-level threads managed by Go runtime
  - Multiple goroutines may run on a single OS-level thread (concurrency) or multiple OS-level threads (parallelism). Go runtime handles this.
  - OS-level threads may be scheduled to run on arbitrary CPU core by OS.
  - All goroutines share the same address space
  - Synchronization → no IPC required. Can use shared memory (mutex, locks, barriers) for synchronization.
- **Spawn goroutine? → `go funcCall(args...)`**

9

Goroutines not guaranteed to run in parallel on multiple CPUs.



The main() function is running as the “MAIN” goroutine.

Once the main thread exits then the application terminates. That is why we need to wait forever in the main function to allow the other goroutines to run. The program is terminated with a CTRL+C

Goroutines

- Join gorountine? pthread_join?
- Nope! Will need to use other tools
- Sync.Mutex


```
1 var childLock sync.Mutex
2
3 func child() {
4     fmt.Println("Getting lunch...")
5     time.Sleep(time.Second)
6     fmt.Println("Getting homework...")
7     time.Sleep(time.Second)
8     fmt.Println("Getting socks...")
9     time.Sleep(time.Second)
10    fmt.Println("Getting jacket...")
11    time.Sleep(time.Second)
12    childLock.Unlock()
13 }
14
15 func main() {
16    childLock.Lock()
17    go child()
18    // Wait for child
19    childLock.Lock()
20    fmt.Println("Walking to school!")
21 }
```

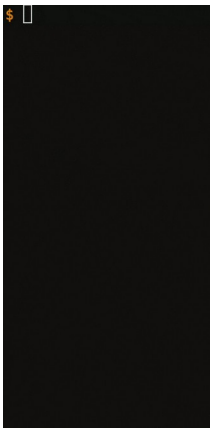
11

Spawning a goroutine does not give an ID of the new goroutine. This is intentional design.

Goroutines

- **Sync.WaitGroup → wait for a group of goroutines**
- `wg.Add(n)` → add n goroutines to wait for (usually 1 or all)
- `wg.Done()` → call by goroutine when done (subtracts 1)
- `wg.Wait()` → block until no goroutines running (wait for 0)

```go
func main() {
    var wg sync.WaitGroup
    const numChildren = 5

    childDoHomework := func(num int) {
        for {
            fmt.Printf("Child %d working...\n", num)
            wait := time.Duration(rand.IntN(2000)+500)
            time.Sleep(time.Millisecond * wait)
            if rand.Float64() <= 0.25 {
                break
            }
        }
        fmt.Printf("Child %d done\n", num)
        wg.Done()
    }

    wg.Add(numChildren)
    for i := range numChildren {
        go childDoHomework(i)
    }
    wg.Wait()
    fmt.Println("Everyone is done!")
}

```



Goroutines

- **Handling synchronization of global data via sync primitives**

```go
func main() {
    var total int = 0
    var wg sync.WaitGroup
    wg.Add(1000000)
    for range 1000000 {
        go func() {
            defer wg.Done()
            total += 1
        }()
    }
    wg.Wait()
    fmt.Println(total)
}
```

```
$ for i in {1..100}; do go run .; done
922539
927714
929655
925084
925332
923226
921020
924350
919063
918562
917953
911826
915570
923962
915425
913478
```

13

Golang will NOT protect you from data races.

Golang will NOT warn you about possible data races.

Goroutines

- **Handling synchronization of global data via sync primitives**

```go
func main() {
    var total int = 0
    var lock sync.Mutex
    var wg sync.WaitGroup
    wg.Add(1000000)
    for range 1000000 {
        go func() {
            defer wg.Done()
            lock.Lock()
            total += 1
            lock.Unlock()
        }()
    }
    wg.Wait()
    fmt.Println(total)
}
```

