**Go Channels**

# What are channels in Go?

Go Channels provide concurrency for goroutines. Channels allow goroutines to exchange values with each other by allowing them to send and receive data via the channels, effectively acting as a pipeline between the two goroutines. It is also possible to get a second boolean value during a receive operation that is set to true if the received value was delivered by a successful send operation to the channel. Channels are declared using the chan keyword and the make function, followed by the variable type to be exchanged.

A basic example of a Go channel:



# The Make Function

The make function is especially important with channels, as it allocates space in the heap for the channel. The make function also returns a pointer of the channel’s slice that has been allocated on the heap.

An example of the make function being used to allocate a slice:



# Unbuffered Channels

An unbuffered channel is a channel that does not have a declared capacity. As a result, an unbuffered channel must immediately have a receiver ready; otherwise, the sender routine will be blocked. This allows for synchronous communication between the two routines. Unbuffered channels must also have the send and receive operations in separate channels, as otherwise it will cause deadlock.

An example of an unbuffered channel that causes deadlock:



An example of an unbuffered Go channel *(Note: The basic example is also an unbuffered channel)*:





# Buffered Channels

A buffered channel is a channel that has a specified capacity, which is given as an extra parameter at the channel’s declaration. Unlike an unbuffered channel, a buffered channel does not need an immediate receiver after accepting a sent value. Instead, a buffered channel can accept a limited number of values to be sent to it without a receiver being immediately available. The buffered channel will be blocked only when the buffer capacity is exceeded.

An example of a buffered Go channel:



# For-loop and Channel

Using a for-loop makes it possible to iterate over a buffered channel without needing multiple receive statements. Using the range keyword, it is possible to iterate through each sent value, but the channel must be closed before iterating through the channel, as range only stops when the channel is told to close.

An example of a for-loop iterating through a channel using range:



An example of a for-loop iterating through a channel without using range:



# Channel close

Closing a channel is useful when no more values need to be sent to the channel. Closing a channel also can indicate completeness to the channel’s receivers. Only the sender should be closing the channel, as sending data to a closed channel will cause a panic. It is still possible, however, to read data from a closed channel. It is even required to close a channel before iterating through it using for and range.

An example of closing a Go channel *(Note: the for-loop example is also an example of a channel close)*:



# Select Statement

The select statement makes it possible to wait for multiple channel operations. It can be used with a mixture of send/receive requests, whatever is available first is executed. If two channel operations are available, then select acts like a scheduler, the order of the operations is not guaranteed.

# Select Statement (For placing/sending into channels)

The select statement makes it possible to wait for the first available goroutine that can receive a value. When one goroutine is available to receive a value, the select statement sends a value to the channel with the available goroutine.

Example of the select statement sending data to channels:



# Select Statement (For consuming/receiving from channels)

The select statement can also be used to wait for the first goroutine to send data through a channel. When one goroutine has sent a value, the select statement then receives a value from the channel with the sent value.

Example of a select statement waiting for a received value:  




# Default (Both send/receive)

The default case is useful with select statements, as it will run when no other cases are ready. Having a default case can also prevent deadlock as it allows the main goroutine to keep running without being blocked by select.

An example of the default case being used while sending data to channels:





An example of the default case being used while receiving data from channels:





# Nil Channel Behavior

A nil channel is a channel that is only declared, not created; thus, the value of the channel is nil. It is not possible to send or receive data to a nil channel, as the channel exists only in name, and attempting to do so will result in a deadlock. In addition, attempting to close a nil channel will result in a panic.

An example of a nil channel:





# Nil Channel Behavior (Select Statement)

A nil channel can prove useful when working with select statements. The select statement will not process nil channels, so it is possible to “block” cases from running using nil channels.

An example of a nil channel being used in a select statement:



