Channels & Coordination

DD1396
Ric Glassey
glassey@kth.se

Concepts

Using channels for communication

Blocking and non-blocking behaviour

Coordination between goroutines

Channels

Channels

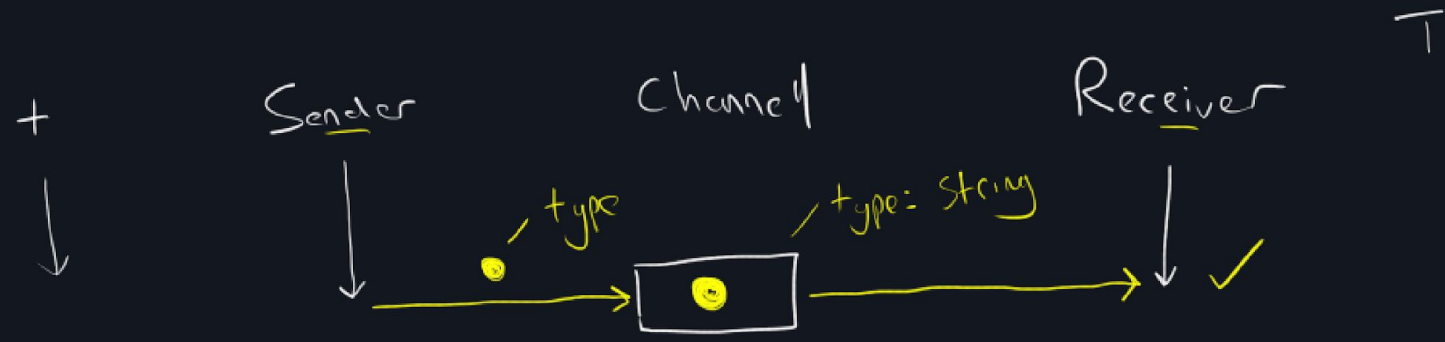
Channels are **shared objects** that let goroutines communicate

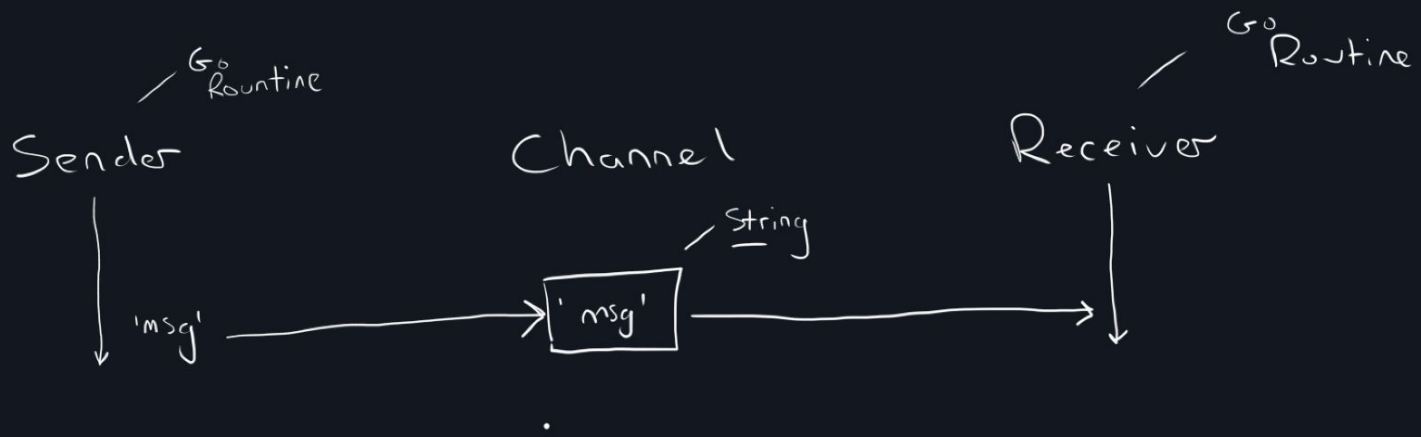
- **Write** values to channel
- **Read** values from a channel
- Contains **typed values**
- **Buffered** or **unbuffered**

They also help us **coordinate** goroutine behaviour

```go
package main

import "fmt"

func main() {

    // Create a new channel
    messages := make(chan string)

    // Send a value into channel
    go func() { messages <- "ping" }()

    // Read value from channel
    msg := <- messages

    fmt.Println(msg)
 }
```

