CHAPTER 9

# Errors

Error handling is one of the biggest challenges for developers moving to Go from other languages. For those used to exceptions, Go's approach feels anachronistic. But solid software engineering principles underlie Go's approach. In this chapter, you'll learn how to work with errors in Go. You'll also take a look at `panic` and `recover`, Go's system for handling errors that should stop execution.

## How to Handle Errors: The Basics

As was covered briefly in Chapter 5, Go handles errors by returning a value of type `error` as the last return value for a function. This is entirely by convention, but it is such a strong convention that it should never be breached. When a function executes as expected, `nil` is returned for the error parameter. If something goes wrong, an error value is returned instead. The calling function then checks the error return value by comparing it to `nil`, handling the error, or returning an error of its own. A simple function with error handling looks like this:

```go
func calcRemainderAndMod(numerator, denominator int) (int, int, error) {
    if denominator == 0 {
        return 0, 0, errors.New("denominator is 0")
    }
    return numerator / denominator, numerator % denominator, nil
}
```

A new error is created from a string by calling the `New` function in the `errors` package. Error messages should not be capitalized nor should they end with punctuation or a newline. In most cases, you should set the other return values to their zero values when a non-nil error is returned. You'll see an exception to this rule when I cover sentinel errors.

Unlike languages with exceptions, Go doesn't have special constructs to detect if an error was returned. When a function returns an error, use an `if` statement to check the error variable to see if it is non-nil:

```go
func main() {
    numerator := 20
    denominator := 3
    remainder, mod, err := calcRemainderAndMod(numerator, denominator)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    fmt.Println(remainder, mod)
}
```

You can try out this code in the *sample_code/error_basics* directory in the Chapter 9 repository.

`error` is a built-in interface that defines a single method:

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

Anything that implements this interface is considered an error. The reason you return `nil` from a function to indicate that no error occurred is that `nil` is the zero value for any interface type.

There are two very good reasons that Go uses a returned error instead of thrown exceptions. First, exceptions add at least one new code path through the code. These paths are sometimes unclear, especially in languages whose functions don't include a declaration that an exception is possible. This produces code that crashes in surprising ways when exceptions aren't properly handled, or, even worse, code that doesn't crash but whose data is not properly initialized, modified, or stored.

The second reason is more subtle but demonstrates how Go's features work together. The Go compiler requires all variables to be read. Making errors returned values forces developers to either check and handle error conditions or make it explicit that they are ignoring errors by using an underscore (_) for the returned error value.

Exception handling may produce shorter code, but having fewer lines doesn't necessarily make code easier to understand or maintain. As you've seen, idiomatic Go favors clear code, even if it takes more lines.

Another thing to note is how code flows in Go. The error handling is indented inside an `if` statement. The business logic is not. This gives a quick visual clue to which code is along the "golden path" and which code is the exceptional condition.

The second situation is a reused err variable. The Go compiler requires every variable to be read at least once. It doesn't require that *every* write to a variable is read. If you use an err variable multiple times, you have to read it only once to make the compiler happy. In "staticcheck" on page 268, you'll see a way to detect this.

## Use Strings for Simple Errors

Go's standard library provides two ways to create an error from a string. The first is the errors.New function. It takes in a string and returns an error. This string is returned when you call the Error method on the returned error instance. If you pass an error to fmt.Println, it calls the Error method automatically:

```
func doubleEven(i int) (int, error) {
    if i % 2 != 0 {
        return 0, errors.New("only even numbers are processed")
    }
    return i * 2, nil
}

func main() {
    result, err := doubleEven(1)
    if err != nil {
        fmt.Println(err) // prints "only even numbers are processed"
    }
    fmt.Println(result)
}
```

The second way is to use the fmt.Errorf function. This function allows you to include runtime information in the error message by using the fmt.Printf verbs to format an error string. Like errors.New, this string is returned when you call the Error method on the returned error instance:

```
func doubleEven(i int) (int, error) {
    if i % 2 != 0 {
        return 0, fmt.Errorf("%d isn't an even number", i)
    }
    return i * 2, nil
}
```

You can find this code in the *sample_code/string_error* directory in the Chapter 9 repository.

