**[Go by Example](https://gobyexample.com/): Channels**

|  |  |
| --- | --- |
| *Channels* are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine. |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/MaLY7AiAkHM)https://gobyexample.com/clipboard.png  package main |
|  | import "fmt" |
|  | func main() { |
| Create a new channel with make(chan val-type). Channels are typed by the values they convey. | messages := make(chan string) |
| *Send* a value into a channel using the channel <- syntax. Here we send "ping" to the messages channel we made above, from a new goroutine. | go func() { messages <- "ping" }() |
| The <-channel syntax *receives* a value from the channel. Here we’ll receive the "ping" message we sent above and print it out. | msg := <-messages  fmt.Println(msg)  } |

|  |  |
| --- | --- |
| When we run the program the "ping" message is successfully passed from one goroutine to another via our channel. | **$** go run channels.go  ping |
| By default sends and receives block until both the sender and receiver are ready. This property allowed us to wait at the end of our program for the "ping" message without having to use any other synchronization. |  |

Next example: [Channel Buffering](https://gobyexample.com/channel-buffering).

[**Go by Example**](https://gobyexample.com/)**: Channel Buffering**

|  |  |
| --- | --- |
| By default channels are *unbuffered*, meaning that they will only accept sends (chan <-) if there is a corresponding receive (<- chan) ready to receive the sent value. *Buffered channels* accept a limited number of values without a corresponding receiver for those values. |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/3BRCdRnRszb)https://gobyexample.com/clipboard.png  package main |
|  | import "fmt" |
|  | func main() { |
| Here we make a channel of strings buffering up to 2 values. | messages := make(chan string, 2) |
| Because this channel is buffered, we can send these values into the channel without a corresponding concurrent receive. | messages <- "buffered"  messages <- "channel" |
| Later we can receive these two values as usual. | fmt.Println(<-messages)  fmt.Println(<-messages)  } |

|  |  |
| --- | --- |
|  | **$** go run channel-buffering.go  buffered  channel |

Next example: [Channel Synchronization](https://gobyexample.com/channel-synchronization).

[**Go by Example**](https://gobyexample.com/)**: Channel Synchronization**

|  |  |
| --- | --- |
| We can use channels to synchronize execution across goroutines. Here’s an example of using a blocking receive to wait for a goroutine to finish. When waiting for multiple goroutines to finish, you may prefer to use a [WaitGroup](https://gobyexample.com/waitgroups). |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/Nw-1DzIGk5f)https://gobyexample.com/clipboard.png  package main |
|  | import (  "fmt"  "time"  ) |
| This is the function we’ll run in a goroutine. The done channel will be used to notify another goroutine that this function’s work is done. | 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 goroutine, giving it the channel to notify on. | done := make(chan bool, 1)  go worker(done) |
| Block until we receive a notification from the worker on the channel. | <-done  } |

|  |  |
| --- | --- |
|  | **$** go run channel-synchronization.go  working...done |
| If you removed the <- done line from this program, the program would exit before the worker even started. |  |

[**Go by Example**](https://gobyexample.com/)**: Channel Directions**

|  |  |
| --- | --- |
| When using channels as function parameters, you can specify if a channel is meant to only send or receive values. This specificity increases the type-safety of the program. |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/mjNJDHwUH4R)https://gobyexample.com/clipboard.png  package main |
|  | import "fmt" |
| This ping function only accepts a channel for sending values. It would be a compile-time error to try to receive on this channel. | func ping(pings chan<- string, msg string) {  pings <- msg  } |
| The pong function accepts one channel for receives (pings) and a second for sends (pongs). | func pong(pings <-chan string, pongs chan<- string) {  msg := <-pings  pongs <- msg  } |
|  | func main() {  pings := make(chan string, 1)  pongs := make(chan string, 1)  ping(pings, "passed message")  pong(pings, pongs)  fmt.Println(<-pongs)  } |

|  |  |
| --- | --- |
|  | **$** go run channel-directions.go  passed message |

Next example: [Select](https://gobyexample.com/select).