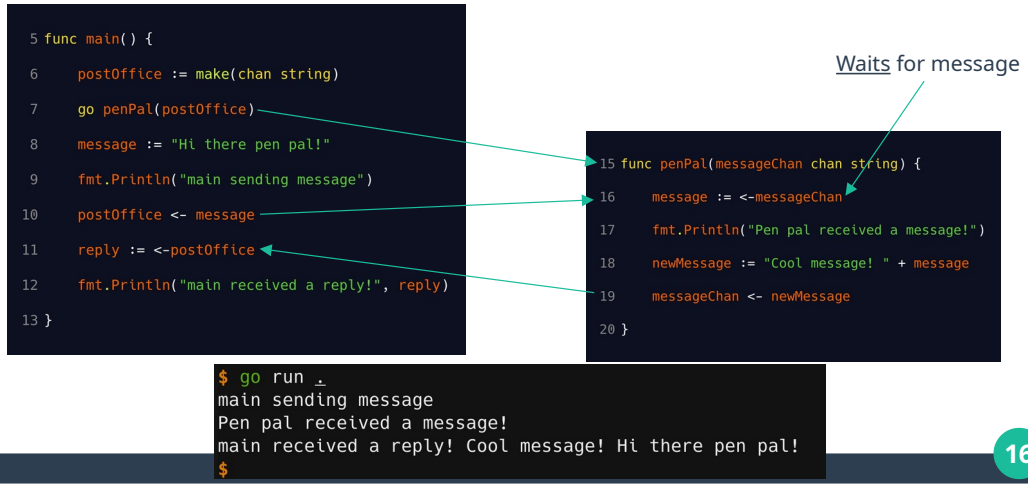
```
$ for i in {1..100}; do go run _; done
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
```


Channels are the preferred way to share data, which we will see next.

Channels

- **What are channels?**
  - bidirectional communication mechanism between goroutines
- **`make(chan TYPE)`**
- **Can send/receive things on channels**
  - Send: `myChannel <- value`
  - Receive: `var := <-myChannel`
- **Channels include the type of data transferred → chan int, chan MyStruct**

Channels




```
$ go run .
main sending message
Pen pal received a message!
main received a reply! Cool message! Hi there pen pal!
$
```


Point out:
- making channel
- reading value from channel
- reading value from channel blocks until message is available
- writing to channel

Channels

- **What if both try to send messages at same time?**

```
1 func main() {
2     postOffice := make(chan string)
```

```
$ go run .
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:6 +0x70

goroutine 5 [chan send]:
main.penPal(...)
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:10
created by main.main in goroutine 1
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:5 +0x5f
exit status 2
$
```

Channels

- **What if both wait to receive messages at same time?**

```
1 func main() {
```

```
$ go run .
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:8 +0x7b

goroutine 18 [chan receive]:
main.penPal(0x0?)
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:13 +0x1e
created by main.main in goroutine 1
        /home/ccramer-loc/repos/GOLANG_TRAINING/golang-training-private/main.go:7 +0x66
exit status 2
$
```

Channels

- Channels are preferred over synchronization primitives
- “Don’t communicate by sharing memory, share memory by communicating”
- Synchronization primitives are shared data and sync mechanism
- Channels have no shared data, send/receive state with other goroutines
- Channels are similar to distributed computing, for example communicating between multiple hosts over HTTP/REST

Distributed computed can get ugly really fast.

Consider higher level abstractions than channels if building complex systems.

Channels

- **Unbuffered vs buffered channels**
- **Unbuffered = channel has no capacity**
  - Think of unbuffered as writer literally handing message to reader
  - Aka **synchronous** communication
  - `make(chan int)`
  - `make(chan int, 0)`
- **Buffered = channel has capacity → `make(chan int, 100)`**
  - Aka **asynchronous** communication
  - Think of buffered channel as a queue

