Welcome!

Learn how to use this tour: including how to navigate the different lessons and how to run code.

The Go Authors

https://golang.org

*** Hello, ..**

Welcome to a tour of the [[/][Go programming language]].

The tour is divided into a list of modules that you can

access by clicking on

[[javascript:highlight(".logo")][A Tour of Go]] on the top left of the page.

You can also view the table of contents at any time by clicking on the [[javascript:highlightAndClick(".nav")][menu]] on the top right of the page.

Throughout the tour you will find a series of slides and exercises for you

to complete.

You can navigate through them using

- [[javascript:highlight(".prev-page")]["previous"]] or `PageUp` to go to the previous page,

- [[javascript:highlight(".next-page")]["next"]] or `PageDown` to go to the next page.

The tour is interactive. Click the

[[javascript:highlightAndClick("#run")][Run]] button now

(or press `Shift` + `Enter`) to compile and run the program on

your computer.

The result is displayed below the code.

These example programs demonstrate different aspects of Go. The programs in the tour are meant to be starting points for your own experimentation.

Edit the program and run it again.

When you click on [[javascript:highlightAndClick("#format")][Format]]

(shortcut: `Ctrl` + `Enter`), the text in the editor is formatted using the

[[/cmd/gofmt/][gofmt]] tool. You can switch syntax highlighting on and off

by clicking on the [[javascript:highlightAndClick(".syntax-checkbox")][syntax]] button.

When you're ready to move on, click the [[javascript:highlightAndClick(".next-page")][right arrow]] below or type the `PageDown` key.

*** Go local**

The tour is available in other languages:

- [[https://go-tour-br.appspot.com/][Brazilian Portuguese — Português do Brasil]]

- [[https://go-tour-ca.appspot.com/][Catalan — Català]]

- [[https://tour.go-zh.org/][Simplified Chinese — ......]]

- [[https://go-tour-cz.appspot.com/][Czech — .esky]]

- [[https://go-tour-id2.appspot.com/][Indonesian — Bahasa Indonesia]]

- [[https://go-tour-jp.appspot.com/][Japanese — ...]]

- [[https://go-tour-ko.appspot.com/][Korean — ...]]

- [[https://go-tour-pl1.appspot.com/][Polish — Polski]]

- [[https://go-tour-lat.appspot.com/][Spanish — Español]]

- [[https://go-tour-th.appspot.com/][Thai — ....]]

- [[https://go-tour-ua-translation.lm.r.appspot.com/][Ukrainian — ....]]

Click the [[javascript:highlightAndClick(".next-page")]["next"]] button or type `PageDown` to continue.

*** Congratulations**

You've finished the first module of the tour!

Now click on [[javascript:highlightAndClick(".logo")][A Tour of Go]] to find out what else
you can learn about Go, or go directly to the [[javascript:click('.next-page')][next lesson]].

Packages, variables, and functions.

Learn the basic components of any Go program.

The Go Authors

https://golang.org

*** Packages**

Every Go program is made up of packages.

Programs start running in package `main`.

This program is using the packages with import paths `"fmt"` and `"math/rand"`.

By convention, the package name is the same as the last element of the import path. For instance, the `"math/rand"` package comprises files that begin with the statement `package`rand`.

*** Imports**

This code groups the imports into a parenthesized, "factored" import statement.

You can also write multiple import statements, like:

```
 import "fmt"

 import "math"
```

But it is good style to use the factored import statement.

*** Exported names**

In Go, a name is exported if it begins with a capital letter.

For example, `Pizza` is an exported name, as is `Pi`, which is exported from the `math` package.

`pizza` and `pi` do not start with a capital letter, so they are not exported.

When importing a package, you can refer only to its exported names.

Any "unexported" names are not accessible from outside the package.

Run the code. Notice the error message.

To fix the error, rename `math.pi` to `math.Pi` and try it again.

*** Functions**

A function can take zero or more arguments.

In this example, `add` takes two parameters of type `int`.

Notice that the type comes _after_ the variable name.

(For more about why types look the way they do, see the [[/blog/gos-declaration-syntax][article on Go's declaration syntax]].)

*** Functions continued**

When two or more consecutive named function parameters share a type, you can omit the type from all but the last.

In this example, we shortened

```
 x int, y int
```

to

 x, y int

*** Multiple results**

A function can return any number of results.

The `swap` function returns two strings.

*** Named return values**

Go's return values may be named. If so, they are treated as variables defined at the top of the function.

These names should be used to document the meaning of the return values.

A `return` statement without arguments returns the named return values. This is known as a "naked" return.

Naked return statements should be used only in short functions, as with the example shown here. They can harm readability in longer functions.

*** Variables**

The `var` statement declares a list of variables; as in function argument lists, the type is last.

A `var` statement can be at package or function level. We see both in this example.

