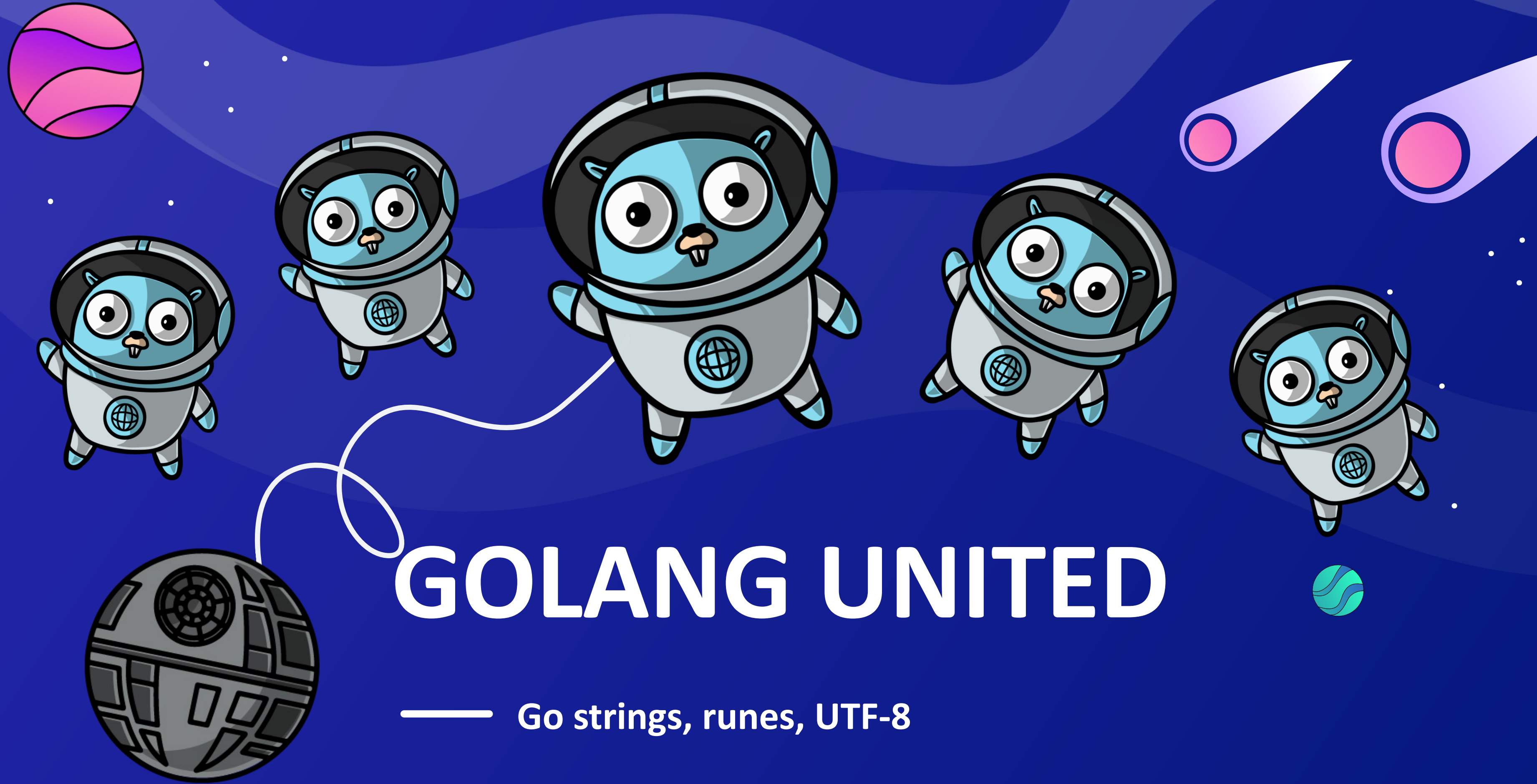


GOLANG UNITED

Go strings, runes, UTF-8

Agenda

Strings and runes

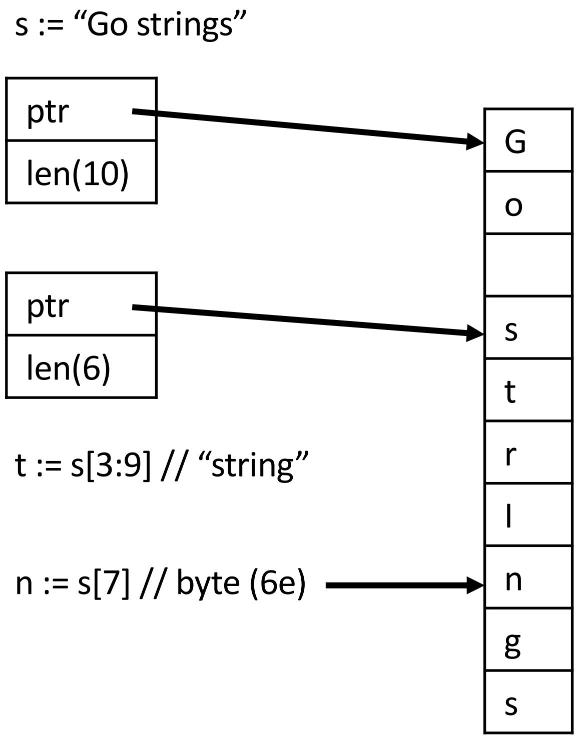
- **Strings** – Go string’s internal structure, basic behavior
- **Runes** – representation of characters with runes and bytes, UTF-8 and Unicode
- ***Strings* package** – standard Go package to manipulate string
- ***Strconv* package** – standard Go package to convert to and from a string

Errors and error handling

- **Errors** – Go errors, panic, recovery and what they are used for
- **Error handling** – common ways of handling errors, don’ts and dos on how to do that properly

STRINGS

Strings



In memory strings are represented as a structure with two fields, it consists of a pointer to the string data(represented as a slice of bytes) and length.

Strings can be indexed, yielding a byte, and sliced, yielding a new string

Creating strings

```go
func main() {
   s1 := "Hello World!"
   fmt.Println(s1)

   s2 := `Hello
            World!`
   fmt.Println(s2)
}
```

Strings are created using **double quotes**.

Alternatively raw strings can be created using **backticks (`)**. In that case the contents are taken literally, preserving new lines and tabs

```
Hello World!
Hello
                                    World!
```

https://go.dev/play/p/KgrbYa8G6tl

Concatenating strings

```
func main() {
   s1 := "Hello"
   s2 := "World!"
   fmt.Println(s1 + " " + s2)
}
```

Two and more strings be concatenated using **+** sign.

```
Hello World!
```

https://go.dev/play/p/W7C62-3ue7b

Comparing strings

```go
func main() {
   s1 := "Hello"
   s2 := "Hello"
   s3 := "hello"
   fmt.Println(s1 == s2)
   fmt.Println(s1 == s3)
}
```

```
true
false
```

https://go.dev/play/p/Knd3-EF-g67

Escape sequences

```
func main() {
   fmt.Println("\"Hitchhiker's Guide to the Galaxy\"\n\n\t\tDouglas Adams")
}
```

```
"Hitchhiker's Guide to the Galaxy"

		Douglas Adams
```

https://go.dev/play/p/F6_gY3zYlGU

Casting string to bytes

```go
func main() {
    str := "Hello"
    bytes := []byte(str)

    fmt.Println(bytes)
}
```

```
[72 101 108 108 111]
```

https://go.dev/play/p/3GueVJ9XLuw

```go
func main() {
    bytes := []byte{72, 101, 108, 108, 111}
    str := string(bytes)

    fmt.Println(str)
}
```

```
Hello
```

https://go.dev/play/p/AhFIAbejPB_g

RUNES

Runes in Go

- Rune is an alias to int32
- Rune represents a Unicode code point (a character, formatting symbol encoded with Unicode)
- Iterating string with *for range* results in runes for each iteration