Draw: Channel


T
Sender
Channel
Receiver
type
type: string


Go Routine
Sender
Channel
Receiver
Go Routine
String
'msg'
'msg'

Video

Pinger, Ponger & Printer

```
func main() {
    // create a channel for communication
    c := make(chan string)

    // create three concurrent threads
    go pinger(c)
    go ponger(c)
    go printer(c)

    // runs until user input
    var input string
    fmt.Scanln(&input)
}
```

Pinger & Ponger functions

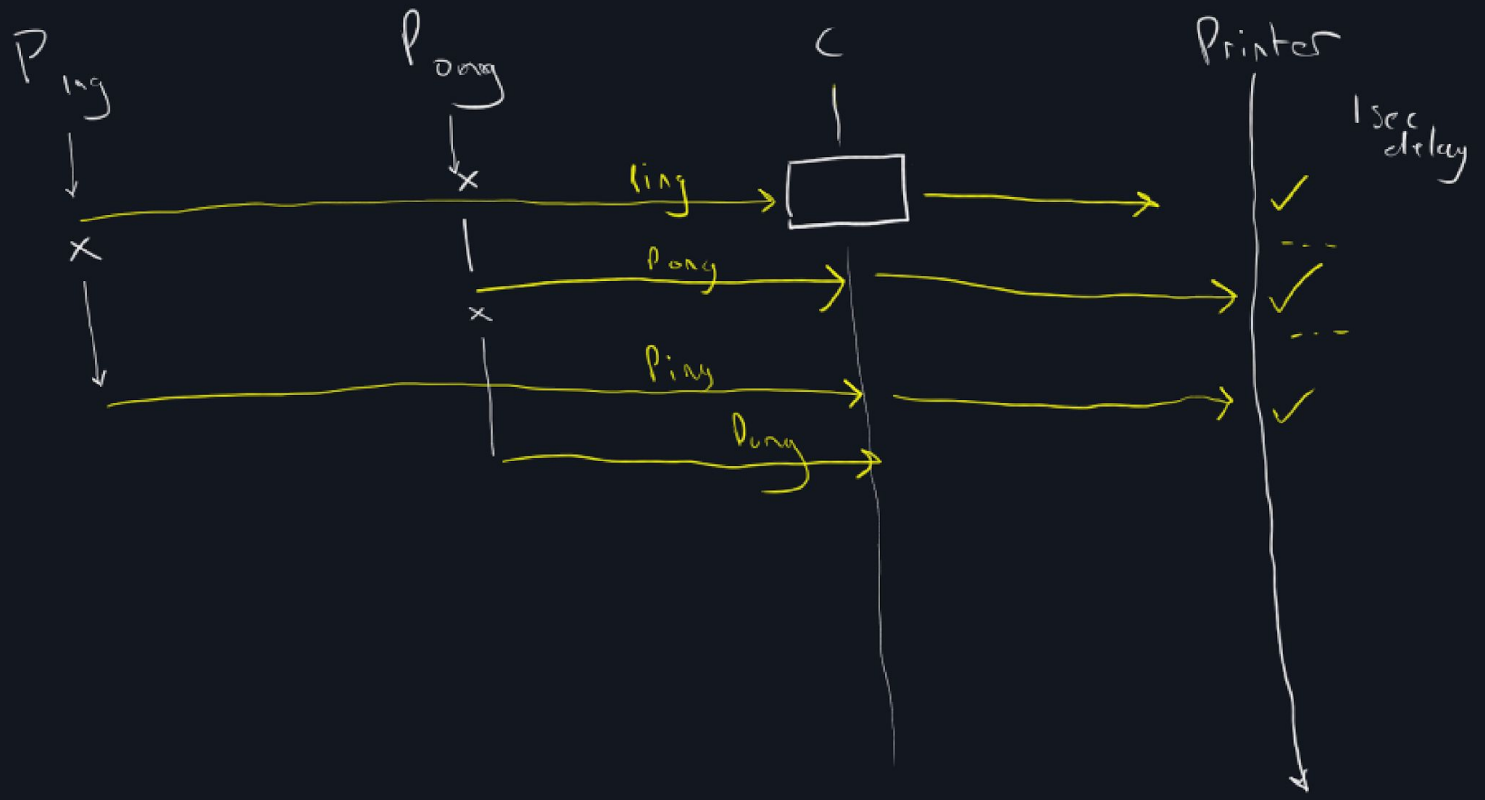
```
func pinger(c chan string) {
    // infinite for loop with counter
    for i := 0; ; i++ {
        // send message into channel
        // synchronises channel - wait for read operation
        // we say it is in a "blocking" state
        c <- "ping"
    }
}

func ponger(c chan string) {
    // same as above
    for i := 0; ; i++ {
        c <- "pong"
    }
}
```