*** Variables with initializers**

A var declaration can include initializers, one per variable.

If an initializer is present, the type can be omitted; the variable will take the type of the initializer.

*** Short variable declarations**

Inside a function, the `:=` short assignment statement can be used in place of a `var` declaration with implicit type.

Outside a function, every statement begins with a keyword (`var`, `func`, and so on) and so the `:=` construct is not available.

*** Basic types**

Go's basic types are

```
 bool

 string

 int  int8  int16  int32  int64
 uint uint8 uint16 uint32 uint64 uintptr

 byte // alias for uint8

 rune // alias for int32
      // represents a Unicode code point

 float32 float64
```

complex64 complex128

The example shows variables of several types,

and also that variable declarations may be "factored" into blocks,

as with import statements.

The `int`, `uint`, and `uintptr` types are usually 32 bits wide on 32-bit systems and 64 bits wide on 64-bit systems.
When you need an integer value you should use `int` unless you have a specific reason to use a sized or unsigned integer type.

*** Zero values**

Variables declared without an explicit initial value are given their

_zero_value_.

The zero value is:

- `0` for numeric types,

- `false` for the boolean type, and

- `""` (the empty string) for strings.

*** Type conversions**

The expression `T(v)` converts the value `v` to the type `T`.

Some numeric conversions:

```
 var i int = 42

 var f float64 = float64(i)

 var u uint = uint(f)
```

Or, put more simply:

```
 i := 42

 f := float64(i)

 u := uint(f)
```

Unlike in C, in Go assignment between items of different type requires an

explicit conversion.

Try removing the `float64` or `uint` conversions in the example and see what happens.

*** Type inference**

When declaring a variable without specifying an explicit type (either by using the `:=` syntax or `var`=` expression syntax), the variable's type is inferred from the value on the right hand side.

When the right hand side of the declaration is typed, the new variable is of that same type:

```
 var i int

 j := i // j is an int
```

But when the right hand side contains an untyped numeric constant, the new variable may be an `int`, `float64`, or `complex128` depending on the precision of the constant:

```
 i := 42           // int

 f := 3.142        // float64

 g := 0.867 + 0.5i // complex128
```

Try changing the initial value of `v` in the example code and observe how its type is affected.

*** Constants**

Constants are declared like variables, but with the `const` keyword.

Constants can be character, string, boolean, or numeric values.

Constants cannot be declared using the `:=` syntax.

*** Numeric Constants**

Numeric constants are high-precision _values_.

An untyped constant takes the type needed by its context.

Try printing `needInt(Big)` too.

(An `int` can store at maximum a 64-bit integer, and sometimes less.)

*** Congratulations!**

You finished this lesson!

You can go back to the list of [[/tour/list][modules]] to find what to learn next, or continue with the [[javascript:click('.next-page')][next lesson]].

Flow control statements: for, if, else, switch and defer

Learn how to control the flow of your code with conditionals, loops, switches and defers.

The Go Authors

https://golang.org

*** For**

Go has only one looping construct, the `for` loop.

The basic `for` loop has three components separated by semicolons:

- the init statement: executed before the first iteration
- the condition expression: evaluated before every iteration
- the post statement: executed at the end of every iteration

The init statement will often be a short variable declaration, and the

variables declared there are visible only in the scope of the `for`

statement.

The loop will stop iterating once the boolean condition evaluates to `false`.

***Note:* Unlike other languages like C, Java, or JavaScript there are no parentheses**
surrounding the three components of the `for` statement and the braces `{`}` are

always required.

*** For continued**

The init and post statements are optional.

*** For is Go's "while"**

At that point you can drop the semicolons: C's `while` is spelled `for` in Go.

*** Forever**

If you omit the loop condition it loops forever, so an infinite loop is compactly expressed.

*** If**

Go's `if` statements are like its `for` loops; the expression need not be

surrounded by parentheses `(`)` but the braces `{`}` are required.

*** If with a short statement**

Like `for`, the `if` statement can start with a short statement to execute before the condition.

Variables declared by the statement are only in scope until the end of the `if`.

(Try using `v` in the last `return` statement.)

*** If and else**

Variables declared inside an `if` short statement are also available inside any

of the `else` blocks.

(Both calls to `pow` return their results before the call to `fmt.Println` in `main` begins.)

*** Exercise: Loops and Functions**

As a way to play with functions and loops, let's implement a square root function: given a number x, we want to find the number z for which z^2 is most nearly x.

Computers typically compute the square root of x using a loop.

Starting with some guess z, we can adjust z based on how close z^2 is to x,

producing a better guess:

```
 z -= (z*z - x) / (2*z)
```

Repeating this adjustment makes the guess better and better

until we reach an answer that is as close to the actual square root as can be.

Implement this in the `func`Sqrt` provided.