Indexing strings

```go
func main() {
   s := "Hello"
   for i := 0; i < len(s); i++ {
       fmt.Printf("%s\t%x\t%08b\n", string(s[i]), s[i], s[i])
   }
}
```

```
H	48	01001000
e	65	01100101
l	6c	01101100
l	6c	01101100
o	6f	01101111
```

https://go.dev/play/p/sj0w6LzXqri

Indexing strings

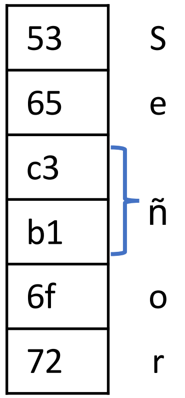
```go
func main() {
   s := "Señor"
   for i := 0; i < len(s); i++ {
    fmt.Printf("%s\t%x\t%08b\n", string(s[i]), s[i], s[i])
   }
}
```

```
S      53     01010011
e      65     01100101
Ã      c3     11000011
±      b1     10110001
o      6f     01101111
r      72     01110010
```

https://go.dev/play/p/cdnAndzDTqm

Unicode characters, beyond the standard ASCII table are encoded usnig UTF-8 with two-four bytes.

You can read about the way UTF-8 is encoded in UTF-8, a transformation format of ISO 10646



Here the symbol is encoded using two bytes

Runes in a string

```go
func main() {
   s := "Señor"
   for _, r := range s {
      fmt.Println(string(r), r)
   }
}
```

```
S 83
e 101
ñ 241
o 111
r 114
```

https://go.dev/play/p/5xHiMLKJj26

```go
func main() {
   s := "Señor"
   runes := []rune(s)
   for i := 0; i < len(runes); i++ {
      fmt.Println(string(runes[i]), runes[i])
   }
}
```

```
S 83
e 101
ñ 241
o 111
r 114
```

https://go.dev/play/p/7DTvU0_9ECm

Len of string vs len of runes

```
func main() {
   s := "Señor"
   runes := []rune(s)
   fmt.Println(len(s))     //number of bytes
   fmt.Println(len(runes)) //number of runes
}
```

```
6
5
```

https://go.dev/play/p/fB3A1cooL_h

Difference between UTF-8 and unicode

UTF-8 is an encoding used to represent the characters in binary.

In other word, it is how the character is stored in bytes

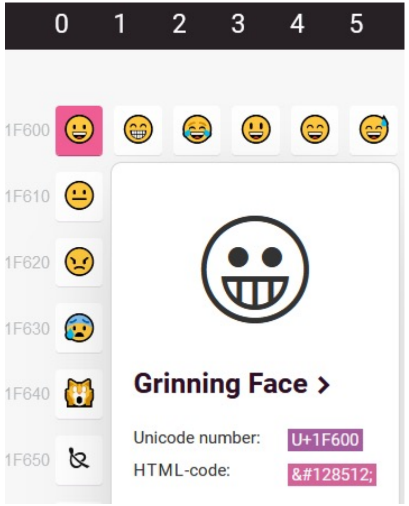
This is how 😀 emoji is encoded in UTF-8

11110000	10011111	10011000	10000000

And this is how this character is stored in **bytes** in Go

Unicode is character set.

Every character in the set is represented by a unique number, which is Go is stored in a **rune**.



https://unicode-table.com/en/#1F600

STRINGS PACKAGE

Strings package

The *strings* package contains functions that allow various manipulations on strings, such as the following:

- changing a string to uppercase/lowercase
- splitting a string into a slice on a delimiter.
- joining the slice of strings into one string with a provided delimiter
- determining if the string has certain prefix/suffix
- discarding a certain prefix/suffix or whitespace
- others

ToUpper/ToLower

```go
func main() {
   fmt.Println(strings.ToLower("Hello"))
   fmt.Println(strings.ToUpper("Hello"))
}
```

```
hello
HELLO
```

https://go.dev/play/p/c6uMGcIg3nl

Split/join

```
func main() {
   path := "api/v1/users"
   pathTokens := strings.Split(path, "/")
   for _, p := range pathTokens {
      fmt.Println(p)
   }
}
```

```
api
v1
users
```

https://go.dev/play/p/70fZy1bQi_6

```
func main() {
   langs := []string{"Go", "C++", "Rust"}
   fmt.Println(strings.Join(langs, ","))
}
```

```
Go,C++,Rust
```