## Sentinel Errors

Some errors are meant to signal that processing cannot continue because of a problem with the current state. In his blog post "Don't Just Check Errors, Handle Them

Gracefully", Dave Cheney, a developer who has been active in the Go community for many years, coined the term *sentinel errors* to describe these:

> The name descends from the practice in computer programming of using a specific value to signify that no further processing is possible. So too with Go, we use specific values to signify an error.

Sentinel errors are one of the few variables that are declared at the package level. By convention, their names start with `Err` (with the notable exception of `io.EOF`). They should be treated as read-only; there's no way for the Go compiler to enforce this, but it is a programming error to change their value.

Sentinel errors are usually used to indicate that you cannot start or continue processing. For example, the standard library includes a package for processing ZIP files, `archive/zip`. This package defines several sentinel errors, including `ErrFormat`, which is returned when data that doesn't represent a ZIP file is passed in. Try out this code on The Go Playground or in the *sample_code/sentinel_error* directory in the Chapter 9 repository:

```go
func main() {
    data := []byte("This is not a zip file")
    notAZipFile := bytes.NewReader(data)
    _, err := zip.NewReader(notAZipFile, int64(len(data)))
    if err == zip.ErrFormat {
        fmt.Println("Told you so")
    }
}
```

Another example of a sentinel error in the standard library is `rsa.ErrMessageTooLong` in the `crypto/rsa` package. It indicates that a message cannot be encrypted because it is too long for the provided public key. The sentinel error `context.Canceled` is covered in Chapter 14.

Be sure you need a sentinel error before you define one. Once you define one, it is part of your public API, and you have committed to it being available in all future backward-compatible releases. It's far better to reuse one of the existing ones in the standard library or to define an error type that includes information about the condition that caused the error to be returned (you'll see how to do that in the next section). But if you have an error condition that indicates a specific state has been reached in your application where no further processing is possible and no additional information needs to be used to explain the error state, a sentinel error is the correct choice.

How do you test for a sentinel error? As you can see in the preceding code sample, use == to test whether the error was returned when calling a function whose documentation explicitly says it returns a sentinel error. In "Is and As" on page 214, I discuss how to check for sentinel errors in other situations.

## Using Constants for Sentinel Errors

In "Constant Errors", Dave Cheney proposes that constants would make useful sentinel errors. You'd have a type like this in a package (I'll talk about creating packages in Chapter 10):

```
package consterr

type Sentinel string

func(s Sentinel) Error() string {
    return string(s)
}
```

Then you'd use it like this:

```
package mypkg

const (
    ErrFoo = consterr.Sentinel("foo error")
    ErrBar = consterr.Sentinel("bar error")
)
```

This looks like a function call, but it's actually casting a string literal to a type that implements the `error` interface. Changing the values of `ErrFoo` and `ErrBar` would be impossible. At first glance, this looks like a good solution.

However, this practice isn't considered idiomatic. If you used the same type to create constant errors across packages, two errors would be equal if their error strings are equal. They'd also be equal to a string literal with the same value. Meanwhile, an error created with `errors.New` is equal only to itself or to variables explicitly assigned its value. You almost certainly do not want to make errors in different packages equal to each other; otherwise, why declare two different errors? (You could avoid this by creating a nonpublic error type in every package, but that's a lot of boilerplate.)

The sentinel error pattern is another example of the Go design philosophy. Sentinel errors should be rare, so they can be handled by convention instead of language rules. Yes, they are public package-level variables. This makes them mutable, but it's highly unlikely someone would accidentally reassign a public variable in a package. In short, it's a corner case that is handled by other features and patterns. The Go philosophy is that it's better to keep the language simple and trust the developers and tooling than it is to add features.

So far, all the errors that you've seen are strings. But Go errors can contain more information. Let's see how.