A decent starting guess for z is 1, no matter what the input.

To begin with, repeat the calculation 10 times and print each z along the way.

See how close you get to the answer for various values of x (1, 2, 3, ...)

and how quickly the guess improves.

Hint: To declare and initialize a floating point value,

give it floating point syntax or use a conversion:

```
 z := 1.0

 z := float64(1)
```

Next, change the loop condition to stop once the value has stopped

changing (or only changes by a very small amount).

See if that's more or fewer than 10 iterations.

Try other initial guesses for z, like x, or x/2.

How close are your function's results to the [[/pkg/math/#Sqrt][math.Sqrt]] in the standard library?

(*Note:* If you are interested in the details of the algorithm, the z^2 . x above

is how far away z^2 is from where it needs to be (x), and the division by 2z is the derivative
of z^2, to scale how much we adjust z by how quickly z^2 is changing.

This general approach is called
[[https://en.wikipedia.org/wiki/Newton%27s_method][Newton's method]].
It works well for many functions but especially well for square root.)

*** Switch**

A `switch` statement is a shorter way to write a sequence of `if`-`else` statements.

It runs the first case whose value is equal to the condition expression.

Go's switch is like the one in C, C++, Java, JavaScript, and PHP,

except that Go only runs the selected case, not all the cases that follow.

In effect, the `break` statement that is needed at the end of each case in those

languages is provided automatically in Go.

Another important difference is that Go's switch cases need not

be constants, and the values involved need not be integers.

*** Switch evaluation order**

Switch cases evaluate cases from top to bottom, stopping when a case succeeds.

(For example,

```
 switch i {

 case 0:

 case f():
}
```

does not call `f` if `i==0`.)

*** Switch with no condition**

Switch without a condition is the same as `switch`true`.

This construct can be a clean way to write long if-then-else chains.

*** Defer**

A defer statement defers the execution of a function until the surrounding function returns.

The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns.

*** Stacking defers**

Deferred function calls are pushed onto a stack. When a function returns, its

deferred calls are executed in last-in-first-out order.

To learn more about defer statements read this

[[/blog/defer-panic-and-recover][blog post]].

*** Congratulations!**

You finished this lesson!

You can go back to the list of [[/tour/list][modules]] to find what to learn next, or continue with the [[javascript:click('.next-page')][next lesson]].

More types: structs, slices, and maps.

Learn how to define types based on existing ones: this lesson covers structs, arrays, slices, and maps.

The Go Authors

https://golang.org

*** Pointers**

Go has pointers.

A pointer holds the memory address of a value.

The type `*T` is a pointer to a `T` value. Its zero value is `nil`.

```
 var p *int
```

The `&` operator generates a pointer to its operand.

```
 i := 42

 p = &i
```

The `*` operator denotes the pointer's underlying value.

```
 fmt.Println(*p) // read i through the pointer p
 *p = 21         // set i through the pointer p
```

This is known as "dereferencing" or "indirecting".

Unlike C, Go has no pointer arithmetic.

*** Structs**

A `struct` is a collection of fields.

*** Struct Fields**

Struct fields are accessed using a dot.

*** Pointers to structs**

Struct fields can be accessed through a struct pointer.

To access the field `X` of a struct when we have the struct pointer `p` we could

write `(*p).X`.

However, that notation is cumbersome, so the language permits us instead to

write just `p.X`, without the explicit dereference.

*** Struct Literals**

A struct literal denotes a newly allocated struct value by listing the values of its fields.

You can list just a subset of fields by using the `Name:` syntax. (And the order of named fields is irrelevant.)

The special prefix `&` returns a pointer to the struct value.

*** Arrays**

The type `[n]T` is an array of `n` values of type `T`.

The expression

```
 var a [10]int
```

declares a variable `a` as an array of ten integers.

An array's length is part of its type, so arrays cannot be resized.

This seems limiting, but don't worry;

Go provides a convenient way of working with arrays.

*** Slices**

An array has a fixed size.

A slice, on the other hand, is a dynamically-sized,

flexible view into the elements of an array.

In practice, slices are much more common than arrays.

The type `[]T` is a slice with elements of type `T`.

A slice is formed by specifying two indices, a low and

high bound, separated by a colon:

```
 a[low : high]
```

This selects a half-open range which includes the first

element, but excludes the last one.

The following expression creates a slice which includes

elements 1 through 3 of `a`:

```
 a[1:4]
```

*** Slices are like references to arrays**

A slice does not store any data,

it just describes a section of an underlying array.

Changing the elements of a slice modifies the

corresponding elements of its underlying array.

Other slices that share the same underlying array will see those changes.

*** Slice literals**

A slice literal is like an array literal without the length.