https://go.dev/play/p/uzqHDdR-xVP

HasPrefix/HasSuffix

```go
func main() {
   s := "Hello world"
   fmt.Println(strings.HasPrefix(s, "Hello"))
   fmt.Println(strings.HasPrefix(s, "world"))
   fmt.Println(strings.HasSuffix(s, "Hello"))
   fmt.Println(strings.HasSuffix(s, "world"))
}
```

```
true
false
false
true
```

https://go.dev/play/p/vZVO4Oup_jY

Trim

```go
func main() {
   fmt.Println(strings.TrimLeft("Hello world", "Hello"))
   fmt.Println(strings.TrimRight("Hello world", "world"))
   fmt.Println(strings.TrimSpace("  \n\tHello world   \t\n"))
}
```

```
 world
Hello
Hello world
```

https://go.dev/play/p/_WUfJ0NScnj

STRCONV PACKAGE

Atoi/Itoa

```
func main() {
   s := strconv.Itoa(42)
   fmt.Printf("%s, %T\n", s, s)

   n, err := strconv.Atoi("42")
   fmt.Println(err)
   fmt.Printf("%d, %T\n", n, n)

   n, err = strconv.Atoi("a")
   fmt.Println(err)
   fmt.Printf("%d, %T\n", n, n)
}
```

```
42, string
<nil>
42, int
strconv.Atoi: parsing "a": invalid syntax
0, int
```

https://go.dev/play/p/qs3yu3Fn7rH

Format

```go
func main() {
   s := strconv.FormatInt(42, 10)
   fmt.Printf("%s, %T\n", s, s)
   s = strconv.FormatInt(42, 2)
   fmt.Printf("%s, %T\n", s, s)
   s = strconv.FormatFloat(math.Pi, byte('f'), 4, 64)
   fmt.Printf("%s, %T\n", s, s)
   b := strconv.FormatComplex(complex(3, 4), byte('f'), 2, 64)
   fmt.Printf("%s, %T\n", b, b)
}
```

```
42, string
101010, string
3.1416, string
(3.00+4.00i), string
```

https://go.dev/play/p/-hIU-udt8zw

ParseInt

```
func main() {
    n, _ := strconv.ParseInt("100", 10, 32)
    fmt.Printf("%d, %T\n", n, n)
    n, _ = strconv.ParseInt("100", 2, 32)
    fmt.Printf("%d, %T\n", n, n)
    n, _ = strconv.ParseInt("a", 16, 32)
    fmt.Printf("%d, %T\n", n, n)
    n, err := strconv.ParseInt("a", 10, 32)
    fmt.Println(err)
}
```

```
100, int64
4, int64
10, int64
strconv.ParseInt: parsing "a": invalid syntax
```

https://go.dev/play/p/Y3NXcM2e_I5

ParseFloat/ParseBool/ParseComplex

```
func main() {
   f, _ := strconv.ParseFloat("3.14", 64)
   fmt.Printf("%g, %T\n", f, f)

   b, _ := strconv.ParseBool("true")
   fmt.Printf("%t, %T\n", b, b)

   c, _ := strconv.ParseComplex("4+5i", 64)
   fmt.Printf("%g, %T\n", c, c)
}
```

```
3.14, float64
true, bool
(4+5i), complex128
```

https://go.dev/play/p/2Z3FHu-OIeN

ERRORS

What is an error?

- Go language does not have concept of Exceptions. Instead, any erroneous or abnormal behavior of a program is caught in an error
- Error is a value, which can be stored in a variable and thus can be returned by a function, passed as a parameters etc.

Error example

```go
func main() {
   str := "12c"
   d, err := strconv.Atoi(str)
   if err != nil {
      fmt.Println(err)
   }
   fmt.Println(d)
}
```

```
strconv.Atoi: parsing "12c": invalid syntax
0
```

https://go.dev/play/p/AeUqqPqkurR

Errors (define custom error)

```
type MyError struct {
   Message string
}

func (e MyError) Error() string {
   return e.Message
}

func main() {
   var myErr error = MyError{Message: "my error"}
   var err error = fmt.Errorf("not my error")

   if IsMyError(myErr) {
      fmt.Printf("myErr type is MyError, err - %q\n", myErr.Error())
   } else {
      fmt.Printf("myErr type is not MyError, err - %q\n", myErr.Error())
   }

   if IsMyError(err) {
      fmt.Printf("err type is MyError, err - %q\n", myErr.Error())
   } else {
      fmt.Printf("err type is not MyError, err - %q\n", myErr.Error())
   }
}
```

Error is an interface. To create a custom error type you just need to implement error interface.