## Errors Are Values

Since `error` is an interface, you can define your own errors that include additional information for logging or error handling. For example, you might want to include a status code as part of the error to indicate the kind of error that should be reported back to the user. This lets you avoid string comparisons (whose text might change) to determine error causes. Let's see how this works. First, define your own enumeration to represent the status codes:

```go
type Status int

const (
    InvalidLogin Status = iota + 1
    NotFound
)
```

Next, define a `StatusErr` to hold this value:

```go
type StatusErr struct {
    Status    Status
    Message   string
}

func (se StatusErr) Error() string {
    return se.Message
}
```

Now you can use `StatusErr` to provide more details about what went wrong:

```go
func LoginAndGetData(uid, pwd, file string) ([]byte, error) {
    token, err := login(uid, pwd)
    if err != nil {
        return nil, StatusErr{
            Status:    InvalidLogin,
            Message: fmt.Sprintf("invalid credentials for user %s", uid),
        }
    }
    data, err := getData(token, file)
    if err != nil {
        return nil, StatusErr{
            Status:    NotFound,
            Message: fmt.Sprintf("file %s not found", file),
        }
    }
    return data, nil
}
```

You can find this code in the *sample_code/custom_error* directory in the Chapter 9 repository.

Even when you define your own custom error types, always use `error` as the return type for the error result. This allows you to return different types of errors from your function and allows callers of your function to choose not to depend on the specific error type.

If you are using your own error type, be sure you don't return an uninitialized instance. Let's see what happens if you do. Try out the following code on The Go Playground or in the *sample_code/return_custom_error* directory in the Chapter 9 repository:

```
func GenerateErrorBroken(flag bool) error {
    var genErr StatusErr
    if flag {
        genErr = StatusErr{
            Status: NotFound,
        }
    }
    return genErr
}

func main() {
    err := GenerateErrorBroken(true)
    fmt.Println("GenerateErrorBroken(true) returns non-nil error:", err != nil)
    err = GenerateErrorBroken(false)
    fmt.Println("GenerateErrorBroken(false) returns non-nil error:", err != nil)
}
```

Running this program produces the following output:

```
true
true
```

This isn't a pointer type versus value type issue; if you declared `genErr` to be of type `*StatusErr`, you'd see the same output. The reason `err` is non-nil is that `error` is an interface. As I discussed in "Interfaces and nil" on page 164, for an interface to be considered `nil`, both the underlying type and the underlying value must be `nil`. Whether or not `genErr` is a pointer, the underlying type part of the interface is not `nil`.

You can fix this in two ways. The most common approach is to explicitly return `nil` for the error value when a function completes successfully:

```
func GenerateErrorOKReturnNil(flag bool) error {
    if flag {
        return StatusErr{
            Status: NotFound,
        }
    }
    return nil
}
```

This has the advantage of not requiring you to read through code to make sure that the error variable on the `return` statement is correctly defined.

Another approach is to make sure that any local variable that holds an `error` is of type `error`:

```
func GenerateErrorUseErrorVar(flag bool) error {
    var genErr error
    if flag {
        genErr = StatusErr{
            Status: NotFound,
        }
    }
    return genErr
}
```

When using custom errors, never define a variable to be of the type of your custom error. Either explicitly return `nil` when no error occurs or define the variable to be of type `error`.

As was covered in "Use Type Assertions and Type Switches Sparingly" on page 170, don't use a type assertion or a type switch to access the fields and methods of a custom error. Instead, use `errors.As`, which is discussed in "Is and As" on page 214.

## Wrapping Errors

When an error is passed back through your code, you often want to add information to it. This can be the name of the function that received the error or the operation it was trying to perform. When you preserve an error while adding information, it is called *wrapping* the error. When you have a series of wrapped errors, it is called an *error tree*.

A function in the Go standard library wraps errors, and you've already seen it. The `fmt.Errorf` function has a special verb, `%w`. Use this to create an error whose formatted string includes the formatted string of another error and which contains the original error as well. The convention is to write `: %w` at the end of the error format string and to make the error to be wrapped the last parameter passed to `fmt.Errorf`.

The standard library also provides a function for unwrapping errors, the `Unwrap` function in the `errors` package. You pass it an error, and it returns the wrapped error if there is one. If there isn't, it returns `nil`. Here's a quick program that demonstrates wrapping with `fmt.Errorf` and unwrapping with `errors.Unwrap`. You can run it on The Go Playground or in the *sample_code/wrap_error* directory in the Chapter 9 repository:

```
func fileChecker(name string) error {
    f, err := os.Open(name)
    if err != nil {
        return fmt.Errorf("in fileChecker: %w", err)
    }
    f.Close()
    return nil
}

func main() {
    err := fileChecker("not_here.txt")
    if err != nil {
        fmt.Println(err)
        if wrappedErr := errors.Unwrap(err); wrappedErr != nil {
            fmt.Println(wrappedErr)
        }
    }
}
```

When you run this program, you see the following output:

```
in fileChecker: open not_here.txt: no such file or directory
open not_here.txt: no such file or directory
```

You don't usually call `errors.Unwrap` directly. Instead, you use `errors.Is` and `errors.As` to find a specific wrapped error. I'll talk about these two functions in the next section.