This is an array literal:

```
 [3]bool{true, true, false}
```

And this creates the same array as above,

then builds a slice that references it:

```
 []bool{true, true, false}
```

*** Slice defaults**

When slicing, you may omit the high or low bounds to use their defaults instead.

The default is zero for the low bound and the length of the slice for the high bound.

For the array

```
 var a [10]int
```

these slice expressions are equivalent:

```
 a[0:10]
 a[:10]
 a[0:]
 a[:]
```

*** Slice length and capacity**

A slice has both a _length_ and a _capacity_.

The length of a slice is the number of elements it contains.

The capacity of a slice is the number of elements in the underlying array,

counting from the first element in the slice.

The length and capacity of a slice `s` can be obtained using the expressions

`len(s)` and `cap(s)`.

You can extend a slice's length by re-slicing it,

provided it has sufficient capacity.

Try changing one of the slice operations in the example program to extend it

beyond its capacity and see what happens.

*** Nil slices**

The zero value of a slice is `nil`.

A nil slice has a length and capacity of 0

and has no underlying array.

*** Creating a slice with make**

Slices can be created with the built-in `make` function;

this is how you create dynamically-sized arrays.

The `make` function allocates a zeroed array

and returns a slice that refers to that array:

```
 a := make([]int, 5)  // len(a)=5
```

To specify a capacity, pass a third argument to `make`:

```
 b := make([]int, 0, 5) // len(b)=0, cap(b)=5

 b = b[:cap(b)] // len(b)=5, cap(b)=5
 b = b[1:]      // len(b)=4, cap(b)=4
```

*** Slices of slices**

Slices can contain any type, including other slices.

*** Appending to a slice**

It is common to append new elements to a slice, and so Go provides a built-in

`append` function. The [[/pkg/builtin/#append][documentation]]

of the built-in package describes `append`.

```
 func append(s []T, vs ...T) []T
```

The first parameter `s` of `append` is a slice of type `T`, and the rest are

`T` values to append to the slice.

The resulting value of `append` is a slice containing all the elements of the

original slice plus the provided values.

If the backing array of `s` is too small to fit all the given values a bigger

array will be allocated. The returned slice will point to the newly allocated

array.

(To learn more about slices, read the [[/blog/go-slices-usage-and-internals][Slices: usage and internals]] article.)

*** Range**

The `range` form of the `for` loop iterates over a slice or map.

When ranging over a slice, two values are returned for each iteration.

The first is the index, and the second is a copy of the element at that index.

*** Range continued**

You can skip the index or value by assigning to `_`.

```
    for i, _ := range pow
    for _, value := range pow
```

If you only want the index, you can omit the second variable.

```
    for i := range pow
```

*** Exercise: Slices**

Implement `Pic`. It should return a slice of length `dy`, each element of which is a slice of `dx` 8-bit unsigned integers. When you run the program, it will display your picture, interpreting the integers as grayscale (well, bluescale) values.

The choice of image is up to you. Interesting functions include `(x+y)/2`, `x*y`, and `x^y`.

(You need to use a loop to allocate each `[]uint8` inside the `[][]uint8`.)

(Use `uint8(intValue)` to convert between types.)

*** Maps**

A map maps keys to values.

The zero value of a map is `nil`.

A `nil` map has no keys, nor can keys be added.

The `make` function returns a map of the given type,

initialized and ready for use.

*** Map literals**

Map literals are like struct literals, but the keys are required.

*** Map literals continued**

If the top-level type is just a type name, you can omit it from the elements of the literal.

*** Mutating Maps**

Insert or update an element in map `m`:

```
 m[key] = elem
```

Retrieve an element:

```
 elem = m[key]
```

Delete an element:

```
 delete(m, key)
```

Test that a key is present with a two-value assignment:

```
 elem, ok = m[key]
```

If `key` is in `m`, `ok` is `true`. If not, `ok` is `false`.

If `key` is not in the map, then `elem` is the zero value for the map's element type.

***Note:* If `elem` or `ok` have not yet been declared you could use a short declaration form:**

```
 elem, ok := m[key]
```

*** Exercise: Maps**

Implement `WordCount`. It should return a map of the counts of each “word” in the string `s`. The `wc.Test` function runs a test suite against the provided function and prints success or failure.

You might find [[/pkg/strings/#Fields][strings.Fields]] helpful.

*** Function values**

Functions are values too. They can be passed around just like other values.

Function values may be used as function arguments and return values.

*** Function closures**

Go functions may be closures. A closure is a function value that references variables from outside its body. The function may access and assign to the referenced variables; in this sense the function is "bound" to the variables.

For example, the `adder` function returns a closure. Each closure is bound to its own `sum` variable.

*** Exercise: Fibonacci closure**

Let's have some fun with functions.