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

https://golang.org/src/builtin/builtin.go?s=11196:11236#L250

```
myErr type is MyError, err - "my error"
err type is not MyError, err - "my error"
```

https://go.dev/play/p/mn8cFn7-RqK

Panic

```go
func main() {
   str := "12c"
   d, err := strconv.Atoi(str)
   if err != nil {
      panic(err)
   }
   fmt.Println(d)
}
```

Panic is used to prematurely terminate the execution of program

Panic can be triggered with the *panic()* function.

```
panic: strconv.Atoi: parsing "12c": invalid syntax

goroutine 1 [running]:
main.main()
                    /tmp/sandbox1313570788/prog.go:12 +0x85
```

https://go.dev/play/p/PKuMNIyI7xk

Panic

```go
func main() {
   s := []int{1, 2, 3}

   fmt.Println(s[3])
}
```

Panic can also occur on invalid operation, like trying to access an element of the slice beyond its size, as pictured in the example.

In this case the panic is invoked by Go runtime

```
panic: runtime error: index out of range [3] with length 3

goroutine 1 [running]:
main.main()
	/tmp/sandbox1162838042/prog.go:10 +0x1b
```

https://go.dev/play/p/dt1fnJuSlIG

When does panic should be used?

Ideally never or almost never.

Instead, the error need to be handled by either logging it or returning to be handled by the calling function.

However, there are cases when panic might be necessary:

- **Unrecoverable error where the program cannot continue execution.** This usually happens during initialization. For example, when a web server tries to connect to a database, that is unavailable on a provided host, which means the whole web server will not be able to function properly, and it simply makes sense to terminate the execution of the program.
- **Panic might occur on a programmer error.** An example would be when a programmer passes a nil pointer to a function, which tries to access this pointers parameters, accessing a slice beyond its size, writing on a closed channel, simultaneous write to a map, etc.

Deferred functions during a panic

```go
func main() {
   defer fmt.Println("This line is executed even if panic occurs")
   str := "12c"
   d, err := strconv.Atoi(str)
   if err != nil {
      panic(err)
   }
   fmt.Println(d)
}
```

Panic does not immediately terminate the execution of a program. All deferred function are executed right before the panic occurs

```
This line is executed even if panic occurs
panic: strconv.Atoi: parsing "12c": invalid syntax

goroutine 1 [running]:
main.main()
		/tmp/sandbox2120023744/prog.go:13 +0xf1
```

https://go.dev/play/p/olmB9xTcEA4

Recovery from panic

```
func main() {
   defer func() {
      if err := recover(); err != nil {
         fmt.Printf("recovery from error - %q\n", err)
      }
   }()

   str := "12c"
   d, err := strconv.Atoi(str)
   if err != nil {
      panic(err)
   }

   fmt.Println(d)
}
```

```
recovery from error - "strconv.Atoi: parsing \"12c\": invalid
syntax"
```

In cases when panic is not acceptable, it can be handled using recovery function.

Because after panic only deferred functions are being called, recovery must be handled in deferred function. In this case the execution of the program is not terminated.

https://go.dev/play/p/UwJ643Sp6YS

HANDLING ERRORS

Handling an error

```
return fmt.Errorf("failed to authenticate: %
w", err)
```

Return – The most common way of handling an error. If an error is received the function is interrupted and exits.

The consumer has to deal with that error now.

```
log.Printf("failed to close reader: %v\n", e
rr)
```

Log – usually happens, when the error is not significant enough to interrupt the flow of function. Serves as a warning in the log output.

Also common in panic recovery.

```
panic(fmt.Sprintf("could not connect to the
database: %v", err))
```

Panic – critical error, when the program needs to be terminated immediately

Don’t ignore errors

```
func main(){
   s, _ := GetString()
   //do stuff with the string
}

func GetString() (s string, err error){
   //do stuff
   return
}
```

You can ignore an error using the *blank identifier*.