If you want to wrap an error with your custom error type, your error type needs to implement the method `Unwrap`. This method takes in no parameters and returns an `error`. Here's an update to the error that you defined earlier to demonstrate how this works. You can find it in the *sample_code/custom_wrapped_error* directory in the Chapter 9 repository:

```
type StatusErr struct {
    Status  Status
    Message string
    Err     error
}

func (se StatusErr) Error() string {
    return se.Message
}

func (se StatusErr) Unwrap() error {
    return se.Err
}
```

Now you can use `StatusErr` to wrap underlying errors:

```go
func LoginAndGetData(uid, pwd, file string) ([]byte, error) {
    token, err := login(uid,pwd)
    if err != nil {
        return nil, StatusErr {
            Status: InvalidLogin,
            Message: fmt.Sprintf("invalid credentials for user %s",uid),
            Err: err,
        }
    }
    data, err := getData(token, file)
    if err != nil {
        return nil, StatusErr {
            Status: NotFound,
            Message: fmt.Sprintf("file %s not found",file),
            Err: err,
        }
    }
    return data, nil
}
```

Not all errors need to be wrapped. A library can return an error that means processing cannot continue, but the error message contains implementation details that aren't needed in other parts of your program. In this situation, it is perfectly acceptable to create a brand-new error and return that instead. Understand the situation and determine what needs to be returned.

If you want to create a new error that contains the message from another error, but don't want to wrap it, use `fmt.Errorf` to create an error but use the `%v` verb instead of `%w`:

```go
err := internalFunction()
if err != nil {
    return fmt.Errorf("internal failure: %v", err)
}
```

## Wrapping Multiple Errors

Sometimes a function generates multiple errors that should be returned. For example, if you wrote a function to validate the fields in a struct, it would be better to return an error for each invalid field. Since the standard function signature returns `error` and not `[]error`, you need to merge multiple errors into a single error. That's what the `errors.Join` function is for:

```go
type Person struct {
    FirstName string
    LastName  string
    Age       int
}
```

```go
func ValidatePerson(p Person) error {
    var errs []error
    if len(p.FirstName) == 0 {
        errs = append(errs, errors.New("field FirstName cannot be empty"))
    }
    if len(p.LastName) == 0 {
        errs = append(errs, errors.New("field LastName cannot be empty"))
    }
    if p.Age < 0 {
        errs = append(errs, errors.New("field Age cannot be negative"))
    }
    if len(errs) > 0 {
        return errors.Join(errs...)
    }
    return nil
}
```

You can find this code in the *sample_code/join_error* directory in the Chapter 9 repository.

Another way to merge multiple errors is to pass multiple `%w` verbs to `fmt.Errorf`:

```go
err1 := errors.New("first error")
err2 := errors.New("second error")
err3 := errors.New("third error")
err := fmt.Errorf("first: %w, second: %w, third: %w", err1, err2, err3)
```

You can implement your own `error` type that supports multiple wrapped errors. To do so, implement the `Unwrap` method but have it return `[]error` instead of `error`:

```go
type MyError struct {
    Code   int
    Errors []error
}

type (m MyError) Error() string {
    return errors.Join(m.Errors...).Error()
}

func (m MyError) Unwrap() []error {
    return m.Errors
}
```

Go doesn't support method overloading, so you can't create a single type that provides both implementations of `Unwrap`. Also note that the `errors.Unwrap` function will return `nil` if you pass it an error that implements the `[]error` variant of `Unwrap`. This is another reason you shouldn't call the `errors.Unwrap` function directly.

