# Run-time Exceptions and Panic

This lesson explains runtime errors in detail and how a program responds to them.

WE'LL COVER THE FOLLOWING



## Run-time panic

When an execution errors occur, such as attempting to index an array out of bounds or a type assertion failing, the Go runtime triggers a *run-time panic* with a value of the interface type `runtime.Error`, and the program crashes with a message of the error. This value has a `RuntimeError()` method, to distinguish it from a normal error.

Panic can also be initiated from code directly: when the error-condition (which we are testing in the code) is so severe and unrecoverable that the program cannot continue, the `panic` function is used, effectively creating a run-time error that will stop the program. It takes one argument of any type, usually a string, to be printed out when the program dies. So, panic resembles the throw statement from Java or C#. The Go runtime takes care to stop the program and issue some debug information. How it works is illustrated below:

```
package main
import "fmt"

func main() {
  fmt.Println("Starting the program")
  panic("A severe error occurred: stopping the program!")
  fmt.Println("Ending the program")
}
```

Panic

**"Ending the program"** is not printed because of `panic`. The call to panic at **line 6** displays the error message and then stops the program.

Here is a concrete example of checking whether the program starts with a known user:

```
var user = os.Getenv("USER")
func check() {
  if user == "" {
    panic("Unknown user: no value for $USER")
  }
}
```

This could be checked in an `init()` function of a package that is imported. `Panic` can also be used in the error-handling pattern when the error must stop the program:

```
if err != nil {
  panic("ERROR occurred: " + err.Error())
}
```

## Go panicking #

If `panic` is called from a nested function, it immediately stops the execution of the current function; all `defer` statements are guaranteed to execute. Then, control is given to the function caller, which receives this call to panic. This bubbles up to the top level, executing defers, and at the top of the stack, the program crashes. The error condition is reported on the command-line using the value given to panic; this termination sequence is called *panicking*.

The standard library contains several functions whose name is prefixed with **Must**, like `regexp.MustCompile` or `template.Must`; these functions `panic()` when converting the string into a regular expression or template produces an error.

---

Of course, taking down a program with panic should not be done lightly, so every effort must be exercised to remedy the situation and let the program continue. In the next lesson, we'll see how to recover a program from panic.