There is a reason, that the function **GetString()** returns an error. That means something went wrong and our string is most probably blank or not what we expect it to be. This can lead to many problems.

Always handle errors!

Don’t simply return an error

```
func AuthenticateUser(r *http.Request) error {
   err := authenticate(r)
   if err != nil{
      //return err
      return fmt.Errorf("failed to authenticate user: %w", err)
   }
   return nil
}

func authenticate(r *http.Request) error{
   u, err := getUser(r)
   if err != nil{
      // return err
      return fmt.Errorf("failed to get user from request: %w", err)
   }
   // do additional work
   return nil
}
```

It is a good idea to add additional context to an error before returning it.

In the example commented return statements just simply return the error as is. It might be difficult to understand where it originates.

Don’t handle errors multiple times

```
func function() error {
   log.Println(err)
   return err
}

func anotherFunction() error{
   err := function()
   log.Println(err)
   return err
}
```

Here we both log the error and return it, creating the situation, where our error appears in the log twice.

Ideally you want either to return an error, and let the caller decide what to do with it or log it.

Errors – happy path alignment

```go
func CopyFile(dstName, srcName string) (written int64, err error) {
   src, err := os.Open(srcName)
   if err == nil {
      defer src.Close()
      dst, err := os.Create(dstName)
      if err == nil {
         defer dst.Close()
         return io.Copy(dst, src) // happy path
      }
   }
   return 0, nil
}
```

Happy path is a default scenario featuring no exceptional or error conditions – ideal execution of a program.

In the example to the left we see that our happy path is hidden inside nested if conditions, which makes it difficult to read.

This is a bad practice and should be avoided!

Errors – happy path alignment

```
func CopyFile(dstName, srcName string) (written int64, err error) {
   src, err := os.Open(srcName)
   if err != nil {
      return
   }
   defer src.Close()

   dst, err := os.Create(dstName)
   if err != nil {
      return
   }
   defer dst.Close()

   return io.Copy(dst, src) // happy path
}
```

Ideally, we want to align our happy path to the left.

To do that we flip the if statements and first do the checks when error is not nil, returning or logging it, which leaves our happy path to usually be the last return. This reduces nesting and increases readability of the code.

Don’t rely on message

```
func main() {
   data, err := client.Get()
   if err.Error() == fmt.Errorf("i'm an error, handle me").Error() {
      fmt.Printf("catch error. err - %s", err.Error())
      return
   }

   fmt.Println(data)
}

-- client/client.go --
func Get() (int, error) {
   return 0, fmt.Errorf(" i'm an error, handle me")
}
```

Here the error is checked by its text message. We catch the error, since the text matches.

```
catch error. err - i'm an error, handle me
```

https://go.dev/play/p/bupa7txc56U

Don’t rely on message

```
func main() {
   data, err := client.Get()
   if err.Error() == fmt.Errorf("i'm an error, handle me").Error() {
      fmt.Printf("catch error. err - %s", err.Error())
      return
   }

   fmt.Println(data)
}

-- client/client.go --
func Get() (int, error) {
   return 0, fmt.Errorf(" i'm an error, please handle me")
}
```

```
0
```

One day a developer of a library might decide to change the text of the message and make it more polite.

Our code no longer works properly, since the text does not match anymore. A bug has appeared as if out of nowhere.

Never handle errors in such a way! The code might and will be broken

https://go.dev/play/p/ACL6N7Ms8k4

Public error variables

```
func main() {
   data, err := client.Get()
   if err == client.MyCustomError {
      fmt.Printf("catch client.MyCustomError. err - %s", err.Error())
      return
   }

   fmt.Println(data)
}
-- client/client.go --
var MyCustomError = fmt.Errorf("i'm an error, handle me")

func Get() (int, error) {
   return 0, MyCustomError
}
```

```
catch client.MyCustomError. err - i'm an error, handle me
```

Here is a better example, where the error is defined as a public variable.

Still, there are some problems.