Implement a `fibonacci` function that returns a function (a closure) that

returns successive [[https://en.wikipedia.org/wiki/Fibonacci_number][fibonacci numbers]]
(0, 1, 1, 2, 3, 5, ...).

*** Congratulations!**

You finished this lesson!

You can go back to the list of [[/tour/list][modules]] to find what to learn next, or continue with the [[javascript:click('.next-page')][next lesson]].

Methods and interfaces

This lesson covers methods and interfaces, the constructs that define objects and their behavior.

The Go Authors

https://golang.org

*** Methods**

Go does not have classes.

However, you can define methods on types.

A method is a function with a special _receiver_ argument.

The receiver appears in its own argument list between the `func` keyword and

the method name.

In this example, the `Abs` method has a receiver of type `Vertex` named `v`.

*** Methods are functions**

Remember: a method is just a function with a receiver argument.

Here's `Abs` written as a regular function with no change in functionality.

*** Methods continued**

You can declare a method on non-struct types, too.

In this example we see a numeric type `MyFloat` with an `Abs` method.

You can only declare a method with a receiver whose type is defined in the same package as the method.
You cannot declare a method with a receiver whose type is defined in another package (which includes the built-in types such as `int`).

*** Pointer receivers**

You can declare methods with pointer receivers.

This means the receiver type has the literal syntax `*T` for some type `T`.

(Also, `T` cannot itself be a pointer such as `*int`.)

For example, the `Scale` method here is defined on `*Vertex`.

Methods with pointer receivers can modify the value to which the receiver points (as `Scale` does here).

Since methods often need to modify their receiver, pointer receivers are more common than value receivers.

Try removing the `*` from the declaration of the `Scale` function on line 16 and observe how the program's behavior changes.

With a value receiver, the `Scale` method operates on a copy of the original `Vertex` value.

(This is the same behavior as for any other function argument.)

The `Scale` method must have a pointer receiver to change the `Vertex` value declared in the `main` function.

*** Pointers and functions**

Here we see the `Abs` and `Scale` methods rewritten as functions.

Again, try removing the `*` from line 16.

Can you see why the behavior changes?

What else did you need to change for the example to compile?

(If you're not sure, continue to the next page.)

*** Methods and pointer indirection**

Comparing the previous two programs, you might notice that

functions with a pointer argument must take a pointer:

```
 var v Vertex
 ScaleFunc(v, 5)  // Compile error!
 ScaleFunc(&v, 5) // OK
```

while methods with pointer receivers take either a value or a pointer as the

receiver when they are called:

```
 var v Vertex
 v.Scale(5)  // OK
 p := &v
 p.Scale(10) // OK
```

For the statement `v.Scale(5)`, even though `v` is a value and not a pointer,

the method with the pointer receiver is called automatically.

That is, as a convenience, Go interprets the statement `v.Scale(5)` as

`(&v).Scale(5)` since the `Scale` method has a pointer receiver.

*** Methods and pointer indirection (2)**

The equivalent thing happens in the reverse direction.

Functions that take a value argument must take a value of that specific type:

```
 var v Vertex
 fmt.Println(AbsFunc(v))  // OK
 fmt.Println(AbsFunc(&v)) // Compile error!
```

while methods with value receivers take either a value or a pointer as the receiver when they are called:

```
 var v Vertex

 fmt.Println(v.Abs()) // OK

 p := &v

 fmt.Println(p.Abs()) // OK
```

In this case, the method call `p.Abs()` is interpreted as `(*p).Abs()`.

*** Choosing a value or pointer receiver**

There are two reasons to use a pointer receiver.

The first is so that the method can modify the value that its receiver points to.

The second is to avoid copying the value on each method call.

This can be more efficient if the receiver is a large struct, for example.

In this example, both `Scale` and `Abs` are methods with receiver type `*Vertex`,

even though the `Abs` method needn't modify its receiver.

In general, all methods on a given type should have either value or pointer

receivers, but not a mixture of both.

(We'll see why over the next few pages.)

*** Interfaces**

An _interface_type_ is defined as a set of method signatures.

A value of interface type can hold any value that implements those methods.

***Note:* There is an error in the example code on line 22.**

`Vertex` (the value type) doesn't implement `Abser` because

the `Abs` method is defined only on `*Vertex` (the pointer type).

*** Interfaces are implemented implicitly**

A type implements an interface by implementing its methods.

There is no explicit declaration of intent, no "implements" keyword.

Implicit interfaces decouple the definition of an interface from its implementation, which could then appear in any package without prearrangement.

*** Interface values**

Under the hood, interface values can be thought of as a tuple of a value and a concrete type:

 (value, type)

An interface value holds a value of a specific underlying concrete type.

Calling a method on an interface value executes the method of the same name on its underlying type.

*** Interface values with nil underlying values**

If the concrete value inside the interface itself is nil,

the method will be called with a nil receiver.

