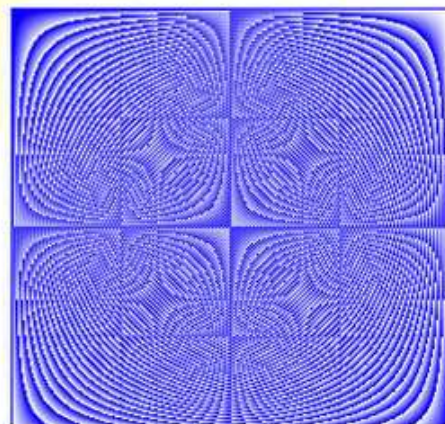


```
1 package main
2
3 import "code.google.com/p/go-tour/pic"
4
5 func Pic(dx, dy int) [][]uint8 {
6     slice := make([][]uint8, dy)
7     for i:=range slice {
8         slice[i] = make([]uint8, dx)
9         for j:=range slice[i] {
10             slice[i][j] = uint8(i*j)
11         }
12     }
13     return slice
14 }
15
16 func main() {
17     pic.Show(Pic)
18 }
```



InfoWorld

 [See larger image](#)

2 of 15

Slices

The Go language extends the idea of arrays with slices. A slice points to an array of values and includes a length. `[]T` is a slice with elements of type `T`. In the pictured exercise, we use slices of slices of unsigned bytes to hold the pixels of an image we generate. With `package main`, programs start running. The `import` statement is an extended version of C and C++'s `include` statement; here we are getting the `pic` file from a Mercurial repository. The `:=` syntax declares and initializes a variable, and the compiler infers a type whenever it can. Also, `make` is used to create slices and some other types. A `for..range` loop is the equivalent of C#'s `for..in` loop.

Need a concise, simple, safe, and fast compiled language with wonderful concurrency f

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     m := make(map[string]int)
7
8     m["Answer"] = 42
9     fmt.Println("The value:", m["Answer"])
10
11     m["Answer"] = 48
12     fmt.Println("The value:", m["Answer"])
13
14     delete(m, "Answer")
15     fmt.Println("The value:", m["Answer"])
16
17     v, ok := m["Answer"]
18     fmt.Println("The value:", v, "Present?", ok)
19 }
```

InfoWorld

 [See larger image](#)

3 of 15

Maps

The Go `map` statement maps keys to values. As with `slice`, you create a `map` with `make`, not `new`. In the example above, we are mapping string keys to integer values. Here we demonstrate inserting, updating, deleting, and testing for `map` elements.

The pictured program prints:

The value: 42

The value: 48 The value: 0

The value: 0

Present? false

```

1 package main
2
3 import (
4     "fmt"
5     "math"
6 )
7
8 type Vertex struct {
9     X, Y float64
10 }
11
12 func (v *Vertex) Abs() float64 {
13     return math.Sqrt(v.X*v.X + v.Y*v.Y)
14 }
15
16 func main() {
17     v := &Vertex{3, 4}
18     fmt.Println(v.Abs())
19 }

```

InfoWorld

 [See larger image](#)

4 of 15

Structs and methods

The Go language lacks classes but has a `struct`, which is a sequence of named elements, called fields, each with a name and a type. A `method` is a function with a receiver. A method declaration binds an identifier (the method name) to a method and associates the method with the receiver's base type. In this example, we declare a `Vertex struct` to contain two floating point fields, `X` and `Y`, and a method `Abs`. Fields that begin with uppercase letters are public; fields that begin with lowercase letters are private. Fields and methods are addressable through the dot notation; `*` and `&` signify pointers, as in C. This program prints 5.

```

8 type Abser interface {
9     Abs() float64
10 }
11
12 func main() {
13     var a Abser
14     f := MyFloat(-math.Sqrt2)
15     v := Vertex{3, 4}
16
17     a = f // a MyFloat implements Abser
18     a = &v // a *Vertex implements Abser
19
20     // In the following line, v is a Vertex (not *Vertex)
21     // and does NOT implement Abser.
22     a = v
23
24     fmt.Println(a.Abs())
25 }
26
27 type MyFloat float64
28
29 func (f MyFloat) Abs() float64 {
30     if f < 0 {
31         return float64(-f)
32     }
33     return float64(f)
34 }
35
36 type Vertex struct {
37     X, Y float64
38 }

```

InfoWorld

[See larger image](#)

5 of 15

Interfaces

An interface type is defined by a set of methods. A value of interface type can hold any value that implements those methods. In this example, we define an interface `Abser` and a variable `a` of type `Abser`. Note that the assignments in lines 17 and 18 work, but the assignment in line 22 does not even compile. The `Abs` method of `Vertex`, which we saw in the previous slide, has a pointer to `Vertex` type for its receiver, so a `*Vertex` implements `Abser`, but a `Vertex` does not.


