GO on the toilet

episode #10

Do not communicate by sharing memory; instead, share memory by communicating

This is one of golang's proverbs. Impact of this sentence is bigger than you might think. What does it even mean? It has a meaning only in a concurrent program where separate goroutines want to share information between each other.

Communication by sharing memory means that many goroutines share access to a data structure (which needs a lock protection). This isn't so bad, right? We are used to it, we do this kind of sharing all the time. It turns out we can do better. We can share memory by communicating. "Why is it better?" you may ask. If you'll think why you need a lock to protect a variable in the first place - the answer is - because it can get changed from many goroutines concurrently. This is where you might end up with data race.
If a variable is used only in one goroutine there is no possibility to have data races at all because single variable is changed in single goroutine. This is simple, yet powerful insight.

```
func main() {
	ch := make(chan int)
	go func() {
		ch <- 1
	}()
	fmt.Println(<-ch)
}
```

How do we communicate then? With **channels**!
Let's have a look at example! Channels are created by calling **`make`** function with **`chan`** keyword and type for which channel is created. We've created channel for integers, which means that this channel can convey integer values between goroutines. If you want to send something to channel you use arrow pointing from value to a channel. In the example **`ch <- 1`** means "*send 1 to the channel ch*". What is important is that sending to the channel blocks execution of a goroutine. It is blocked as long as someone will receive from the channel from another goroutine. Receiving from the channel is done also with the arrow but pointing from channel, not to it. Receiving operation also blocks execution of goroutine until other goroutine will send to it.

To illustrate it let's change order of channel operations and try to run it. What will happen is that runtime will exit with **`fatal error: all goroutines are asleep - deadlock!`** This happens because when main goroutine encounters receive operation it blocks forever. This is important piece of information for us because now we know that channels are not only used to convey values between goroutines - they are crucial for synchronization between code which concurrently runs in many goroutines.

```
func main() {
	ch := make(chan int)
	fmt.Println(<-ch)
	go func() {
		ch <- 1
	}()
}
```

This episode just scratched the surface of beautiful go concurrency model. Next time we will explore another type of channels! Stay tuned!