In some languages this would trigger a null pointer exception,

but in Go it is common to write methods that gracefully handle being called

with a nil receiver (as with the method `M` in this example.)

Note that an interface value that holds a nil concrete value is itself non-nil.

*** Nil interface values**

A nil interface value holds neither value nor concrete type.

Calling a method on a nil interface is a run-time error because there is no

type inside the interface tuple to indicate which _concrete_ method to call.

*** The empty interface**

The interface type that specifies zero methods is known as the _empty_interface_:

```
 interface{}
```

An empty interface may hold values of any type.

(Every type implements at least zero methods.)

Empty interfaces are used by code that handles values of unknown type.

For example, `fmt.Print` takes any number of arguments of type `interface{}`.

*** Type assertions**

A _type_assertion_ provides access to an interface value's underlying concrete value.

```
 t := i.(T)
```

This statement asserts that the interface value `i` holds the concrete type `T`

and assigns the underlying `T` value to the variable `t`.

If `i` does not hold a `T`, the statement will trigger a panic.

To _test_ whether an interface value holds a specific type,

a type assertion can return two values: the underlying value

and a boolean value that reports whether the assertion succeeded.

```
 t, ok := i.(T)
```

If `i` holds a `T`, then `t` will be the underlying value and `ok` will be true.

If not, `ok` will be false and `t` will be the zero value of type `T`,

and no panic occurs.

Note the similarity between this syntax and that of reading from a map.

*** Type switches**

A _type_switch_ is a construct that permits several type assertions in series.

A type switch is like a regular switch statement, but the cases in a type

switch specify types (not values), and those values are compared against the type of the value held by the given interface value.

```
switch v := i.(type) {
case T:
 // here v has type T
case S:
 // here v has type S
default:
 // no match; here v has the same type as i
}
```

The declaration in a type switch has the same syntax as a type assertion `i.(T)`, but the specific type `T` is replaced with the keyword `type`.

This switch statement tests whether the interface value `i` holds a value of type `T` or `S`.

In each of the `T` and `S` cases, the variable `v` will be of type `T` or `S` respectively and hold the value held by `i`.

In the default case (where there is no match), the variable `v` is of the same interface type and value as `i`.

*** Stringers**

One of the most ubiquitous interfaces is [[/pkg/fmt/#Stringer][`Stringer`]] defined by the [[/pkg/fmt/][`fmt`]] package.

```
 type Stringer interface {

  String() string

 }
```

A `Stringer` is a type that can describe itself as a string. The `fmt` package

(and many others) look for this interface to print values.

*** Exercise: Stringers**

Make the `IPAddr` type implement `fmt.Stringer` to print the address as

a dotted quad.

For instance, `IPAddr{1,`2,`3,`4}` should print as `"1.2.3.4"`.

*** Errors**

Go programs express error state with `error` values.

The `error` type is a built-in interface similar to `fmt.Stringer`:

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

(As with `fmt.Stringer`, the `fmt` package looks for the `error` interface when printing values.)

Functions often return an `error` value, and calling code should handle errors by testing whether the error equals `nil`.

```
 i, err := strconv.Atoi("42")
 if err != nil {
  fmt.Printf("couldn't convert number: %v\n", err)
  return
 }
 fmt.Println("Converted integer:", i)
```

A nil `error` denotes success; a non-nil `error` denotes failure.

*** Exercise: Errors**

Copy your `Sqrt` function from the [[/tour/flowcontrol/8][earlier exercise]] and modify it to return an `error` value.

`Sqrt` should return a non-nil error value when given a negative number, as it doesn't support complex numbers.

Create a new type

```
 type ErrNegativeSqrt float64
```

and make it an `error` by giving it a

```
 func (e ErrNegativeSqrt) Error() string
```

method such that `ErrNegativeSqrt(-2).Error()` returns `"cannot`Sqrt`negative`number:`-2"`.

***Note:* A call to `fmt.Sprint(e)` inside the `Error` method will send the program into an infinite loop. You can avoid this by converting `e` first: `fmt.Sprint(float64(e))`. Why?**

Change your `Sqrt` function to return an `ErrNegativeSqrt` value when given a negative number.

*** Readers**

The `io` package specifies the `io.Reader` interface,

which represents the read end of a stream of data.

The Go standard library contains
[[https://cs.opensource.google/search?q=Read%5C(%5Cw%2B%5Cs%5C%5B%5C%5Dbyte
implementations]] of this interface, including files, network connections, compressors, ciphers, and others.
The `io.Reader` interface has a `Read` method:

```
 func (T) Read(b []byte) (n int, err error)
```

`Read` populates the given byte slice with data and returns the number of bytes

populated and an error value. It returns an `io.EOF` error when the stream

ends.