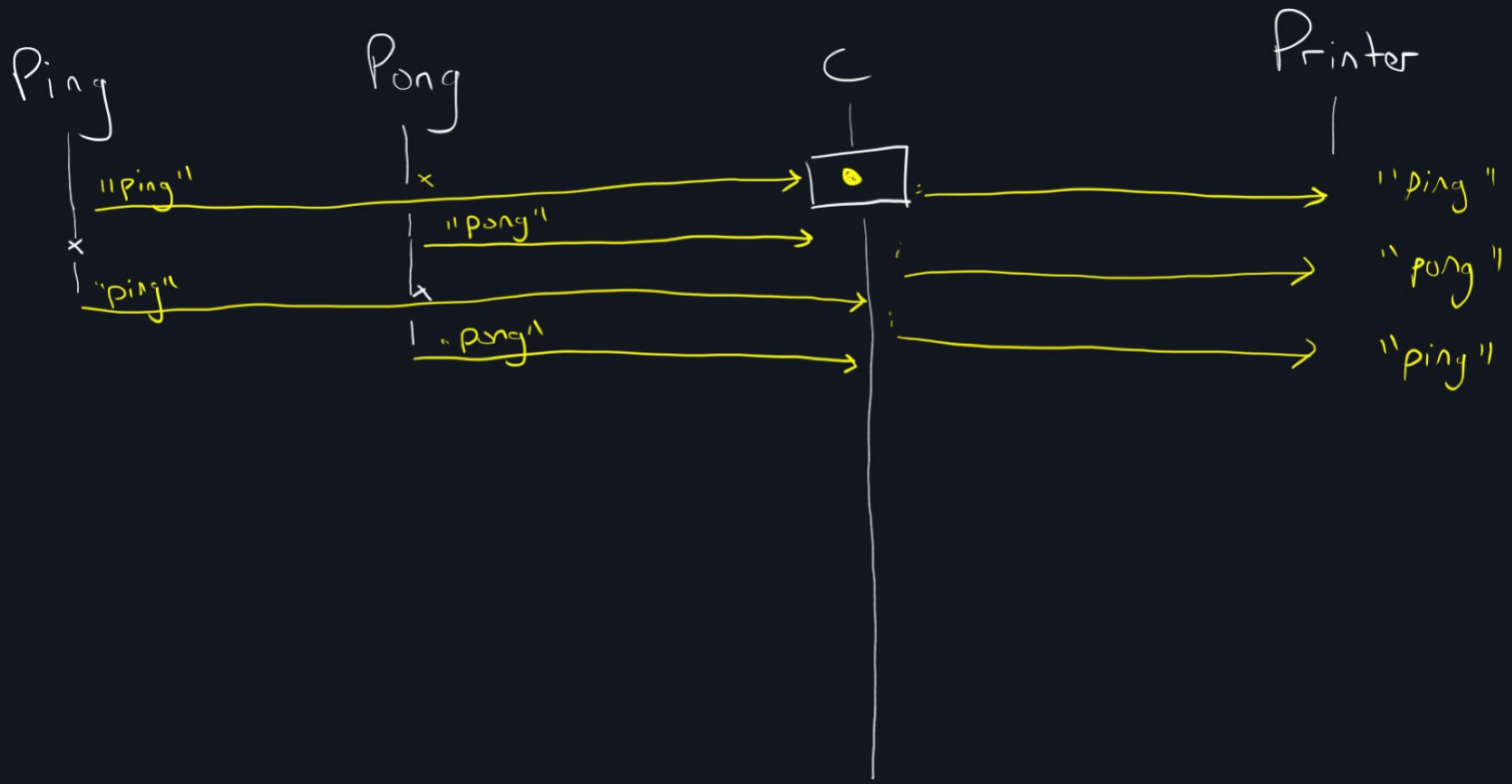
Printer function

```
func printer(c chan string) {
    for {
        // receive messsages in channel
        msg := <- c
        fmt.Println("printer received:", msg)
        time.Sleep(time.Second * 1)
    }
}
```

Demo: Pinger, Ponger & Printer

Draw: Pinger, Ponger & Printer


P ing
P ong
C
Printer
1 sec delay
ping
pong
ping
pong



Video

Channel Buffering

Unbuffered channel

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

No buffer

Either empty or full

Read/write operations are **blocking**

- Read blocks until something is sent into channel
- Write blocks until something has been read from chanel