[**Go by Example**](https://gobyexample.com/)**: Select**

|  |  |
| --- | --- |
| Go’s *select* lets you wait on multiple channel operations. Combining goroutines and channels with select is a powerful feature of Go. |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/FzONhs4-tae)https://gobyexample.com/clipboard.png  package main |
|  | import (  "fmt"  "time"  ) |
|  | func main() { |
| For our example we’ll select across two channels. | c1 := make(chan string)  c2 := make(chan string) |
| Each channel will receive a value after some amount of time, to simulate e.g. blocking RPC operations executing in concurrent goroutines. | go func() {  time.Sleep(1 \* time.Second)  c1 <- "one"  }()  go func() {  time.Sleep(2 \* time.Second)  c2 <- "two"  }() |
| We’ll use select to await both of these values simultaneously, printing each one as it arrives. | for i := 0; i < 2; i++ {  select {  case msg1 := <-c1:  fmt.Println("received", msg1)  case msg2 := <-c2:  fmt.Println("received", msg2)  }  }  } |

|  |  |
| --- | --- |
| We receive the values "one" and then "two" as expected. | **$** time go run select.go  received one  received two |
| Note that the total execution time is only ~2 seconds since both the 1 and 2 second Sleeps execute concurrently. | real 0m2.245s |

Next example: [Timeouts](https://gobyexample.com/timeouts).

[**Go by Example**](https://gobyexample.com/)**: Timeouts**

|  |  |
| --- | --- |
| *Timeouts* are important for programs that connect to external resources or that otherwise need to bound execution time. Implementing timeouts in Go is easy and elegant thanks to channels and select. |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/gyr0NbVKBVf)https://gobyexample.com/clipboard.png  package main |
|  | import (  "fmt"  "time"  ) |
|  | func main() { |
| For our example, suppose we’re executing an external call that returns its result on a channel c1 after 2s. Note that the channel is buffered, so the send in the goroutine is nonblocking. This is a common pattern to prevent goroutine leaks in case the channel is never read. | c1 := make(chan string, 1)  go func() {  time.Sleep(2 \* time.Second)  c1 <- "result 1"  }() |
| Here’s the select implementing a timeout. res := <-c1 awaits the result and <-time.After awaits a value to be sent after the timeout of 1s. Since select proceeds with the first receive that’s ready, we’ll take the timeout case if the operation takes more than the allowed 1s. | select {  case res := <-c1:  fmt.Println(res)  case <-time.After(1 \* time.Second):  fmt.Println("timeout 1")  } |
| If we allow a longer timeout of 3s, then the receive from c2 will succeed and we’ll print the result. | c2 := make(chan string, 1)  go func() {  time.Sleep(2 \* time.Second)  c2 <- "result 2"  }()  select {  case res := <-c2:  fmt.Println(res)  case <-time.After(3 \* time.Second):  fmt.Println("timeout 2")  }  } |

|  |  |
| --- | --- |
| Running this program shows the first operation timing out and the second succeeding. | **$** go run timeouts.go  timeout 1  result 2 |

Next example: [Non-Blocking Channel Operations](https://gobyexample.com/non-blocking-channel-operations).

[**Go by Example**](https://gobyexample.com/)**: Non-Blocking Channel Operations**

|  |  |
| --- | --- |
| Basic sends and receives on channels are blocking. However, we can use select with a default clause to implement *non-blocking* sends, receives, and even non-blocking multi-way selects. |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/TFv6-7OVNVq)https://gobyexample.com/clipboard.png  package main |
|  | import "fmt" |
|  | func main() {  messages := make(chan string)  signals := make(chan bool) |
| Here’s a non-blocking receive. If a value is available on messages then select will take the <-messages case with that value. If not it will immediately take the default case. | select {  case msg := <-messages:  fmt.Println("received message", msg)  default:  fmt.Println("no message received")  } |
| A non-blocking send works similarly. Here msg cannot be sent to the messages channel, because the channel has no buffer and there is no receiver. Therefore the default case is selected. | msg := "hi"  select {  case messages <- msg:  fmt.Println("sent message", msg)  default:  fmt.Println("no message sent")  } |
| We can use multiple cases above the default clause to implement a multi-way non-blocking select. Here we attempt non-blocking receives on both messages and signals. | select {  case msg := <-messages:  fmt.Println("received message", msg)  case sig := <-signals:  fmt.Println("received signal", sig)  default:  fmt.Println("no activity")  }  } |

|  |  |
| --- | --- |
|  | **$** go run non-blocking-channel-operations.go  no message received  no message sent  no activity |

Next example: [Closing Channels](https://gobyexample.com/closing-channels).

[**Go by Example**](https://gobyexample.com/)**: Closing Channels**

|  |  |
| --- | --- |
| *Closing* a channel indicates that no more values will be sent on it. This can be useful to communicate completion to the channel’s receivers. |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/vCvRjcMq7p3)https://gobyexample.com/clipboard.png  package main |
|  | import "fmt" |
| In this example we’ll use a jobs channel to communicate work to be done from the main() goroutine to a worker goroutine. When we have no more jobs for the worker we’ll close the jobs channel. | func main() {  jobs := make(chan int, 5)  done := make(chan bool) |
| Here’s the worker goroutine. It repeatedly receives from jobs with j, more := <-jobs. In this special 2-value form of receive, the more value will be false if jobs has been closed and all values in the channel have already been received. We use this to notify on done when we’ve worked all our jobs. | go func() {  for {  j, more := <-jobs  if more {  fmt.Println("received job", j)  } else {  fmt.Println("received all jobs")  done <- true  return  }  }  }() |
| This sends 3 jobs to the worker over the jobs channel, then closes it. | for j := 1; j <= 3; j++ {  jobs <- j  fmt.Println("sent job", j)  }  close(jobs)  fmt.Println("sent all jobs") |
| We await the worker using the [synchronization](https://gobyexample.com/channel-synchronization) approach we saw earlier. | <-done  } |

