TOUR OF GO

Basics

Packages, variables, and functions

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`.

```
package main

import (
	"fmt"
	"math/rand"
)

func main() {
	fmt.Println("My favorite number is", rand.Intn(10))
}
```

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.

```
package main

import (
	"fmt"
	"math"
)

func main() {
	fmt.Printf("Now you have %g problems.\n", math.Sqrt(7))
}
```

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.

```
package main

import (
	"fmt"
	"math"
)

func main() {
	fmt.Println(math.pi)
}
```

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 article on Go's declaration syntax.)

```
package main

import "fmt"

func add(x int, y int) int {
	return x + y
}

func main() {
	fmt.Println(add(42, 13))
}
```

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`

```
package main

import "fmt"

func add(x, y int) int {
	return x + y
}

func main() {
	fmt.Println(add(42, 13))
}
```

Multiple results

A function can return any number of results.

The `swap` function returns two strings.

```
package main

import "fmt"

func swap(x, y string) (string, string) {
	return y, x
}

func main() {
	a, b := swap("hello", "world")
	fmt.Println(a, b)
}
```

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.

```
package main

import "fmt"

func split(sum int) (x, y int) {
	x = sum * 4 / 9
	y = sum - x
	return
}

func main() {
	fmt.Println(split(17))
}
```

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.

```
package main

import "fmt"

var c, python, java bool

func main() {
	var i int
	fmt.Println(i, c, python, java)
}
```

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.

```
package main

import "fmt"

var i, j int = 1, 2

func main() {
	var c, python, java = true, false, "no!"
```

```
        fmt.Println(i, j, c, python, java)
}
```

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.

```
package main

import "fmt"

func main() {
        var i, j int = 1, 2
        k := 3
        c, python, java := true, false, "no!"

        fmt.Println(i, j, k, c, python, java)
}
```

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.

```
package main

import (
	"fmt"
	"math/cmplx"
)

var (
	ToBe   bool       = false
	MaxInt uint64     = 1<<64 - 1
	z      complex128 = cmplx.Sqrt(-5 + 12i)
)

func main() {
	fmt.Printf("Type: %T Value: %v\n", ToBe, ToBe)
	fmt.Printf("Type: %T Value: %v\n", MaxInt, MaxInt)
	fmt.Printf("Type: %T Value: %v\n", z, z)
}
```

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.

```
package main

import "fmt"

func main() {
	var i int
	var f float64
	var b bool
	var s string
	fmt.Printf("%v %v %v %q\n", i, f, b, s)
}
```

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.

```
package main

import (
	"fmt"
	"math"
)

func main() {
	var x, y int = 3, 4
	var f float64 = math.Sqrt(float64(x*x + y*y))
	var z uint = uint(f)
	fmt.Println(x, y, z)
}
```

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.

```
package main

import "fmt"

func main() {
	v := 42 // change me!
	fmt.Printf("v is of type %T\n", v)
}
```

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.

```
package main

import "fmt"

const Pi = 3.14

func main() {
	const World = "世界"
	fmt.Println("Hello", World)
	fmt.Println("Happy", Pi, "Day")

	const Truth = true
	fmt.Println("Go rules?", Truth)
}
```

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.)

```
package main

import "fmt"

const (
	// Create a huge number by shifting a 1 bit left 100 places.
	// In other words, the binary number that is 1 followed by 100 zeroes.
	Big = 1 << 100
	// Shift it right again 99 places, so we end up with 1<<1, or 2.
	Small = Big >> 99
)

func needInt(x int) int { return x*10 + 1 }
func needFloat(x float64) float64 {
	return x * 0.1
}

func main() {
	fmt.Println(needInt(Small))
```

```
	fmt.Println(needFloat(Small))
	fmt.Println(needFloat(Big))
}
```

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

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.

```
package main

import "fmt"

func main() {
	sum := 0
	for i := 0; i < 10; i++ {
		sum += i
	}
	fmt.Println(sum)
}
```

For continued

The init and post statements are optional.

```
package main

import "fmt"

func main() {
	sum := 1
	for ; sum < 1000; {
		sum += sum
	}
```

```
	fmt.Println(sum)
}
```

For is Go's "while"

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

```
package main

import "fmt"

func main() {
	sum := 1
	for sum < 1000 {
		sum += sum
	}
	fmt.Println(sum)
}
```