Buffered channel

```
messages := make(chan string, 10)
```

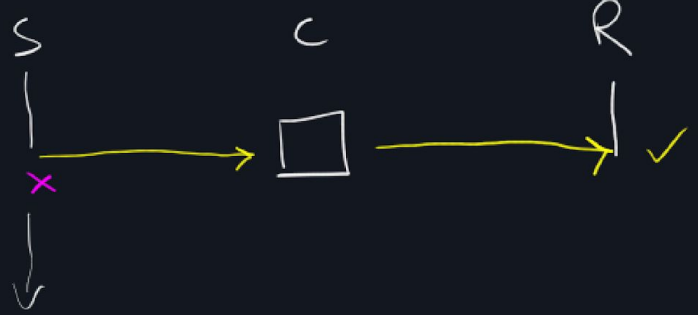
Buffer, capacity of 10 strings

Read/write operations are **non-blocking**

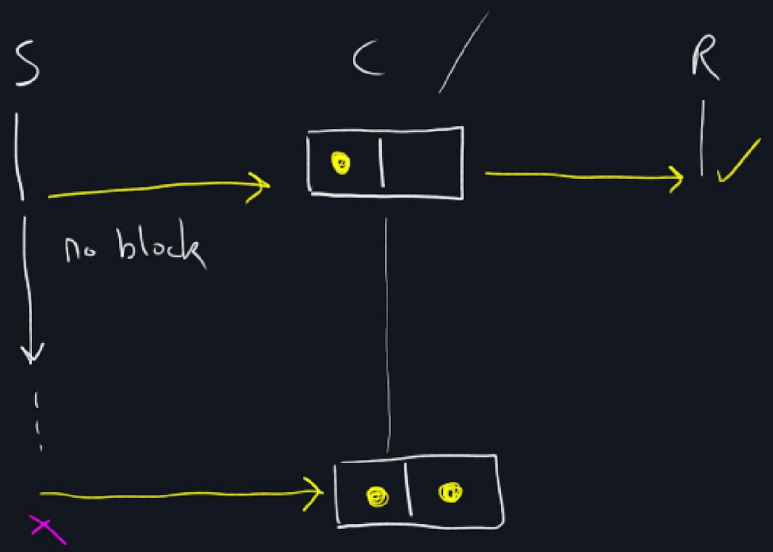
- Read only blocks when channel is empty
- Write blocks only when channel is full

Draw: buffering

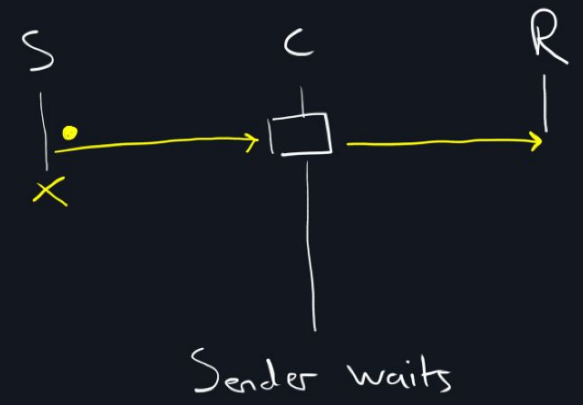
Unbuffered

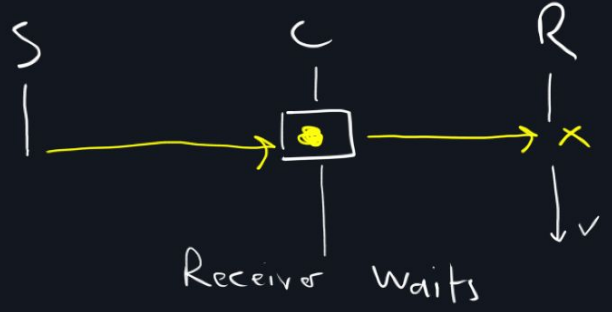


Buffered

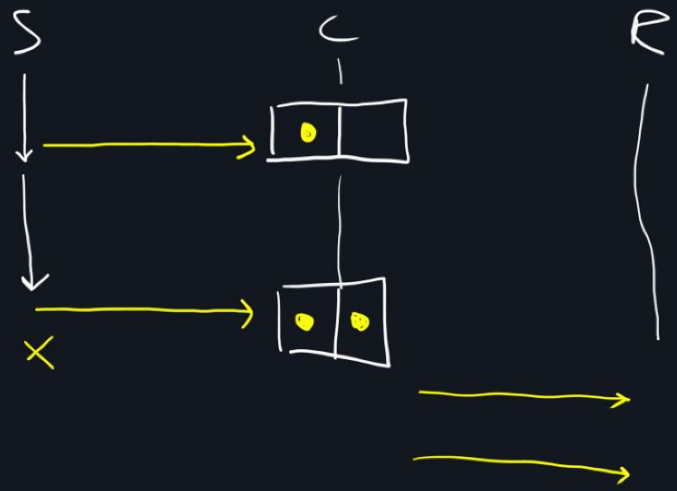


Unbuffered





Buffered



Video

Buffered Pinger

```
func main() {
    // create a channel for communication
     c := make(chan string, 5)

    // create two concurrent threads
    go buffered_pinger(c)
    go printer(c)

    // runs until user input
    var input string
    fmt.Scanln(&input)

}
```

Buffered Pinger and Printer functions

```
func buffered_pinger(c chan string) {
    for i := 0; i < 10; i++ {
        // sender only waits to copy value into channel
        // it only blocks when buffer is full
        c <- "ping"
    }
    fmt.Println("pinger done.")
}

func printer(c chan string) {
    for {
        msg := <- c
        fmt.Println("printer received:", msg)
        // introduce a short delay to force the buffer to fill
        time.Sleep(time.Millisecond * 100)
    }

}
```

Demo: Buffered Pinger

Coordination

Coordinating goroutines

Differing types of coordination are desirable

- Fire and forget
- Send a signal
- Shared message queues
- Choose between alternatives

#1 Fire and forget

Use buffered channel for asynchronous behaviour

Sending goroutine does not block

```
func worker(c chan string) {
    c <- "message"
    fmt.Println("I'm done!")
}

func main()  {
    messages := make(chan string, 1)

    go worker(messages)

    time.Sleep(time.Second * 1)
}
```

#2 Send a signal

Channels can be used to send signals

- Explicit value acts as signal
- Close channel to send a signal

Explicit signal

```
// simple thread that prints, waits, prints
func worker(done chan bool) {
    fmt.Print("working...")
    time.Sleep(time.Second)
    fmt.Println("done")
    // send a value to notify that we're done
    done <- true
}

func main() {
    // start a worker thread
    // pass it the channel to notify on
    done := make(chan bool)
    go worker(done)

    // block until we receive a notification
    <-done
}
```

Close channel

```
func main() {
    jobs := make(chan int, 5)
    done := make(chan bool)

    go func() {
        for {
            j, more := <-jobs
            if more {
                fmt.Println("received job", j)
            } else {
                fmt.Println("received all jobs")
                done <- true
                return
            }
        }
    }()

    for j := 1; j <= 3; j++ {
        jobs <- j
        fmt.Println("sent job", j)
    }
    close(jobs)
    fmt.Println("sent all jobs")
    <-done
}
```

Close channel
Prevents any further send ops

#3 Shared message queue

Channels **guarantee order** (FIFO)

Many goroutines can share a channel as a queue

One goroutine can process the queue in order

Range function makes it convenient to iterate a channel

```
func main() {

    queue := make(chan string, 10)

    for i := 0; i < 10; i++ {
        go func() {
            queue <- fmt.Sprintf("msg")
        }()
    }

    time.Sleep(time.Second)

    close(queue)

    for elem := range queue {
        fmt.Println(elem)
    }
}
```

