Errors

Handling Errors

In Go there is no `try/catch` paradigm. Errors are simply return values that need to be handled like any other value returned from a function call.

```go
package main

import (
    "log"
    "os"
)

func main() {
    _, err := os.Open("/path/to/some/content.file")
    if err != nil {
        // open /path/to/some/content.file: no such file or directory
        log.Fatal(err)
    }
}
```

Handling Errors

Because errors are values, we can handle them gracefully in our programs.

```go
var f io.Reader
var err error

// try to read a file
f, err = os.Open("/path/to/some/content.file")
if err != nil {
    // create a fall back io.Reader so our program works
    f = bytes.NewBufferString("some fall back content")
}

b, err := ioutil.ReadAll(f)
if err != nil {
    log.Fatal(err)
}
fmt.Println(string(b))
```

Returning Errors

Errors are returned from functions just like any other type.

```go
func boom() error {
    return errors.New("boom!")
}

func greetOrBoom() (string, error) {
    return "hello", errors.New("boom!")
}
```

Ø

Returning Errors

There are two "built-in" ways to create errors in the standard libary.

```
// in the `errors` package
errors.New("a message")
```

```
// in the `fmt` package
fmt.Errorf("a %s message", "formatted")
```

Errors As Interfaces

Errors in Go are defined as an interface.

```go
type error interface {
  Error() string
}
```

Custom Error Types

By implementing the `error` interface we can make any type an error.

```go
type Yoko struct{}

func (Yoko) Error() string {
    return "i broke up the Beatles"
}
```

Ø

Custom Error Types

```go
func main() {
    for _, b := range []string{"Paul", "George", "John", "Ringo"} {
        err := play(b)
        if err != nil {
            fmt.Println(err)
            break
        }
    }
    // Paul
    // George
    // i broke up the Beatles
}

func play(b string) error {
    if b == "John" {
        return Yoko{}
    }
    fmt.Println(b)
    return nil
}
```

Ø

Panics

Occasionally in your code you will do something that the Go runtime does not like.

```
a := []string{}
a[42] = "Bring a towel"
```

This code will `panic` and your application will crash.

```
panic: runtime error: index out of range

goroutine 1 [running]:
main.main()
    panic.go:5 +0x11
```

Recover From A Panic

With a combination of the `defer` keyword and the the `recover` function we can recover gracefully from panics in our applications and gracefully handle them.

```go
package main

import "fmt"

func main() {
    defer func() {
        if err := recover(); err != nil {
            fmt.Println(err)
        }
    }()
    a := []string{}
    a[42] = "Bring a towel"
}
```

Don't Panic

While it is possible to raise a `panic` in your application, you should absolutely **NEVER** do this!

To `panic` is considered to be non-idiomatic. The correct solution when things go bad is to return an `error` and let others figure out how to handle it in their applications.

Don't Just Check Errors...

Dave Cheney shows how to assert errors for `Behavior`, not `Type` Don't just check errors, handle them gracefully

Exercises

```go
package main

import "fmt"

// Task: Implement the error interface on the Command type

type Command struct {
    ID     int
    Result string
}

func main() {
    fmt.Println("Starting")
    if err := process(); err != nil {
        fmt.Println(err)
    }
    fmt.Println("Completed")
}

func process() error {
    c := Command{ID: 1, Result: "unable to initialize command"}
    return c
}
```

play

Solution

```go
package main

import "fmt"

type Command struct {
    ID     int
    Result string
}

func (c Command) Error() string {
    return fmt.Sprintf("%s %d", c.Result, c.ID)
}

func main() {
    fmt.Println("Starting")
    if err := process(); err != nil {
        fmt.Println(err)
    }
    fmt.Println("Completed")
}

func process() error {
    c := Command{ID: 1, Result: "unable to initialize command"}
    return c
}
```

play

Extra Credit

Use the `switch` statement and `type` keyword to handle different error types differently