Forever

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

```
package main

func main() {
	for {
	}
}
```

If

Go's `if` statements are like its `for` loops; the expression need not be surrounded by parentheses `( )` but the braces `{ }` are required.

```
package main

import (
	"fmt"
	"math"
)

func sqrt(x float64) string {
	if x < 0 {
		return sqrt(-x) + "i"
	}
	return fmt.Sprint(math.Sqrt(x))
}

func main() {
	fmt.Println(sqrt(2), sqrt(-4))
}
```

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.)

```
package main

import (
	"fmt"
	"math"
)

func pow(x, n, lim float64) float64 {
	if v := math.Pow(x, n); v < lim {
		return v
	}
	return lim
}

func main() {
	fmt.Println(
		pow(3, 2, 10),
		pow(3, 3, 20),
	)
}
```

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.)

```
package main

import (
	"fmt"
	"math"
)

func pow(x, n, lim float64) float64 {
	if v := math.Pow(x, n); v < lim {
		return v
	} else {
		fmt.Printf("%g >= %g\n", v, lim)
	}
	// can't use v here, though
	return lim
```

```
}

func main() {
	fmt.Println(
		pow(3, 2, 10),
		pow(3, 3, 20),
	)
}
```

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 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 Newton's method. It works well for many functions but especially well for square root.)

```
package main

import (
	"fmt"
```

```
)

func Sqrt(x float64) float64 {
}

func main() {
	fmt.Println(Sqrt(2))
}
```

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.

```
package main

import (
	"fmt"
	"runtime"
)

func main() {
	fmt.Print("Go runs on ")
	switch os := runtime.GOOS; os {
	case "darwin":
		fmt.Println("OS X.")
	case "linux":
		fmt.Println("Linux.")
	default:
		// freebsd, openbsd,
		// plan9, windows...
		fmt.Printf("%s.\n", os)
	}
}
```

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`.)

Note: Time in the Go playground always appears to start at 2009-11-10 23:00:00 UTC, a value whose significance is left as an exercise for the reader.

```
package main

import (
	"fmt"
	"time"
)

func main() {
	fmt.Println("When's Saturday?")
	today := time.Now().Weekday()
	switch time.Saturday {
	case today + 0:
		fmt.Println("Today.")
	case today + 1:
		fmt.Println("Tomorrow.")
	case today + 2:
		fmt.Println("In two days.")
	default:
		fmt.Println("Too far away.")
	}
}
```

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.

```
package main

import (
	"fmt"
	"time"
)

func main() {
	t := time.Now()
	switch {
	case t.Hour() < 12:
		fmt.Println("Good morning!")
	case t.Hour() < 17:
		fmt.Println("Good afternoon.")
	default:
		fmt.Println("Good evening.")
	}
}
```

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.

```
package main

import "fmt"

func main() {
	defer fmt.Println("world")

	fmt.Println("hello")
}
```

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 post.

```
package main

import "fmt"

func main() {
	fmt.Println("counting")

	for i := 0; i < 10; i++ {
		defer fmt.Println(i)
	}

	fmt.Println("done")
}
```

More types: structs, slices, and maps.

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.

```
package main

import "fmt"

func main() {
	i, j := 42, 2701

	p := &i         // point to i
	fmt.Println(*p) // read i through the pointer
	*p = 21         // set i through the pointer
	fmt.Println(i)  // see the new value of i

	p = &j         // point to j
	*p = *p / 37   // divide j through the pointer
	fmt.Println(j) // see the new value of j
}
```

Structs

A `struct` is a collection of fields.

```
package main

import "fmt"

type Vertex struct {
	X int
	Y int
}

func main() {
	fmt.Println(Vertex{1, 2})
}
```

Struct Fields

Struct fields are accessed using a dot.

```
package main

import "fmt"

type Vertex struct {
	X int
	Y int
}

func main() {
	v := Vertex{1, 2}
	v.X = 4
	fmt.Println(v.X)
}
```

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.

```
package main

import "fmt"

type Vertex struct {
	X int
	Y int
}

func main() {
	v := Vertex{1, 2}
	p := &v
	p.X = 1e9
	fmt.Println(v)
}
```