#4 Choose between alternatives

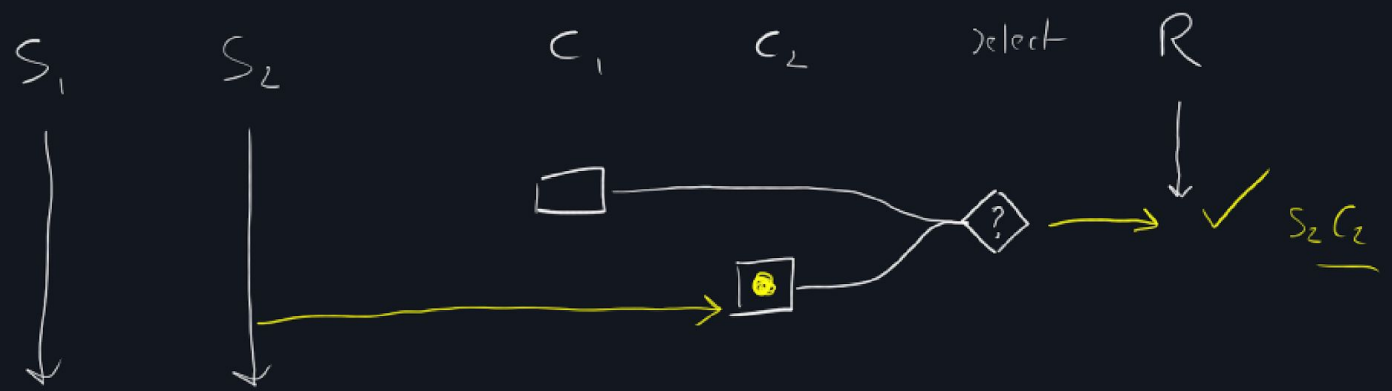
Default is to block when trying to read from a channel

Select statement allows multiple channel reads

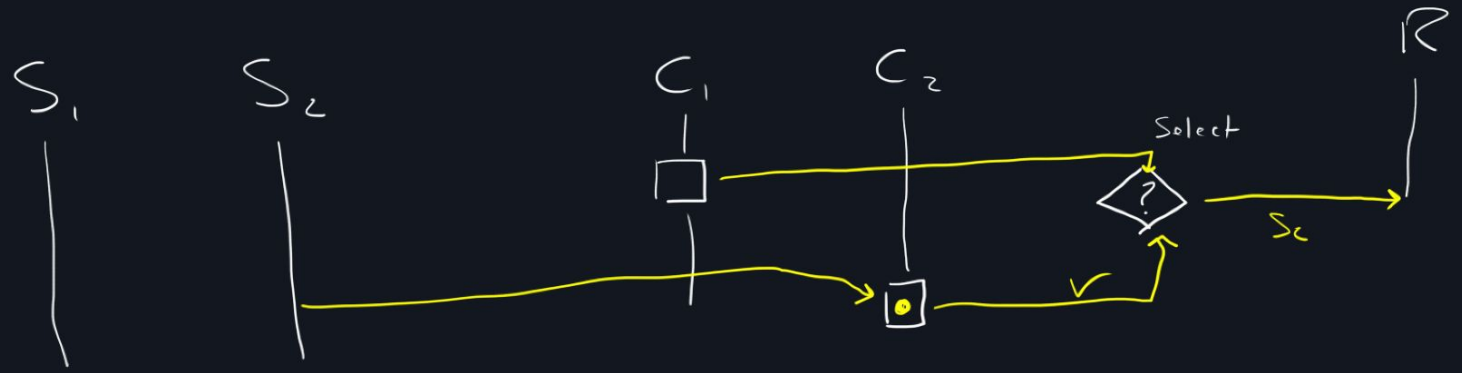
- First available channel is read
- Random pick if multiple channels are ready
- Supports timeouts

Draw: Select

Select



Select



Video

Select statement

```
func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    // sends message into c1 channel, then waits
    go func() {
        for {
            c1 <- "from 1"
            time.Sleep(time.Second * 2)
        }
    }()
    // sends message into c2 channel, then waits
    go func() {
        for {
            c2 <- "from 2"
            time.Sleep(time.Second * 3)
        }
    }()
    // selects from whichever channel has a message to be received
    go func() {
        for {
            select {
            case msg1 := <-c1:
                fmt.Println(msg1)
            case msg2 := <-c2:
                fmt.Println(msg2)
            }
        }
    }()
}
```

Select with timeout

```
func main() {
    c := make(chan string, 1)

    // thread that takes 2 seconds to get a result
    go func() {
        time.Sleep(time.Second * 2)
        c <- "result"
    }()

    // choose whichever channel becomes available first
    select {
        case res := <- c:
            fmt.Println(res)
        case <- time.After(time.Second * 1):
            fmt.Println("timeout!")
    }
}
```

Concept Review

Concept review

Surprisingly sophisticated coordination can be achieved with channels

- Ordered sequence with blocking behaviour
- Buffering to get more done asynchronously
- Coordinate goroutines by sending signals
- Selecting non-blocking alternatives and timeouts

Recommended Reading

Fundamentals of Concurrent Programming

- https://yourbasic.org/golang/channels-explained/
- https://yourbasic.org/golang/select-explained/

Go for Java Programmers

- http://yourbasic.org/golang/go-java-tutorial/

Golang website

- http://golang.org/