|  |  |
| --- | --- |
|  | **$** go run closing-channels.go  sent job 1  received job 1  sent job 2  received job 2  sent job 3  received job 3  sent all jobs  received all jobs |
| The idea of closed channels leads naturally to our next example: range over channels. |  |

Next example: [Range over Channels](https://gobyexample.com/range-over-channels).

[**Go by Example**](https://gobyexample.com/)**: Range over Channels**

|  |  |
| --- | --- |
| In a [previous](https://gobyexample.com/range) example we saw how for and range provide iteration over basic data structures. We can also use this syntax to iterate over values received from a channel. |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/yQMclmwOYs9)https://gobyexample.com/clipboard.png  package main |
|  | import "fmt" |
|  | func main() { |
| We’ll iterate over 2 values in the queue channel. | queue := make(chan string, 2)  queue <- "one"  queue <- "two"  close(queue) |
| This range iterates over each element as it’s received from queue. Because we closed the channel above, the iteration terminates after receiving the 2 elements. | for elem := range queue {  fmt.Println(elem)  }  } |

|  |  |
| --- | --- |
|  | **$** go run range-over-channels.go  one  two |
| This example also showed that it’s possible to close a non-empty channel but still have the remaining values be received. |  |

Next example: [Timers](https://gobyexample.com/timers).

[**Go by Example**](https://gobyexample.com/)**: Timers**

|  |  |
| --- | --- |
| We often want to execute Go code at some point in the future, or repeatedly at some interval. Go’s built-in *timer* and *ticker* features make both of these tasks easy. We’ll look first at timers and then at [tickers](https://gobyexample.com/tickers). |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/gF7VLRz3URM)https://gobyexample.com/clipboard.png  package main |
|  | import (  "fmt"  "time"  ) |
|  | func main() { |
| Timers represent a single event in the future. You tell the timer how long you want to wait, and it provides a channel that will be notified at that time. This timer will wait 2 seconds. | timer1 := time.NewTimer(2 \* time.Second) |
| The <-timer1.C blocks on the timer’s channel C until it sends a value indicating that the timer fired. | <-timer1.C  fmt.Println("Timer 1 fired") |
| If you just wanted to wait, you could have used time.Sleep. One reason a timer may be useful is that you can cancel the timer before it fires. Here’s an example of that. | timer2 := time.NewTimer(time.Second)  go func() {  <-timer2.C  fmt.Println("Timer 2 fired")  }()  stop2 := timer2.Stop()  if stop2 {  fmt.Println("Timer 2 stopped")  } |
| Give the timer2 enough time to fire, if it ever was going to, to show it is in fact stopped. | time.Sleep(2 \* time.Second)  } |

|  |  |
| --- | --- |
| The first timer will fire ~2s after we start the program, but the second should be stopped before it has a chance to fire. | **$** go run timers.go  Timer 1 fired  Timer 2 stopped |

Next example: [Tickers](https://gobyexample.com/tickers).

[**Go by Example**](https://gobyexample.com/)**: Tickers**

|  |  |
| --- | --- |
| [Timers](https://gobyexample.com/timers) are for when you want to do something once in the future - *tickers* are for when you want to do something repeatedly at regular intervals. Here’s an example of a ticker that ticks periodically until we stop it. |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/gs6zoJP-Pl9)https://gobyexample.com/clipboard.png  package main |
|  | import (  "fmt"  "time"  ) |
|  | func main() { |
| Tickers use a similar mechanism to timers: a channel that is sent values. Here we’ll use the select builtin on the channel to await the values as they arrive every 500ms. | ticker := time.NewTicker(500 \* time.Millisecond)  done := make(chan bool) |
|  | go func() {  for {  select {  case <-done:  return  case t := <-ticker.C:  fmt.Println("Tick at", t)  }  }  }() |
| Tickers can be stopped like timers. Once a ticker is stopped it won’t receive any more values on its channel. We’ll stop ours after 1600ms. | time.Sleep(1600 \* time.Millisecond)  ticker.Stop()  done <- true  fmt.Println("Ticker stopped")  } |

|  |  |
| --- | --- |
| When we run this program the ticker should tick 3 times before we stop it. | **$** go run tickers.go  Tick at 2012-09-23 11:29:56.487625 -0700 PDT  Tick at 2012-09-23 11:29:56.988063 -0700 PDT  Tick at 2012-09-23 11:29:57.488076 -0700 PDT  Ticker stopped |

Next example: [Worker Pools](https://gobyexample.com/worker-pools).