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.

```
package main

import "fmt"

type Vertex struct {
	X, Y int
}

var (
	v1 = Vertex{1, 2}  // has type Vertex
	v2 = Vertex{X: 1}  // Y:0 is implicit
	v3 = Vertex{}      // X:0 and Y:0
	p  = &Vertex{1, 2} // has type *Vertex
)

func main() {
	fmt.Println(v1, p, v2, v3)
}
```

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.

```
package main

import "fmt"

func main() {
	var a [2]string
	a[0] = "Hello"
	a[1] = "World"
	fmt.Println(a[0], a[1])
	fmt.Println(a)

	primes := [6]int{2, 3, 5, 7, 11, 13}
	fmt.Println(primes)
}
```

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]`

```
package main

import "fmt"

func main() {
	primes := [6]int{2, 3, 5, 7, 11, 13}

	var s []int = primes[1:4]
	fmt.Println(s)
}
```

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.

```
package main

import "fmt"

func main() {
	names := [4]string{
		"John",
		"Paul",
		"George",
		"Ringo",
	}
	fmt.Println(names)

	a := names[0:2]
```

```
	b := names[1:3]
	fmt.Println(a, b)

	b[0] = "XXX"
	fmt.Println(a, b)
	fmt.Println(names)
}
```

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}`

```
package main

import "fmt"

func main() {
	q := []int{2, 3, 5, 7, 11, 13}
	fmt.Println(q)

	r := []bool{true, false, true, true, false, true}
	fmt.Println(r)

	s := []struct {
		i int
		b bool
	}{
		{2, true},
		{3, false},
		{5, true},
		{7, true},
		{11, false},
		{13, true},
	}
	fmt.Println(s)
}
```

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[:]
```

```
package main

import "fmt"

func main() {
	s := []int{2, 3, 5, 7, 11, 13}

	s = s[1:4]
	fmt.Println(s)

	s = s[:2]
	fmt.Println(s)

	s = s[1:]
	fmt.Println(s)
}
```

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.

```
package main

import "fmt"

func main() {
	s := []int{2, 3, 5, 7, 11, 13}
	printSlice(s)

	// Slice the slice to give it zero length.
	s = s[:0]
	printSlice(s)
```

```
	// Extend its length.
	s = s[:4]
	printSlice(s)

	// Drop its first two values.
	s = s[2:]
	printSlice(s)
}

func printSlice(s []int) {
	fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
}
```

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.

```
package main

import "fmt"

func main() {
	var s []int
	fmt.Println(s, len(s), cap(s))
	if s == nil {
		fmt.Println("nil!")
	}
}
```

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
```

```
package main

import "fmt"
```

```
func main() {
	a := make([]int, 5)
	printSlice("a", a)

	b := make([]int, 0, 5)
	printSlice("b", b)

	c := b[:2]
	printSlice("c", c)

	d := c[2:5]
	printSlice("d", d)
}

func printSlice(s string, x []int) {
	fmt.Printf("%s len=%d cap=%d %v\n",
		s, len(x), cap(x), x)
}
```

Slices of slices

Slices can contain any type, including other slices.

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	// Create a tic-tac-toe board.
	board := [][]string{
		[]string{"_", "_", "_"},
		[]string{"_", "_", "_"},
		[]string{"_", "_", "_"},
	}

	// The players take turns.
	board[0][0] = "X"
	board[2][2] = "O"
	board[1][2] = "X"
	board[1][0] = "O"
	board[0][2] = "X"

	for i := 0; i < len(board); i++ {
		fmt.Printf("%s\n", strings.Join(board[i], " "))
	}
}
```

Appending to a slice

It is common to append new elements to a slice, and so Go provides a built-in `append` function. The 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 Slices: usage and internals article.)

```
package main

import "fmt"

func main() {
	var s []int
	printSlice(s)

	// append works on nil slices.
	s = append(s, 0)
	printSlice(s)

	// The slice grows as needed.
	s = append(s, 1)
	printSlice(s)

	// We can add more than one element at a time.
	s = append(s, 2, 3, 4)
	printSlice(s)
}

func printSlice(s []int) {
	fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
}
```

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.