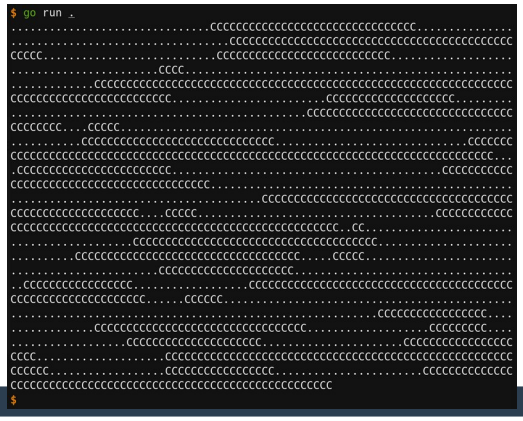
Channels

- **When do channels block?**

	Read	Write
Unbuffered	Blocks until value available	Blocks until value is read
Buffered	Blocks if channel empty	Blocks if channel is full

Channels

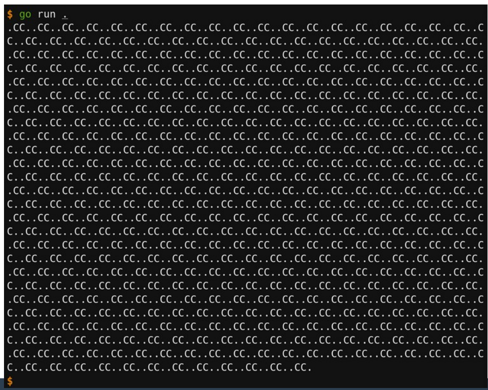
- **Buffered channel example**




```go
var wg sync.WaitGroup

func main() {
    ch := make(chan bool, 100)
    wg.Add(2)
    go producer(ch)
    go consumer(ch)
    wg.Wait()
    fmt.Print("\n")
}

func producer(ch chan bool) {
    defer wg.Done()
    for range 1000 {
        ch <- false
        fmt.Print(".")
    }
    ch <- true
}

func consumer(ch chan bool) {
    defer wg.Done()
    for {
        val := <-ch
        if val == true {
            break
        }
        fmt.Print("C")
    }
}
```

Channels

- **Same example but UNBUFFERED**




```go
var wg sync.WaitGroup

func main() {
    ch := make(chan bool, 0)
    wg.Add(2)
    go producer(ch)
    go consumer(ch)
    wg.Wait()
    fmt.Print("\n")
}

func producer(ch chan bool) {
    defer wg.Done()
    for range 1000 {
        ch <- false
        fmt.Print(".")
    }
    ch <- true
}

func consumer(ch chan bool) {
    defer wg.Done()
    for {
        val := <-ch
        if val == true {
            break
        }
        fmt.Print("C")
    }
}
```

Channels

- Really should treat channels as unidirectional most of the time.
- If need bidirectional then just use two channels. Especially for buffered channels.

```go
func main() {
    fruitSalad := make(chan bool, 10)
    pancake := make(chan bool, 10)
    wg.Add(2)
    go chef1(fruitSalad, pancake)
    go chef2(fruitSalad, pancake)
    wg.Wait()
}
```

```go
func chef1(fruitSalad chan bool, pancake chan bool) {
    defer wg.Done()
    for range 1000 {
        <-fruitSalad    // chef1 likes fruit salad
        pancake <- true // chef1 makes pancaked
    }
}
```

```go
func chef2(fruitSalad chan bool, pancake chan bool) {
    defer wg.Done()
    for range 1000 {
        fruitSalad <- true // chef2 makes fruit salad
        <-pancake          // chef2 likes pancakes
    }
}
```

Channels

- Select control flow
- wait for multiple channels
- Useful for timeout and stopping conditions

```
1 func main() {
2     chNumber := time.Tick(time.Second)
3     chFizz := time.Tick(time.Second * 3)
4     chBuzz := time.Tick(time.Second * 5)
5     chFizzBuzz := time.Tick(time.Second * 3 * 5)
6     chTimeout := time.After(time.Second * 20)
7     num := 1
8
9     for {
10        select {
11        case <-chNumber:
12            fmt.Println(num)
13            num += 1
14        case <-chFizz:
15            fmt.Println("Fizz")
16        case <-chBuzz:
17            fmt.Println("Buzz")
18        case <-chFizzBuzz:
19            fmt.Println("Fizz Buzz")
20        case <-chTimeout:
21            return
22        }
23    }
24 }
```

```
$
```

25

time.Tick() returns a channel which periodically sends a message of the current time

time.After() returns a channel that sends a single message after a specified duration.