If you need to handle errors that may wrap zero, one, or multiple errors, use this code as a basis. You can find it in the *sample_code/custom_wrapped_multi_error* directory in the Chapter 9 repository:

```
var err error
err = funcThatReturnsAnError()
switch err := err.(type) {
case interface {Unwrap() error}:
    // handle single error
    innerErr := err.Unwrap()
    // process innerErr
case interface {Unwrap() []error}:
    //handle multiple wrapped errors
    innerErrs := err.Unwrap()
    for _, innerErr := range innerErrs {
        // process each innerErr
    }
default:
    // handle no wrapped error
}
```

Since the standard library doesn't define interfaces to represent errors with either `Unwrap` variant, this code uses anonymous interfaces in a type switch to match the methods and access the wrapped errors. Before writing your own code, see if you can use `errors.Is` and `errors.As` to examine your error trees. Let's see how they work.

## Is and As

Wrapping errors is a useful way to get additional information about an error, but it introduces problems. If a sentinel error is wrapped, you cannot use == to check for it, nor can you use a type assertion or type switch to match a wrapped custom error. Go solves this problem with two functions in the `errors` package, `Is` and `As`.

To check whether the returned error or any errors that it wraps match a specific sentinel error instance, use `errors.Is`. It takes in two parameters: the error being checked and the instance you are comparing it against. The `errors.Is` function returns `true` if any error in the error tree matches the provided sentinel error. You'll write a short program to see `errors.Is` in action. You can run it yourself on The Go Playground or in the *sample_code/is_error* directory in the Chapter 9 repository:

```
func fileChecker(name string) error {
    f, err := os.Open(name)
    if err != nil {
        return fmt.Errorf("in fileChecker: %w", err)
    }
    f.Close()
    return nil
}

func main() {
    err := fileChecker("not_here.txt")
    if err != nil {
        if errors.Is(err, os.ErrNotExist) {
            fmt.Println("That file doesn't exist")
        }
    }
}
```

Running this program produces the following output:

```
That file doesn't exist
```

By default, `errors.Is` uses `==` to compare each wrapped error with the specified error. If this does not work for an error type that you define (for example, if your error is a noncomparable type), implement the `Is` method on your error:

```
type MyErr struct {
    Codes []int
}

func (me MyErr) Error() string {
    return fmt.Sprintf("codes: %v", me.Codes)
}

func (me MyErr) Is(target error) bool {
    if me2, ok := target.(MyErr); ok {
        return slices.Equal(me.Codes, me2.Codes)
    }
    return false
}
```

(The `slices.Equal` function was mentioned back in "Slices" on page 39.)

Another use for defining your own `Is` method is to allow comparisons against errors that aren't identical instances. You might want to pattern match your errors, specifying a filter instance that matches errors that have some of the same fields. Define a new error type, `ResourceErr`:

```
type ResourceErr struct {
    Resource     string
    Code         int
}

func (re ResourceErr) Error() string {
    return fmt.Sprintf("%s: %d", re.Resource, re.Code)
}
```

If you want two `ResourceErr` instances to match when either field is set, you can do so by writing a custom `Is` method:

```
func (re ResourceErr) Is(target error) bool {
    if other, ok := target.(ResourceErr); ok {
        ignoreResource := other.Resource == ""
        ignoreCode := other.Code == 0
        matchResource := other.Resource == re.Resource
        matchCode := other.Code == re.Code
        return matchResource && matchCode ||
            matchResource && ignoreCode ||
            ignoreResource && matchCode
    }
    return false
}
```

Now you can find, for example, all errors that refer to the database, no matter the code:

```
if errors.Is(err, ResourceErr{Resource: "Database"}) {
    fmt.Println("The database is broken:", err)
    // process the codes
}
```

You can see this code on The Go Playground or in the *sample_code/custom_is_error_pattern_match* directory in the Chapter 9 repository.

The `errors.As` function allows you to check whether a returned error (or any error it wraps) matches a specific type. It takes in two parameters. The first is the error being examined, and the second is a pointer to a variable of the type that you are looking for. If the function returns `true`, an error in the error tree was found that matched, and that matching error is assigned to the second parameter. If the function returns `false`, no match was found in the error tree. Try it out with `MyErr`:

```
err := AFunctionThatReturnsAnError()
var myErr MyErr
if errors.As(err, &myErr) {
    fmt.Println(myErr.Codes)
}
```

Note that you use `var` to declare a variable of a specific type set to the zero value. You then pass a pointer to this variable into `errors.As`.