```
package main
```

```
import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
	for i, v := range pow {
		fmt.Printf("2**%d = %d\n", i, v)
	}
}
```

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
```

```
package main

import "fmt"

func main() {
	pow := make([]int, 10)
	for i := range pow {
		pow[i] = 1 << uint(i) // == 2**i
	}
	for _, value := range pow {
		fmt.Printf("%d\n", value)
	}
}
```

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.)

```
package main
```

```
import "golang.org/x/tour/pic"

func Pic(dx, dy int) [][]uint8 {
}

func main() {
	pic.Show(Pic)
}
```

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.

```
package main

import "fmt"

type Vertex struct {
	Lat, Long float64
}

var m map[string]Vertex

func main() {
	m = make(map[string]Vertex)
	m["Bell Labs"] = Vertex{
		40.68433, -74.39967,
	}
	fmt.Println(m["Bell Labs"])
}
```

Map literals

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

```
package main

import "fmt"

type Vertex struct {
	Lat, Long float64
}

var m = map[string]Vertex{
	"Bell Labs": Vertex{
		40.68433, -74.39967,
	},
```

```
		"Google": Vertex{
			37.42202, -122.08408,
		},
	}

	func main() {
		fmt.Println(m)
	}
```

Map literals continued

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

```
	package main

	import "fmt"

	type Vertex struct {
		Lat, Long float64
	}

	var m = map[string]Vertex{
		"Bell Labs": {40.68433, -74.39967},
		"Google":    {37.42202, -122.08408},
	}

	func main() {
		fmt.Println(m)
	}
```

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]`

```
package main

import "fmt"

func main() {
	m := make(map[string]int)

	m["Answer"] = 42
	fmt.Println("The value:", m["Answer"])

	m["Answer"] = 48
	fmt.Println("The value:", m["Answer"])

	delete(m, "Answer")
	fmt.Println("The value:", m["Answer"])

	v, ok := m["Answer"]
	fmt.Println("The value:", v, "Present?", ok)
}
```

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 strings.Fields helpful.

```
package main

import (
	"golang.org/x/tour/wc"
)

func WordCount(s string) map[string]int {
	return map[string]int{"x": 1}
}

func main() {
	wc.Test(WordCount)
}
```

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.

```
package main

import (
	"fmt"
	"math"
)

func compute(fn func(float64, float64) float64) float64 {
	return fn(3, 4)
}

func main() {
	hypot := func(x, y float64) float64 {
		return math.Sqrt(x*x + y*y)
	}
	fmt.Println(hypot(5, 12))

	fmt.Println(compute(hypot))
	fmt.Println(compute(math.Pow))
}
```

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.

```
package main

import "fmt"

func adder() func(int) int {
	sum := 0
	return func(x int) int {
		sum += x
		return sum
	}
}

func main() {
	pos, neg := adder(), adder()
	for i := 0; i < 10; i++ {
		fmt.Println(
			pos(i),
			neg(-2*i),
		)
	}
}
```

Exercise: Fibonacci closure

Let's have some fun with functions.

Implement a `fibonacci` function that returns a function (a closure) that returns successive fibonacci numbers (0, 1, 1, 2, 3, 5, ...).

```
package main

import "fmt"

// fibonacci is a function that returns
// a function that returns an int.
func fibonacci() func() int {
}

func main() {
	f := fibonacci()
	for i := 0; i < 10; i++ {
		fmt.Println(f())
	}
}
```

Methods and interfaces

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`.

```
package main

import (
	"fmt"
	"math"
)

type Vertex struct {
	X, Y float64
}

func (v Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

```
func main() {
	v := Vertex{3, 4}
	fmt.Println(v.Abs())
}
```

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.

```
package main

import (
	"fmt"
	"math"
)

type Vertex struct {
	X, Y float64
}

func Abs(v Vertex) float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
	v := Vertex{3, 4}
	fmt.Println(Abs(v))
}
```

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`).

```
package main

import (
	"fmt"
	"math"
)

type MyFloat float64

func (f MyFloat) Abs() float64 {
```

```
	if f < 0 {
		return float64(-f)
	}
	return float64(f)
}

func main() {
	f := MyFloat(-math.Sqrt2)
	fmt.Println(f.Abs())
}
```

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.

```
package main

import (
	"fmt"
	"math"
)

type Vertex struct {
	X, Y float64
}

func (v Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func (v *Vertex) Scale(f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}

func main() {
```

```
	v := Vertex{3, 4}
	v.Scale(10)
	fmt.Println(v.Abs())
}
```

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.)

