Error-Handling and Panicking in a User-Defined Package

This lesson provides an implementation and a detailed explanation about catching errors in custom packages and recovering programs in case of a panic.

Here are a couple of best practices which every writer of custom packages should apply:

- Always recover from panic in your package: no explicit `panic()` should be allowed to cross a package boundary.
- Return errors as error values to the callers of your package.

This is nicely illustrated in the following code:

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package parse
import (
        "fmt"
        "strings"
        "strconv"
)

// A ParseError indicates an error in converting a word into an integer.
type ParseError struct {
        Index int      // The index into the space-separated list of words.
        Word  string   // The word that generated the parse error.
        Err error      // The raw error that precipitated this error, if any.
}

// String returns a human-readable error message.
func (e *ParseError) String() string {
        return fmt.Sprintf("pkg parse: error parsing %q as int", e.Word)
}

// Parse parses the space-separated words in in put as integers.
```

```
func Parse(input string) (numbers []int, err error) {
        defer func() {
                if r := recover(); r != nil {

                        var ok bool
                        err, ok = r.(error)
                        if !ok {
                                err = fmt.Errorf("pkg: %v", r)
                        }
                }
        }()

        fields := strings.Fields(input)
        numbers = fields2numbers(fields)
        return
}

func fields2numbers(fields []string) (numbers []int) {
        if len(fields) == 0 {
                panic("no words to parse")
        }
        for idx, field := range fields {
                num, err := strconv.Atoi(field)
                if err != nil {
                        panic(&ParseError{idx, field, err})
                }
                numbers = append(numbers, num)
        }
        return
}
```

In **parse.go**, we implement a simple version of a `parse` package. From **line 9** to **line 13**, we define a `ParseError` type (see the comments in the code for more info). Then, we have a `String()` method (from **line 16** to **line 18**) to display the error info.

Now, look at the header of the `Parse` function at **line 21**. It takes a string `input`, and returns a slice of *int* and `nil` or a possible error. In other words, the `input` is supposed to be a number of integers, and we transform the `input` string to that. The important lines in the `Parse` function are **line 32** and **line 33**:

- At **line 32**, `strings.Fields` splits the `input` around white spaces and returns a slice of substrings.
- At **line 33**, the function `fields2numbers` is called on that slice and converts it to a slice of integers.

This `fields2numbers` function is defined at **line 37**. It iterates over the slice fields, at **line 41**, and converts each field to a number `num` at **line 42**. If `strconv.Atoi` results in an error because the field is not a string, this is handled with an *if* condition (see the implementation from **line 43** and **line** handled with an *if* condition (see the implementation from **line 43** and **line 45**), causing panic and displaying a `ParseError` with the detailed info of the

problem. If everything is ok, `num` is appended to the slice `nums` at **line 46** and returned.

The `Parse` function starts with a `defer` of an anonymous function call (implemented from **line 22** to **line 30**). This tries to recover from any panic that has happened and returns the error that occurred as `err`.

In **main.go**, starting at **line 9**, we use the `parse` package we discussed above. We build a slice of strings to be parsed. A for-loop (from **line 17** to **line 25**), iterates over this slice, applying `Parse` to each slice `ex` at **line 19**. The slice of integers `nums` that is returned is printed at **line 24**, unless there is an error which is handled (from **line 20** to **line 23**). This produces the output you can see in the terminal.

Now that you're familiar with the *defer/panic/recover* mechanisms, in the next lesson, you'll come across *closures* for error-handling purposes.