Channels

- Channel direction “send-only” and “read-only”
- Type of channel changes
- Compile time error if read a send-only channel, and vice-versa

```go
var wg sync.WaitGroup

func main() {
    wg.Add(2)
    var chNum chan int = make(chan int, 100)
    go producer(chNum)
    go consumer(chNum)
    wg.Wait()
}

func producer(ch chan<- int) {
    defer wg.Done()
    for n := range 1000 {
        ch <- n
    }
    close(ch)
}

func consumer(ch <-chan int) {
    defer wg.Done()
    for {
        num, ok := <-ch
        if !ok {
            return
        }
        fmt.Println(num)
    }
}
```

Channels

- **Multi-reader multi-writer example**

```go
var wg sync.WaitGroup

func main() {
    wg.Add(10)
    ch := make(chan string, 100)
    go writer(ch, "A")
    go writer(ch, "B")
    go writer(ch, "C")
    go writer(ch, "D")
    go writer(ch, "E")
    go reader(ch)
    go reader(ch)
    go reader(ch)
    go reader(ch)
    go reader(ch)
    wg.Wait()
    fmt.Print("\n")
}
```

```go
func writer(ch chan string, letter string) {
    defer wg.Done()
    for range 100 {
        ch <- letter
    }
    ch <- "DONE"
}

func reader(ch chan string) {
    defer wg.Done()
    for {
        letter := <-ch
        if letter == "DONE" {
            return
        }
        fmt.Print(letter)
    }
}
```

```
$ go run .
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAEBCDDDDDDDBCAEBABBBBDCA
BBBBBBBCEDCCCCDCCCCCCEDBCDCDD
DBCEBBBBBBBBBBEBCEBBBBBCEDCCCC
CBCEDCCCCCCCBCDEBCDEBBBBBBBBB
BCDEDBCEDBCEBBBDBBBBDBBBBBDDCE
BBBBBBCEDBECDBCCCCCCCCCCEDEBCD
EBCDEEEEBEEEEEEBCDEBCDEBCDEBBB
BBBBBBBBDCEBDCBEEEEEEEEEEDCBDE
CCDEBDCEBCDEBCDEBCDEBCBEEEEEEE
EEBCDEBCDEDDDDDDDDDDBCEEEEDEEE
EEEEEBCDEBCDBBEEDDDDDDDDDECED
CEDCCCCCDCCCCCECDECDECCCCCCCCC
CCCDECDEDEEEEEEECEEEEDEDEDEEED
DDDDDDDDDDDDDDDDDDD
```

Channels

- **Closing channels → close(). Closed = no more values sent.**
  - Reading a closed channel → OK while channel has values, after gives zero value
  - Writing a closed channel → panic
  - Closing a closed channel → panic
  - Reader detecting a closed channel → `val, ok := <-myChannel`
- **Not necessary to close channel, but good way to indicate end-of-transmission EOT**

28

Golang will garbage collect non-closed channels

ONLY THE SENDER SHOULD CLOSE CHANNEL

Since closing a closed channel gives panics, be careful with multiple writers.

Extra: Runtime Type Comparison

```
1 func main() {
2     var bird interface{} = ""
3     if bird == "" {
4         fmt.Println("true branch") // taken
5     } else {
6         fmt.Println("false branch")
7     }
8 }
```

```
1 func main() {
2     var bird interface{} = ""
3     switch birdVal := bird.(type) {
4     case string:
5         if birdVal == "" {
6             fmt.Println("true branch") // taken
7         } else {
8             // unexpected value
9         }
10    default:
11        panic("bird really should be a string.")
12    }
13 }
```

- A ***runtime*** type check is performed first, if types not equal then always false, otherwise do regular comparison

29

Very interesting to see this behavior of Golang, since it checks types at compile time mostly.

Practice!

- https://github.com/calvincramer/golang-training-3