- Sometimes an error might not have predefined text and some of its parts are dynamic (for example an http status code)
- Another problem is that every exported error in a package becomes a part of the package API, which needs to be maintained accordingly. Any change in the public error is also a change in API

https://go.dev/play/p/ZjKzYFRy3qp

Encapsulate error check in a function

```go
func main() {
   data, err := client.Get()
   if client.IsCustomError(err) {
      fmt.Printf("catch custom error. err - %s", err.Error())
      return
   }
   fmt.Println(data)
}
-- client/client.go --
var myCustomError = fmt.Errorf("i'm an error, handle me")

func IsCustomError(err error) bool {
   return err == myCustomError
}

func Get() (int, error) {
   return 0, myCustomError
}
```

This example solves the second problem of the previous example – exposing the error.

https://go.dev/play/p/YRQLmHRkM-C

```
catch client.MyCustomError. err - i'm an error, handle me
```

Type casting

```
func main() {
   data, err := client.Get()
   if cErr, ok := err.(client.MyCustomError); ok {
      fmt.Printf("catch client.MyCustomError. err - %s", cErr.Error())
      return
   }
   fmt.Println(data)
}
-- client/client.go --
type MyCustomError struct {
   Message string
}

func (m MyCustomError) Error() string {
   return m.Message
}
```

Type casting is the most flexible and elegant approach. This way developer relies on the type of an error, not it's internal structure. This also allows creating a more complex error, containing dynamic context, and the ability to return this context with helper methods.

It is a good idea to combine this approach with encapsulation described on a previous slide.

https://go.dev/play/p/bfbjZZDnn9x

```
catch client.MyCustomError. err - i'm an error, handle me
```

Functions introduced in Go 1.13

```
func main() {
   err := MyCustomError{
      msg: "Hello",
      err: MyCustomError{msg: "World"},
   }
   fmt.Println("errors.Is:", errors.Is(err, MyCustomError{
      msg: "World",
   }))

   fmt.Println("err.Unwrap:", err.Unwrap())

   err3 := &MyCustomError{}
   fmt.Println("errors.As(err, err3):", errors.As(err, err3))
}
```

```
errors.Is true
err.Unwrap Hello:
               World
errors.As(err, err3) true
```

These functions allow to more deeply examine errors

func Is(err, target error) bool
func As(err error, target interface{}) bool
func Unwrap(err error) error

https://go.dev/play/p/WcA20fLqqrS

Functions introduced in Go 1.13

```go
func main() {
    err1 := errors.New("my error")
    err2 := fmt.Errorf("an error occurred:
%w", err1)

    fmt.Println("err2:", err2)

    fmt.Println("errors.Is(err2, err1):",
errors.Is(err2, err1))

    e := errors.Unwrap(err2)

    fmt.Println("e == err1:", e == err1)
}
```

Click to add text

```
err2: an error occurred: my error
errors.Is(err2, err1): true
e == err1: true
```

fmt.Errorf function allows us to wrap an error into a new one.

Formatting the new error is similar to **fmt.Sprintf**, but has an extra verb **%w**.

To benefit from functions like **errors.As**, **errors.Is** and **errors.Unwrap**, you need to use **%w** and only use it once.

https://go.dev/play/p/idab1CmQDUE

Questions

Homework

Homework

Task 1: Strings and runes

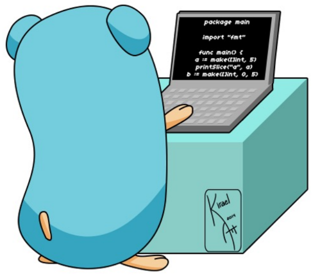
Implement a function, that reverses the input string
Consider that the string may have international symbols, emojis and consist of several lines
Example:
input: Hello world!
output: !dlrow olleH

Task 2: String packages and errors

Implement a function that computes the sum of two int numbers written as a string
For example, having an input string "3+5", it should return output string "8" and nil error
Consider cases, when operands are negative ("-3+5" or "-3-5") and when input string contains whitespace (" 3 + 5 ")

For the cases, when the input expression is not valid(contains characters, that are not numbers, +, - or whitespace) the function should return an empty string and an appropriate error from strconv package wrapped into your own error with fmt.Errorf function

Use the provided errors appropriately for cases when the input string is empty(contains not character, but whitespace) or an expression contains one or greater than two operands, again wrapping into fmt.Errorf



Thanks