The example code creates a

[[/pkg/strings/#Reader][`strings.Reader`]]

and consumes its output 8 bytes at a time.

*** Exercise: Readers**

Implement a `Reader` type that emits an infinite stream of the ASCII character

`'A'`.

*** Exercise: rot13Reader**

A common pattern is an [[/pkg/io/#Reader][io.Reader]] that wraps another `io.Reader`, modifying the stream in some way.

For example, the [[/pkg/compress/gzip/#NewReader][gzip.NewReader]] function takes an `io.Reader` (a stream of compressed data) and returns a `*gzip.Reader` that also implements `io.Reader` (a stream of the decompressed data).

Implement a `rot13Reader` that implements `io.Reader` and reads from an `io.Reader`, modifying the stream by applying the [[https://en.wikipedia.org/wiki/ROT13][rot13]] substitution cipher to all alphabetical characters.
The `rot13Reader` type is provided for you.

Make it an `io.Reader` by implementing its `Read` method.

*** Images**

[[/pkg/image/#Image][Package image]] defines the `Image` interface:

```
 package image

 type Image interface {
  ColorModel() color.Model
  Bounds() Rectangle
  At(x, y int) color.Color
 }
```

***Note*: the `Rectangle` return value of the `Bounds` method is actually an**

[[/pkg/image/#Rectangle][`image.Rectangle`]], as the

declaration is inside package `image`.

(See [[/pkg/image/#Image][the documentation]] for all the details.)

The `color.Color` and `color.Model` types are also interfaces, but we'll ignore that by using the predefined implementations `color.RGBA` and `color.RGBAModel`. These interfaces and types are specified by the [[/pkg/image/color/][image/color package]]

*** Exercise: Images**

Remember the [[/tour/moretypes/18][picture generator]] you wrote earlier? Let's write another one, but this time it will return an implementation of `image.Image` instead of a slice of data.

Define your own `Image` type, implement [[/pkg/image/#Image][the necessary methods]], and call `pic.ShowImage`.

`Bounds` should return a `image.Rectangle`, like `image.Rect(0,`0,`w,`h)`.

`ColorModel` should return `color.RGBAModel`.

`At` should return a color; the value `v` in the last picture generator corresponds to `color.RGBA{v,`v,`255,`255}` in this one.

*** Congratulations!**

You finished this lesson!

You can go back to the list of [[/tour/list][modules]] to find what to learn next, or continue with the [[javascript:click('.next-page')][next lesson]].

Generics

Go supports generic programming using type parameters. This lesson shows some examples for employing generics in your code.

The Go Authors

https://golang.org

*** Type parameters**

Go functions can be written to work on multiple types using type parameters. The

type parameters of a function appear between brackets, before the function's

arguments.

```
  func Index[T comparable](s []T, x T) int
```

This declaration means that `s` is a slice of any type `T` that fulfills the

built-in constraint `comparable`. `x` is also a value of the same type.

`comparable` is a useful constraint that makes it possible to use the `==` and

`!=` operators on values of the type. In this example, we use it to compare a

value to all slice elements until a match is found. This `Index` function works

for any type that supports comparison.

*** Generic types**

In addition to generic functions, Go also supports generic types. A type can

be parameterized with a type parameter, which could be useful for implementing

generic data structures.

This example demonstrates a simple type declaration for a singly-linked list

holding any type of value.

As an exercise, add some functionality to this list implementation.

*** Congratulations!**

You finished this lesson!

You can go back to the list of [[/tour/list][modules]] to find what to learn next, or
continue with the [[javascript:click('.next-page')][next lesson]].

Concurrency

Go provides concurrency constructions as part of the core language. This lesson presents them and gives some examples on how they can be used.

The Go Authors

https://golang.org

*** Goroutines**

A _goroutine_ is a lightweight thread managed by the Go runtime.

```
 go f(x, y, z)
```

starts a new goroutine running

```
 f(x, y, z)
```

The evaluation of `f`, `x`, `y`, and `z` happens in the current goroutine and the execution of `f` happens in the new goroutine.

Goroutines run in the same address space, so access to shared memory must be synchronized. The [[/pkg/sync/][`sync`]] package provides useful primitives, although you won't need them much in Go as there are other primitives. (See the next slide.)

*** Channels**

Channels are a typed conduit through which you can send and receive values with the channel operator, `<-`.

```
 ch <- v    // Send v to channel ch.

 v := <-ch  // Receive from ch, and

            // assign value to v.
```

(The data flows in the direction of the arrow.)

Like maps and slices, channels must be created before use:

```
 ch := make(chan int)
```

By default, sends and receives block until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

The example code sums the numbers in a slice, distributing the work between two goroutines.
Once both goroutines have completed their computation, it calculates the final result.