[**Go by Example**](https://gobyexample.com/)**: Worker Pools**

|  |  |
| --- | --- |
| In this example we’ll look at how to implement a *worker pool* using goroutines and channels. |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/hiSJJsYZJKL)https://gobyexample.com/clipboard.png  package main |
|  | import (  "fmt"  "time"  ) |
| Here’s the worker, of which we’ll run several concurrent instances. These workers will receive work on the jobs channel and send the corresponding results on results. We’ll sleep a second per job to simulate an expensive task. | func worker(id int, jobs <-chan int, results chan<- int) {  for j := range jobs {  fmt.Println("worker", id, "started job", j)  time.Sleep(time.Second)  fmt.Println("worker", id, "finished job", j)  results <- j \* 2  }  } |
|  | func main() { |
| In order to use our pool of workers we need to send them work and collect their results. We make 2 channels for this. | const numJobs = 5  jobs := make(chan int, numJobs)  results := make(chan int, numJobs) |
| This starts up 3 workers, initially blocked because there are no jobs yet. | for w := 1; w <= 3; w++ {  go worker(w, jobs, results)  } |
| Here we send 5 jobs and then close that channel to indicate that’s all the work we have. | for j := 1; j <= numJobs; j++ {  jobs <- j  }  close(jobs) |
| Finally we collect all the results of the work. This also ensures that the worker goroutines have finished. An alternative way to wait for multiple goroutines is to use a [WaitGroup](https://gobyexample.com/waitgroups). | for a := 1; a <= numJobs; a++ {  <-results  }  } |

|  |  |
| --- | --- |
| Our running program shows the 5 jobs being executed by various workers. The program only takes about 2 seconds despite doing about 5 seconds of total work because there are 3 workers operating concurrently. | **$** time go run worker-pools.go  worker 1 started job 1  worker 2 started job 2  worker 3 started job 3  worker 1 finished job 1  worker 1 started job 4  worker 2 finished job 2  worker 2 started job 5  worker 3 finished job 3  worker 1 finished job 4  worker 2 finished job 5 |
|  | real 0m2.358s |

Next example: [WaitGroups](https://gobyexample.com/waitgroups).

[**Go by Example**](https://gobyexample.com/)**: WaitGroups**

|  |  |
| --- | --- |
| To wait for multiple goroutines to finish, we can use a *wait group*. |  |
|  | [https://gobyexample.com/play.png](https://go.dev/play/p/S98GjeaGBX0)https://gobyexample.com/clipboard.png  package main |
|  | import (  "fmt"  "sync"  "time"  ) |
| This is the function we’ll run in every goroutine. | func worker(id int) {  fmt.Printf("Worker %d starting\n", id) |
| Sleep to simulate an expensive task. | time.Sleep(time.Second)  fmt.Printf("Worker %d done\n", id)  } |
|  | func main() { |
| This WaitGroup is used to wait for all the goroutines launched here to finish. Note: if a WaitGroup is explicitly passed into functions, it should be done *by pointer*. | var wg sync.WaitGroup |
| Launch several goroutines and increment the WaitGroup counter for each. | for i := 1; i <= 5; i++ {  wg.Add(1) |
| Avoid re-use of the same i value in each goroutine closure. See [the FAQ](https://golang.org/doc/faq#closures_and_goroutines) for more details. | i := i |
| Wrap the worker call in a closure that makes sure to tell the WaitGroup that this worker is done. This way the worker itself does not have to be aware of the concurrency primitives involved in its execution. | go func() {  defer wg.Done()  worker(i)  }()  } |
| Block until the WaitGroup counter goes back to 0; all the workers notified they’re done. | wg.Wait() |
| Note that this approach has no straightforward way to propagate errors from workers. For more advanced use cases, consider using the [errgroup package](https://pkg.go.dev/golang.org/x/sync/errgroup). | } |

|  |  |
| --- | --- |
|  | **$** go run waitgroups.go  Worker 5 starting  Worker 3 starting  Worker 4 starting  Worker 1 starting  Worker 2 starting  Worker 4 done  Worker 1 done  Worker 2 done  Worker 5 done  Worker 3 done |
| The order of workers starting up and finishing is likely to be different for each invocation. |  |

Next example: [Rate Limiting](https://gobyexample.com/rate-limiting).