```

1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func say(s string) {
9     for i := 0; i < 5; i++ {
10         time.Sleep(100 * time.Millisecond)
11         fmt.Println(s)
12     }
13 }
14
15 func main() {
16     go say("world")
17     say("hello")
18 }

```

InfoWorld

 See larger image

7 of 15

Goroutines

Goroutines are, to a rough approximation, extremely lightweight threads, in the spirit of Tony Hoare's Communicating Sequential Processes. Line 16 in the sample above calls the `say` function asynchronously, while line 17 calls the `say` function synchronously. Goroutines, channels, and `select` statements form the core of Go's highly scalable concurrency, one of the strongest selling points of the language. The language also has conventional synchronization objects, but they are rarely needed. The program above outputs:

```

hello
world
hello
world
hello
world
hello

```

```
1 package main
2
3 import (
4     "fmt"
5     "runtime"
6 )
7
8 func main() {
9     fmt.Print("Go runs on ")
10    switch os := runtime.GOOS; os {
11        case "darwin":
12            fmt.Println("OS X.")
13        case "linux":
14            fmt.Println("Linux.")
15        default:
16            // freebsd, openbsd,
17            // plan9, windows...
18            fmt.Printf("%s.", os)
19    }
20 }
```

InfoWorld

 See larger image

6 of 15

Switch

The `switch` statement in Go is similar to the `switch` statement in other C-like languages, except that the `case` statements can be types or expressions in addition to simple values, and the cases automatically break unless they end with `fallthrough` statements. The cases are evaluated in the order they are defined.

```

1 package main
2
3 import "fmt"
4
5 func sum(a []int, c chan int) {
6     sum := 0
7     for _, v := range a {
8         sum += v
9     }
10    c <- sum // send sum to c
11 }
12
13 func main() {
14     a := []int{7, 2, 8, -9, 4, 0}
15
16     c := make(chan int)
17     go sum(a[:len(a)/2], c)
18     go sum(a[len(a)/2:], c)
19     x, y := <-c, <-c // receive from c
20
21     fmt.Println(x, y, x+y)
22 }

```

InfoWorld

[See larger image](#)

8 of 15

Channels

Channels in Go provide a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type. The value of an uninitialized channel is nil. In line 16, we create a bidirectional channel of integers. We could also make unidirectional sending `<-c` and receiving `c<-` channels. In lines 17 and 18, we call `sum` asynchronously with slices of the first and second half of `a`. In line 19, the integer variables `x` and `y` receive the two sums from the channel. In line 7, the underscore `_`, the blank identifier, means to ignore the first result value from the `for..range` loop, which is the index. The program output is

17 -5 12.

```

1 package main
2
3 import (
4     "fmt"
5 )
6
7 func fibonacci(n int, c chan int) {
8     x, y := 0, 1
9     for i := 0; i < n; i++ {
10         c <- x
11         x, y = y, x+y
12     }
13     close(c)
14 }
15
16 func main() {
17     c := make(chan int, 10)
18     go fibonacci(cap(c), c)
19     for i := range c {
20         fmt.Println(i)
21     }
22 }

```

InfoWorld

 See larger image

9 of 15

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. A loop `for i := range c` receives values from the channel repeatedly until it is closed. The `cap` of the channel is the capacity, which is the size of the buffer in the channel, set as the optional second argument when you make a channel, as in line 17. Note the compact form of the assignment statements in the `fibonacci` function. The program output is the first 10 values of the Fibonacci series, 0 through 34.


```

1 package main
2
3 import "fmt"
4
5 func fibonacci(c, quit chan int) {
6     x, y := 0, 1
7     for {
8         select {
9             case c <- x:
10                x, y = y, x+y
11             case <-quit:
12                fmt.Println("quit")
13                return
14            }
15        }
16    }
17
18 func main() {
19     c := make(chan int)
20     quit := make(chan int)
21     go func() {
22         for i := 0; i < 10; i++ {
23             fmt.Println(<-c)
24         }
25         quit <- 0
26     }()
27     fibonacci(c, quit)
28 }

```

InfoWorld

 [See larger image](#)

10 of 15

Select

A `select` statement chooses which of a set of possible `send` or `receive` operations will proceed. It looks similar to a `switch` statement but with all the cases referring to 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.

Here the `main` function calls the `fibonacci` function with two unbuffered channels, one for results and one for a `quit` signal. The `fibonacci` function uses a `select` statement to wait on both channels. The anonymous, asynchronous `go` function that starts at line 21 waits to receive values at line 23, then prints them. After 10 values, it sets the `quit` channel, so the `fibonacci` function knows to stop.

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            select {
                case s := <-input1: c <- s
                case s := <-input2: c <- s
            }
        }
    }()
    return c
}
```


InfoWorld

 [See larger image](#)

11 of 15

Concurrency patterns, example 1

In this example we are using `select` to create a fan-in goroutine that combines two input channels of string, `input1` and `input2`, into one unbuffered output channel, `c`. The `select` statement allows `fanIn` to listen to both input channels simultaneously and relay whichever is ready to the output channel. It doesn't matter that both cases are using the same temporary variable name to hold the string from its respective input channel. The example is from [Rob Pike's 2012 talk on Concurrency Patterns in Go](#).



```
func First(query string, replicas ...Search) Result {  
    c := make(chan Result)  
    searchReplica := func(i int) { c <- replicas[i]  
(query) }  
    for i := range replicas {  
        go searchReplica(i)  
    }  
    return <-c  
}
```

InfoWorld

 [See larger image](#)

12 of 15

Concurrency patterns, example 2

This sample implements a parallel search of the Internet, sort of like what Google actually does. To begin with, `replicas ...Search` is a variadic parameter to the function; both `Search` and `Result` are types defined elsewhere.

The caller passes N search server functions to the `First` function, which creates a channel `c` for results and defines a function to query the *i*th server and saves it in `searchReplica`. Then `First` calls `searchReplica` asynchronously for all N servers, always returning the answer on channel `c`, and returns the first result to come back from the N servers. The example is from [Rob Pike's 2012 talk on Concurrency Patterns in Go](#).