```
package main

import (
	"fmt"
	"math"
)

type Vertex struct {
	X, Y float64
}

func Abs(v Vertex) float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func Scale(v *Vertex, f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}

func main() {
	v := Vertex{3, 4}
	Scale(&v, 10)
	fmt.Println(Abs(v))
}
```

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.

```
package main

import "fmt"

type Vertex struct {
	X, Y float64
}

func (v *Vertex) Scale(f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}

func ScaleFunc(v *Vertex, f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}

func main() {
	v := Vertex{3, 4}
	v.Scale(2)
	ScaleFunc(&v, 10)

	p := &Vertex{4, 3}
	p.Scale(3)
	ScaleFunc(p, 8)

	fmt.Println(v, p)
}
```

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()`.

```
package main

import (
	"fmt"
	"math"
)

type Vertex struct {
	X, Y float64
}

func (v Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func AbsFunc(v Vertex) float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
	v := Vertex{3, 4}
	fmt.Println(v.Abs())
	fmt.Println(AbsFunc(v))

	p := &Vertex{4, 3}
	fmt.Println(p.Abs())
	fmt.Println(AbsFunc(*p))
}
```

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.)

```
package main
```

```
import (
	"fmt"
	"math"
)

type Vertex struct {
	X, Y float64
}

func (v *Vertex) Scale(f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}

func (v *Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
	v := &Vertex{3, 4}
	fmt.Printf("Before scaling: %+v, Abs: %v\n", v, v.Abs())
	v.Scale(5)
	fmt.Printf("After scaling: %+v, Abs: %v\n", v, v.Abs())
}
```

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).

```
package main

import (
	"fmt"
	"math"
)

type Abser interface {
	Abs() float64
}

func main() {
	var a Abser
	f := MyFloat(-math.Sqrt2)
	v := Vertex{3, 4}

	a = f  // a MyFloat implements Abser
```

```
	a = &v // a *Vertex implements Abser

	// In the following line, v is a Vertex (not *Vertex)
	// and does NOT implement Abser.
	a = v

	fmt.Println(a.Abs())
}

type MyFloat float64

func (f MyFloat) Abs() float64 {
	if f < 0 {
		return float64(-f)
	}
	return float64(f)
}

type Vertex struct {
	X, Y float64
}

func (v *Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

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.

```
package main

import "fmt"

type I interface {
	M()
}

type T struct {
	S string
}

// This method means type T implements the interface I,
// but we don't need to explicitly declare that it does so.
func (t T) M() {
	fmt.Println(t.S)
}
```

```
func main() {
	var i I = T{"hello"}
	i.M()
}
```

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.

```
package main

import (
	"fmt"
	"math"
)

type I interface {
	M()
}

type T struct {
	S string
}

func (t *T) M() {
	fmt.Println(t.S)
}

type F float64

func (f F) M() {
	fmt.Println(f)
}

func main() {
	var i I

	i = &T{"Hello"}
	describe(i)
	i.M()

	i = F(math.Pi)
	describe(i)
	i.M()
}
```

```
func describe(i I) {
	fmt.Printf("(%v, %T)\n", i, i)
}
```

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.

```
package main

import "fmt"

type I interface {
	M()
}

type T struct {
	S string
}

func (t *T) M() {
	if t == nil {
		fmt.Println("<nil>")
		return
	}
	fmt.Println(t.S)
}

func main() {
	var i I

	var t *T
	i = t
	describe(i)
	i.M()

	i = &T{"hello"}
	describe(i)
	i.M()
}

func describe(i I) {
	fmt.Printf("(%v, %T)\n", i, i)
}
```

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.

```
package main

import "fmt"

type I interface {
	M()
}

func main() {
	var i I
	describe(i)
	i.M()
}

func describe(i I) {
	fmt.Printf("(%v, %T)\n", i, i)
}
```

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{}`.

```
package main

import "fmt"

func main() {
	var i interface{}
	describe(i)

	i = 42
	describe(i)

	i = "hello"
```

```
	describe(i)
}

func describe(i interface{}) {
	fmt.Printf("(%v, %T)\n", i, i)
}
```

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.

```
package main

import "fmt"

func main() {
	var i interface{} = "hello"

	s := i.(string)
	fmt.Println(s)

	s, ok := i.(string)
	fmt.Println(s, ok)

	f, ok := i.(float64)
	fmt.Println(f, ok)

	f = i.(float64) // panic
	fmt.Println(f)
}
```