You don't have to pass a pointer to a variable of an error type as the second parameter to `errors.As`. You can pass a pointer to an interface to find an error that meets the interface:

```
err := AFunctionThatReturnsAnError()
var coder interface {
    CodeVals() []int
}
if errors.As(err, &coder) {
    fmt.Println(coder.CodeVals())
}
```

The example uses an anonymous interface, but any interface type is acceptable. You can find both `errors.As` examples in the *sample_code/custom_as_error* directory in the Chapter 9 repository.

If the second parameter to `errors.As` is anything other than a pointer to an error or a pointer to an interface, the method panics.

Just as you can override the default `errors.Is` comparison with an `Is` method, you can override the default `errors.As` comparison with an `As` method on your error. Implementing an `As` method is nontrivial and requires reflection (I will talk about reflection in Go in Chapter 16). You should do it only in unusual circumstances, such as when you want to match an error of one type and return another.

Use `errors.Is` when you are looking for a specific *instance* or specific *values*. Use `errors.As` when you are looking for a specific *type*.

## Wrapping Errors with defer

Sometimes you find yourself wrapping multiple errors with the same message:

```
func DoSomeThings(val1 int, val2 string) (string, error) {
    val3, err := doThing1(val1)
    if err != nil {
        return "", fmt.Errorf("in DoSomeThings: %w", err)
    }
    val4, err := doThing2(val2)
    if err != nil {
        return "", fmt.Errorf("in DoSomeThings: %w", err)
    }
    result, err := doThing3(val3, val4)
    if err != nil {
        return "", fmt.Errorf("in DoSomeThings: %w", err)
    }
    return result, nil
}
```

You can simplify this code by using `defer`:

```
func DoSomeThings(val1 int, val2 string) (_ string, err error) {
    defer func() {
        if err != nil {
            err = fmt.Errorf("in DoSomeThings: %w", err)
        }
    }()
    val3, err := doThing1(val1)
    if err != nil {
        return "", err
    }
    val4, err := doThing2(val2)
    if err != nil {
        return "", err
    }
    return doThing3(val3, val4)
}
```

You have to name the return values so that you can refer to `err` in the deferred function. If you name a single return value, you must name all of them, so you use an underscore here for the string return value that isn't explicitly assigned.

In the `defer` closure, the code checks whether an error was returned. If so, it reassigns the error to a new error that wraps the original error with a message indicating which function detected the error.

This pattern works well when you are wrapping every error with the same message. If you want to customize the wrapping error with more detail, put both the specific and the general message in every `fmt.Errorf`.

## panic and recover

Previous chapters have mentioned panics in passing without going into any details on what they are. A *panic* is similar to an `Error` in Java or Python. It is a state generated by the Go runtime whenever it is unable to figure out what should happen next. This is almost always due to a programming error, like an attempt to read past the end of a slice or passing a negative size to `make`. The Go runtime also panics if it detects bugs in itself, such as the garbage collector misbehaving. However, I've never seen this happen. If there's a panic, blame the runtime last.

As soon as a panic happens, the current function exits immediately, and any `defers` attached to the current function start running. When those `defers` complete, the `defers` attached to the calling function run, and so on, until `main` is reached. The program then exits with a message and a stack trace.

> If there is a panic in a goroutine other than the main goroutine (goroutines are covered in "Goroutines" on page 289), the chain of defers ends at the function used to launch the goroutine. A program exits if *any* goroutine panics without being recovered.

If any situations in your programs are unrecoverable, you can create your own panics. The built-in function `panic` takes one parameter, which can be of any type. Usually, it is a string. Following is a trivial program that panics; you can run it on The Go Playground or in the *sample_code/panic* directory in the Chapter 9 repository:

```go
func doPanic(msg string) {
    panic(msg)
}

func main() {
    doPanic(os.Args[0])
}
```

Running this code produces the following output:

```
panic: /tmpfs/play

goroutine 1 [running]:
main.doPanic(...)
    /tmp/sandbox567884271/prog.go:6
main.main()
    /tmp/sandbox567884271/prog.go:10 +0x5f
```

As you can see, a `panic` prints out its message followed by a stack trace.

