Recover

This lesson is a guide for programmers to recover their code from panic or any error condition.

WE'LL COVER THE FOLLOWING

What is `recover()` function?

As the name indicates, this built-in function can be used to **recover** from a *panic* or an *error-condition*: it allows a program to regain control of a panicking Go routine, stopping the terminating sequence and resuming normal execution. The `recover` is only useful when called inside a deferred function: it then retrieves the error value passed through the call to `panic`. When used in normal execution, a call to `recover` will return *nil* and have no other effect. The `panic` causes the stack to unwind until a deferred `recover()` is found or the program terminates.

Explanation

The `protect` function in the example below invokes the function argument `g` and protects callers from run-time panics raised by `g`, showing the message `x` of the panic:

```
func protect(g func()) {
  defer func() {
    log.Println("done") // Println executes normally even if there is a pa
nic
    if err := recover(); err != nil {
      log.Printf("run time panic: %v", err)
    }
  }()
  log.Println("start")
```

```
  g() // possible runtime-error
}
```

It is analogous to the *catch* block in the Java and .NET languages. The `log` implements a simple *logging* package. The default logger writes to standard error and prints the date and time of each logged message. Apart from the `Println` and `Printf` functions, the *Fatal functions* call `os.Exit(1)` after writing the log message; `Exit` functions identically. The `Panic` functions call panic after writing the log message; use this when a critical condition occurs and the program must be stopped, like in the case when a web server could not be started.

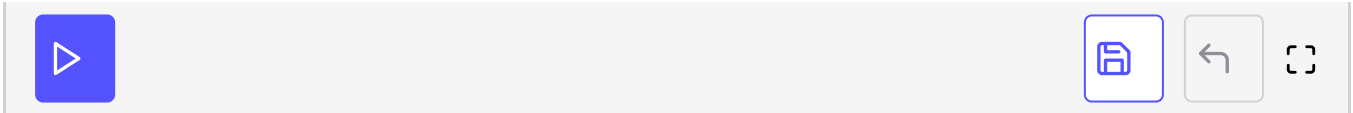
The `log` package also defines an interface type `Logger` with the same methods when you want to define a customized logging system. Here is a complete example which illustrates how `panic`, `defer` and `recover` work together:

```
package main
import (
  "fmt"
)

func badCall() {
  panic("bad end")
}

func test() {
  defer func() {
    if e := recover(); e != nil {
      fmt.Printf("Panicking %s\r\n", e);
    }
  }()
  badCall()
  fmt.Printf("After bad call\r\n");
}

func main() {
  fmt.Printf("Calling test\r\n");
  test()
  fmt.Printf("Test completed\r\n");
}
```



Panic and Recover

To follow the flow of the program, look at its output:

```
Calling test
```

```
Panicking bad end
Test completed
```

Let's see how we got here. At the start, **Calling test** will be printed from **line 21** in `main()` function. Then, control goes to **line 22**, where `test()` is called.

This starts with a `defer` of an *anonymous* function (implemented from **line 11** to **line 15**); this will not be executed now. `badCall()` is called at **line 16**, which causes a *panic* at **line 7**, and **After bad call** (from **line 17**) is never printed.

Normally, the program stops here, but if there is a remaining `defer`, this is executed first before the panic starts its actions. At **line 12**, the `recover` stops the panic and stores its error in `e`. Its message is printed at **line 13**, which is the 2nd output line. Because the panic is recovered, `test()` completes normally, and the end message **Test completed** is printed.

Defer, panic and recover in a sense are also control-flow mechanisms, like if, for, etc. This mechanism is used at several places in the Go standard library, e.g., in the `json` package when decoding or in the `regexp` package in the `Compile` function. The convention in the Go libraries is that even when a package uses `panic` internally, a `recover` is done so that its external API still presents explicit error return values.

Until now, we have seen a program panicking in the `main` package and recovering it. What if the code in a user-defined package faces an error that causes it to get stuck. In the next lesson, you'll learn how to handle this situation.