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`.

```
package main

import "fmt"

func do(i interface{}) {
	switch v := i.(type) {
	case int:
		fmt.Printf("Twice %v is %v\n", v, v*2)
	case string:
		fmt.Printf("%q is %v bytes long\n", v, len(v))
	default:
		fmt.Printf("I don't know about type %T!\n", v)
	}
}

func main() {
	do(21)
	do("hello")
	do(true)
}
```

Stringers

One of the most ubiquitous interfaces is `Stringer` defined by the `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.

```
package main

import "fmt"

type Person struct {
	Name string
	Age  int
}

func (p Person) String() string {
	return fmt.Sprintf("%v (%v years)", p.Name, p.Age)
}

func main() {
	a := Person{"Arthur Dent", 42}
	z := Person{"Zaphod Beeblebrox", 9001}
	fmt.Println(a, z)
}
```

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"`.

```
package main

import "fmt"

type IPAddr [4]byte

// TODO: Add a "String() string" method to IPAddr.

func main() {
	hosts := map[string]IPAddr{
		"loopback":  {127, 0, 0, 1},
		"googleDNS": {8, 8, 8, 8},
	}
	for name, ip := range hosts {
		fmt.Printf("%v: %v\n", name, ip)
```

```
        }
}
```

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.

```
package main

import (
	"fmt"
	"time"
)

type MyError struct {
	When time.Time
	What string
}

func (e *MyError) Error() string {
	return fmt.Sprintf("at %v, %s",
		e.When, e.What)
}

func run() error {
	return &MyError{
		time.Now(),
		"it didn't work",
	}
}
```

```
func main() {
	if err := run(); err != nil {
		fmt.Println(err)
	}
}
```

Exercise: Errors

Copy your `Sqrt` function from the 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.

```
package main

import (
	"fmt"
)

func Sqrt(x float64) (float64, error) {
	return 0, nil
}

func main() {
	fmt.Println(Sqrt(2))
	fmt.Println(Sqrt(-2))
}
```

Readers

The `io` package specifies the `io.Reader` interface, which represents the read end of a stream of data.

The Go standard library contains many 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 `strings.Reader` and consumes its output 8 bytes at a time.

```
package main

import (
	"fmt"
	"io"
	"strings"
)

func main() {
	r := strings.NewReader("Hello, Reader!")

	b := make([]byte, 8)
	for {
		n, err := r.Read(b)
		fmt.Printf("n = %v err = %v b = %v\n", n, err, b)
		fmt.Printf("b[:n] = %q\n", b[:n])
		if err == io.EOF {
			break
		}
	}
}
```

Exercise: Readers

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

```
package main

import "golang.org/x/tour/reader"

type MyReader struct{}

// TODO: Add a Read([]byte) (int, error) method to MyReader.
```

```
func main() {
	reader.Validate(MyReader{})
}
```

Exercise: rot13Reader

A common pattern is an io.Reader that wraps another `io.Reader`, modifying the stream in some way.

For example, the 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 rot13 substitution cipher to all alphabetical characters.

The `rot13Reader` type is provided for you. Make it an `io.Reader` by implementing its `Read` method.

```
package main

import (
	"io"
	"os"
	"strings"
)

type rot13Reader struct {
	r io.Reader
}

func main() {
	s := strings.NewReader("Lbh penpxrq gur pbqr!")
	r := rot13Reader{s}
	io.Copy(os.Stdout, &r)
}
```

Images

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 `image.Rectangle`, as the declaration is inside package `image`.

(See 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 image/color package

```
package main

import (
	"fmt"
	"image"
)

func main() {
	m := image.NewRGBA(image.Rect(0, 0, 100, 100))
	fmt.Println(m.Bounds())
	fmt.Println(m.At(0, 0).RGBA())
}
```

Exercise: Images

Remember the 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 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.

```
package main

import "golang.org/x/tour/pic"

type Image struct{}

func main() {
	m := Image{}
	pic.ShowImage(m)
}
```