Go provides a way to capture a `panic` to provide a more graceful shutdown or to prevent shutdown at all. The built-in `recover` function is called from within a `defer` to check whether a `panic` happened. If there was a `panic`, the value assigned to the `panic` is returned. Once a `recover` happens, execution continues normally. Let's take a look with another sample program. Run it on The Go Playground or in the *sample_code/panic_recover* directory in the Chapter 9 repository:

```go
func div60(i int) {
    defer func() {
        if v := recover(); v != nil {
            fmt.Println(v)
        }
    }()
    fmt.Println(60 / i)
}

func main() {
    for _, val := range []int{1, 2, 0, 6} {
        div60(val)
    }
}
```

There's a specific pattern for using `recover`. You register a function with `defer` to handle a potential `panic`. You call `recover` within an `if` statement and check whether a non-nil value was found. You must call `recover` from within a `defer` because once a `panic` happens, only deferred functions are run.

Running this code produces the following output:

```
60
30
runtime error: integer divide by zero
10
```

Since `recover` uses a non-nil value to detect whether a `panic` happened, a clever reader might raise the question: *What happens if you call `panic(nil)` and there's a `recover`?* In code compiled with Go versions before 1.21, the answer was "nothing great." In those versions, `recover` stops a `panic` from propagating, but no message or data indicates what happened. Starting with Go 1.21, a `panic(nil)` call is identical to `panic(new(runtime.PanicNilError))`.

While `panic` and `recover` look a lot like exception handling in other languages, they are not intended to be used that way. Reserve panics for fatal situations and use `recover` as a way to gracefully handle these situations. If your program panics, be careful about trying to continue executing after the panic. You'll rarely want to keep your program running after a `panic` occurs. If the `panic` was triggered because the computer is out of a resource like memory or disk space, the safest thing to do is use `recover` to log the situation to monitoring software and shut down with `os.Exit(1)`. If a programming error caused the panic, you can try to continue, but you'll likely hit the same problem again. In the preceding sample program, it would be idiomatic to check for division by zero and return an error if one was passed in.

The reason you don't rely on `panic` and `recover` is that `recover` doesn't make clear *what* could fail. It just ensures that *if* something fails, you can print out a message and continue. Idiomatic Go favors code that explicitly outlines the possible failure conditions over shorter code that handles anything while saying nothing.

Using `recover` is recommended in one situation. If you are creating a library for third parties, do not let panics escape the boundaries of your public API. If a `panic` is possible, a public function should use `recover` to convert the `panic` into an error, return it, and let the calling code decide what to do with them.

> While the HTTP server built into Go recovers from panics in handlers, David Symonds said in a GitHub comment that as of 2015, this is now considered a mistake by the Go team.

## Getting a Stack Trace from an Error

One of the reasons that new Go developers are tempted to use `panic` and `recover` is they want to get a stack trace when something goes wrong. By default, Go doesn't provide that. As I've shown, you can use error wrapping to build a call stack by hand, but some third-party libraries with error types generate those stacks automatically (see Chapter 10 to learn how to incorporate third-party code in your program). Cockroachdb provides a third-party library with functions for wrapping errors with stack traces.

By default, the stack trace is not printed out. If you want to see the stack trace, use `fmt.Printf` and the verbose output verb (`%+v`). Check the documentation to learn more.

> When you have a stack trace in your error, the output includes the full path to the file on the computer where the program was compiled. If you don't want to expose the path, use the `-trimpath` flag when building your code. This replaces the full path with the package.

## Exercises

Look at the code in the *sample_code/exercise* directory in the Chapter 9 repository. You are going to modify this code in each of these exercises. It works correctly, but improvements should be made to its error handling.

1. Create a sentinel error to represent an invalid ID. In `main`, use `errors.Is` to check for the sentinel error, and print a message when it is found.
2. Define a custom error type to represent an empty field error. This error should include the name of the empty `Employee` field. In `main`, use `errors.As` to check for this error. Print out a message that includes the field name.
3. Rather than returning the first error found, return back a single error that contains all errors discovered during validation. Update the code in `main` to properly report multiple errors.

## Wrapping Up

This chapter covered errors in Go, what they are, how to define your own, and how to examine them. You also took a look at `panic` and `recover`. The next chapter discusses packages and modules, how to use third-party code in your programs, and how to publish your own code for others to use.