*** Buffered Channels**

Channels can be _buffered_. Provide the buffer length as the second argument to `make` to initialize a buffered channel:

```
 ch := make(chan int, 100)
```

Sends to a buffered channel block only when the buffer is full. Receives block when the buffer is empty.

Modify the example to overfill the buffer and see what happens.

*** Range and Close**

A sender can `close` a channel to indicate that no more values will be sent. Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression: after

```
 v, ok := <-ch
```

`ok` is `false` if there are no more values to receive and the channel is closed.

The loop `for`i`:=`range`c` receives values from the channel repeatedly until it is closed.

***Note:* Only the sender should close a channel, never the receiver. Sending on a closed channel will cause a panic.**

***Another*note:* Channels aren't like files; you don't usually need to close them. Closing is only necessary when the receiver must be told there are no more values coming, such as to terminate a `range` loop.**

*** Select**

The `select` statement lets a goroutine wait on multiple communication operations.

A `select` blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

*** Default Selection**

The `default` case in a `select` is run if no other case is ready.

Use a `default` case to try a send or receive without blocking:

```
 select {

 case i := <-c:

  // use i

 default:
  // receiving from c would block
 }
```

*** Exercise: Equivalent Binary Trees**

There can be many different binary trees with the same sequence of values stored in it. For example, here are two binary trees storing the sequence 1, 1, 2, 3, 5, 8, 13.

A function to check whether two binary trees store the same sequence is quite complex in most languages. We'll use Go's concurrency and channels to write a simple solution.

This example uses the `tree` package, which defines the type:

```
 type Tree struct {
  Left  *Tree
  Value int
  Right *Tree
 }
```

Continue description on [[javascript:click('.next-page')][next page]].

*** Exercise: Equivalent Binary Trees**

***1.* Implement the `Walk` function.**

***2.* Test the `Walk` function.**

The function `tree.New(k)` constructs a randomly-structured (but always sorted) binary tree holding the values `k`, `2k`, `3k`, ..., `10k`.

Create a new channel `ch` and kick off the walker:

```
 go Walk(tree.New(1), ch)
```

Then read and print 10 values from the channel. It should be the numbers 1, 2, 3, ..., 10.

***3.* Implement the `Same` function using `Walk` to determine whether `t1` and `t2` store the same values.**

***4.* Test the `Same` function.**

`Same(tree.New(1),`tree.New(1))` should return true, and
`Same(tree.New(1),`tree.New(2))` should return false.

The documentation for `Tree` can be found
[[https://godoc.org/golang.org/x/tour/tree#Tree][here]].

*** sync.Mutex**

We've seen how channels are great for communication among goroutines.

But what if we don't need communication? What if we just want to make sure only

one goroutine can access a variable at a time to avoid conflicts?

This concept is called _mutual_exclusion_, and the conventional name for the data structure that provides it is _mutex_.

Go's standard library provides mutual exclusion with

[[/pkg/sync/#Mutex][`sync.Mutex`]] and its two methods:

- `Lock`

- `Unlock`

We can define a block of code to be executed in mutual exclusion by surrounding it

with a call to `Lock` and `Unlock` as shown on the `Inc` method.

We can also use `defer` to ensure the mutex will be unlocked as in the `Value` method.

*** Exercise: Web Crawler**

In this exercise you'll use Go's concurrency features to parallelize a web crawler.

Modify the `Crawl` function to fetch URLs in parallel without fetching the same URL twice.

Hint: you can keep a cache of the URLs that have been fetched on a map, but maps alone are not
safe for concurrent use!

*** Where to Go from here...**

The

[[/doc/][Go Documentation]] is a great place to

start.

It contains references, tutorials, videos, and more.

To learn how to organize and work with Go code, read [[/doc/code][How to Write Go Code]].

If you need help with the standard library, see the [[/pkg/][package reference]]. For help with the language itself, you might be surprised to find the [[/ref/spec][Language Spec]] is quite readable.

To further explore Go's concurrency model, watch

[[https://www.youtube.com/watch?v=f6kdp27TYZs][Go Concurrency Patterns]]

([[/talks/2012/concurrency.slide][slides]])

and

[[https://www.youtube.com/watch?v=QDDwwePbDtw][Advanced Go Concurrency Patterns]]
([[/talks/2013/advconc.slide][slides]])

and read the

[[/doc/codewalk/sharemem/][Share Memory by Communicating]]

codewalk.

To get started writing web applications, watch

[[https://vimeo.com/53221558][A simple programming environment]]

([[/talks/2012/simple.slide][slides]])

and read the

[[/doc/articles/wiki/][Writing Web Applications]] tutorial.

The [[/doc/codewalk/functions/][First Class Functions in Go]] codewalk gives an interesting perspective on Go's function types.

The [[/blog/][Go Blog]] has a large archive of informative Go articles.

Visit [[/][the Go home page]] for more.