Generics

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.

```
package main

import "fmt"

// Index returns the index of x in s, or -1 if not found.
func Index[T comparable](s []T, x T) int {
	for i, v := range s {
		// v and x are type T, which has the comparable
		// constraint, so we can use == here.
		if v == x {
			return i
		}
	}
	return -1
}

func main() {
	// Index works on a slice of ints
	si := []int{10, 20, 15, -10}
	fmt.Println(Index(si, 15))

	// Index also works on a slice of strings
	ss := []string{"foo", "bar", "baz"}
	fmt.Println(Index(ss, "hello"))
}
```

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.

```go
package main

// List represents a singly-linked list that holds
// values of any type.
type List[T any] struct {
	next *List[T]
	val  T
}

func main() {
}
```

Concurrency

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 `sync` package provides useful primitives, although you won't need them much in Go as there are other primitives. (See the next slide.)

```go
package main

import (
	"fmt"
	"time"
)

func say(s string) {
	for i := 0; i < 5; i++ {
		time.Sleep(100 * time.Millisecond)
		fmt.Println(s)
	}
}

func main() {
	go say("world")
	say("hello")
}
```

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.

```
package main

import "fmt"

func sum(s []int, c chan int) {
	sum := 0
	for _, v := range s {
		sum += v
	}
	c <- sum // send sum to c
}

func main() {
	s := []int{7, 2, 8, -9, 4, 0}

	c := make(chan int)
	go sum(s[:len(s)/2], c)
	go sum(s[len(s)/2:], c)
	x, y := <-c, <-c // receive from c

	fmt.Println(x, y, x+y)
}
```

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.

```
package main

import "fmt"

func main() {
	ch := make(chan int, 2)
	ch <- 1
	ch <- 2
	fmt.Println(<-ch)
	fmt.Println(<-ch)
}
```

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.

```
package main

import (
	"fmt"
)

func fibonacci(n int, c chan int) {
	x, y := 0, 1
	for i := 0; i < n; i++ {
		c <- x
		x, y = y, x+y
	}
	close(c)
```

```
}

func main() {
	c := make(chan int, 10)
	go fibonacci(cap(c), c)
	for i := range c {
		fmt.Println(i)
	}
}
```

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.

```
package main

import "fmt"

func fibonacci(c, quit chan int) {
	x, y := 0, 1
	for {
		select {
		case c <- x:
			x, y = y, x+y
		case <-quit:
			fmt.Println("quit")
			return
		}
	}
}

func main() {
	c := make(chan int)
	quit := make(chan int)
	go func() {
		for i := 0; i < 10; i++ {
			fmt.Println(<-c)
		}
		quit <- 0
	}()
	fibonacci(c, quit)
}
```

Default Selection

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

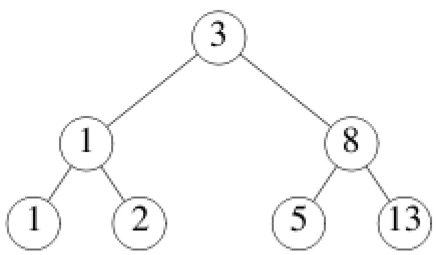
Use a `default` case to try a send or receive without blocking:

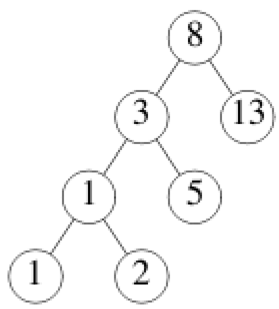
```
select {
case i := <-c:
    // use i
default:
    // receiving from c would block
}
```

```
package main

import (
	"fmt"
	"time"
)

func main() {
	tick := time.Tick(100 * time.Millisecond)
	boom := time.After(500 * time.Millisecond)
	for {
		select {
		case <-tick:
			fmt.Println("tick.")
		case <-boom:
			fmt.Println("BOOM!")
			return
		default:
			fmt.Println("    .")
			time.Sleep(50 * time.Millisecond)
		}
	}
}
```

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 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 here.

```
package main

import "golang.org/x/tour/tree"

// Walk walks the tree t sending all values
// from the tree to the channel ch.
func Walk(t *tree.Tree, ch chan int)

// Same determines whether the trees
// t1 and t2 contain the same values.
func Same(t1, t2 *tree.Tree) bool

func main() {
}
```

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 `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.

```
package main

import (
	"fmt"
	"sync"
	"time"
)

// SafeCounter is safe to use concurrently.
type SafeCounter struct {
	mu sync.Mutex
	v  map[string]int
}

// Inc increments the counter for the given key.
func (c *SafeCounter) Inc(key string) {
	c.mu.Lock()
	// Lock so only one goroutine at a time can access the map c.v.
	c.v[key]++
	c.mu.Unlock()
}

// Value returns the current value of the counter for the given key.
func (c *SafeCounter) Value(key string) int {
	c.mu.Lock()
	// Lock so only one goroutine at a time can access the map c.v.
	defer c.mu.Unlock()
	return c.v[key]
}

func main() {
	c := SafeCounter{v: make(map[string]int)}
```

```
	for i := 0; i < 1000; i++ {
		go c.Inc("somekey")
	}
package main

import (
	"fmt"
	"sync"
	"time"
)

// SafeCounter is safe to use concurrently.
type SafeCounter struct {
	mu sync.Mutex
	v  map[string]int
}

// Inc increments the counter for the given key.
func (c *SafeCounter) Inc(key string) {
	c.mu.Lock()
	// Lock so only one goroutine at a time can access the map c.v.
	c.v[key]++
	c.mu.Unlock()
}

// Value returns the current value of the counter for the given key.
func (c *SafeCounter) Value(key string) int {
	c.mu.Lock()
	// Lock so only one goroutine at a time can access the map c.v.
	defer c.mu.Unlock()
	return c.v[key]
}

func main() {
	c := SafeCounter{v: make(map[string]int)}
	for i := 0; i < 1000; i++ {
		go c.Inc("somekey")
	}

	time.Sleep(time.Second)
	fmt.Println(c.Value("somekey"))
}
```

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!

```go
package main

import (
	"fmt"
)

type Fetcher interface {
	// Fetch returns the body of URL and
	// a slice of URLs found on that page.
	Fetch(url string) (body string, urls []string, err error)
}

// Crawl uses fetcher to recursively crawl
// pages starting with url, to a maximum of depth.
func Crawl(url string, depth int, fetcher Fetcher) {
	// TODO: Fetch URLs in parallel.
	// TODO: Don't fetch the same URL twice.
	// This implementation doesn't do either:
	if depth <= 0 {
		return
	}
	body, urls, err := fetcher.Fetch(url)
	if err != nil {
		fmt.Println(err)
		return
	}
	fmt.Printf("found: %s %q\n", url, body)
	for _, u := range urls {
		Crawl(u, depth-1, fetcher)
	}
	return
}

func main() {
	Crawl("https://golang.org/", 4, fetcher)
}

// fakeFetcher is Fetcher that returns canned results.
type fakeFetcher map[string]*fakeResult

type fakeResult struct {
	body string
	urls []string
}

func (f fakeFetcher) Fetch(url string) (string, []string, error) {
	if res, ok := f[url]; ok {
		return res.body, res.urls, nil
	}
	return "", nil, fmt.Errorf("not found: %s", url)
}

// fetcher is a populated fakeFetcher.
```

```
var fetcher = fakeFetcher{
	"https://golang.org/": &fakeResult{
		"The Go Programming Language",
		[]string{
			"https://golang.org/pkg/",
			"https://golang.org/cmd/",
		},
	},
	"https://golang.org/pkg/": &fakeResult{
		"Packages",
		[]string{
			"https://golang.org/",
			"https://golang.org/cmd/",
			"https://golang.org/pkg/fmt/",
			"https://golang.org/pkg/os/",
		},
	},
	"https://golang.org/pkg/fmt/": &fakeResult{
		"Package fmt",
		[]string{
			"https://golang.org/",
			"https://golang.org/pkg/",
		},
	},
	"https://golang.org/pkg/os/": &fakeResult{
		"Package os",
		[]string{
			"https://golang.org/",
			"https://golang.org/pkg/",
		},
	},
}
```

Where to Go from here...

You can get started by installing Go.

Once you have Go installed, the Go Documentation is a great place to continue. It contains references, tutorials, videos, and more.

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

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

To further explore Go's concurrency model, watch Go Concurrency Patterns (slides) and Advanced Go Concurrency Patterns (slides) and read the Share Memory by Communicating codewalk.

To get started writing web applications, watch A simple programming environment (slides) and read the Writing Web Applications tutorial.

The First Class Functions in Go codewalk gives an interesting perspective on Go's function types.

The Go Blog has a large archive of informative Go articles.

Visit the Go home page for more.