Go Programming Language

Interfaces

Department of Computer Science and Engineering

Go Programming Language

Interfaces

- Interface types express generalizations or abstractions about the behaviors of other types.
- Interfaces let us write functions that are more flexible and adaptable because they are not tied to the details of one particular implementation
- Go’s interfaces are satisfied **implicitly**. In other words, there’s no need to declare all the interfaces that a given concrete type satisfies; simply possessing the necessary methods is enough.
- This design lets you create new interfaces that are satisfied by existing concrete types without changing the existing types, which is particularly useful for types defined in packages that you don’t control.

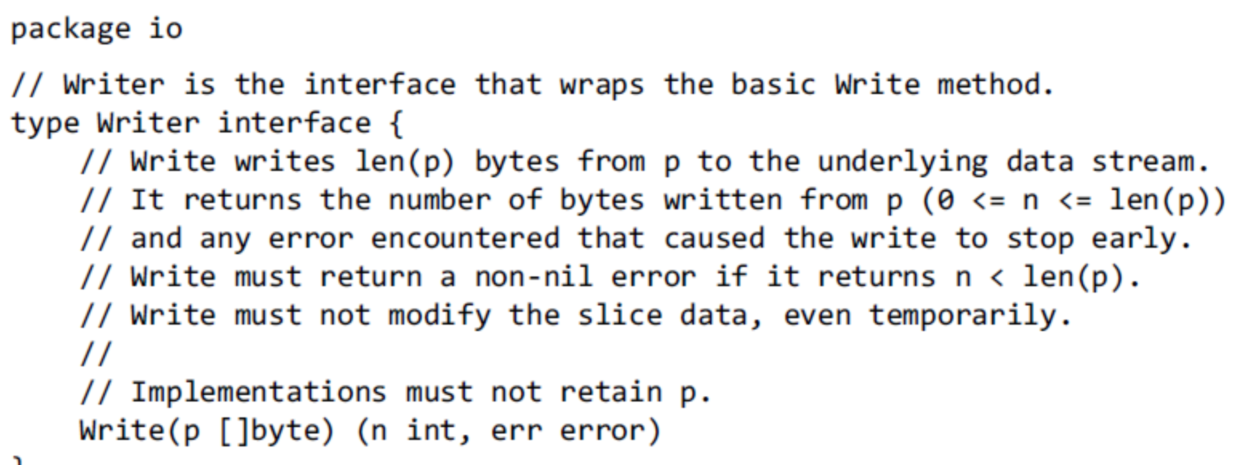
Interfaces as Contracts

- A concrete type specifies the exact representation of its values and exposes the intrinsic operations of that representation, such as arithmetic for numbers, or indexing, append, and range for slices.
- A concrete type may also provide additional behaviors through its methods. When you have a value of a concrete type, you know exactly what it is and what you can do with it.
- **An interface is an abstract type.**
- Interfaces doesn’t expose the representation or internal structure of its values, or the set of basic operations they support; it reveals only some of their methods.
- When you have a value of an interface type, you know nothing about what it is; you know only what it can do, or more precisely, what behaviors are provided by its methods.

Go Programming Language

Interfaces as Contracts

- The first parameter of Fprintf is an io.Writer, which is an interface type with the following declaration:



```
package io

// Writer is the interface that wraps the basic Write method.
type Writer interface {
    // Write writes len(p) bytes from p to the underlying data stream.
    // It returns the number of bytes written from p (0 <= n <= len(p))
    // and any error encountered that caused the write to stop early.
    // Write must return a non-nil error if it returns n < len(p).
    // Write must not modify the slice data, even temporarily.
    //
    // Implementations must not retain p.
    Write(p []byte) (n int, err error)
}
```

- The io.Write llers.
- On the one hand, the contract requires that the caller provide a value of a concrete type like *os.File or *bytes.Buffer that has a method called **Write** with the appropriate signature and behavior.
- On the other hand, the contract guarantees that Fprintf will do its job given any value that satisfies the io.Writer interface

Go Programming Language

Interfaces as Contracts

Example:

- **fmt.Printf** writes the result to the standard output, and **fmt.Sprintf**, returns the result as a string.
- It would be hard if formatting the result, had to be duplicated because of these superficial differences in how the result is used.
- Both of these functions are, in effect, wrappers around a third function, **fmt.Fprintf**, that is agnostic about what happens to the result it computes

```
package fmt

func Fprintf(w io.Writer, format string, args ...interface{}) (int, error)

func Printf(format string, args ...interface{}) (int, error) {
    return Fprintf(os.Stdout, format, args...)
}

func Sprintf(format string, args ...interface{}) string {
    var buf bytes.Buffer
    Fprintf(&buf, format, args...)
    return buf.String()
}
```

Interfaces as Contracts

- fmt.Fprintf assumes nothing about the representation of the value and relies only on the behaviors guaranteed by the io.Writer contract
- We can safely pass a value of any concrete type that satisfies io.Writer as the first argument to fmt.Fprintf.
- This freedom to substitute one type for another that satisfies the same interface is called **substitutability**, and is a hallmark of object-oriented programming

Go Programming Language

Interfaces Types

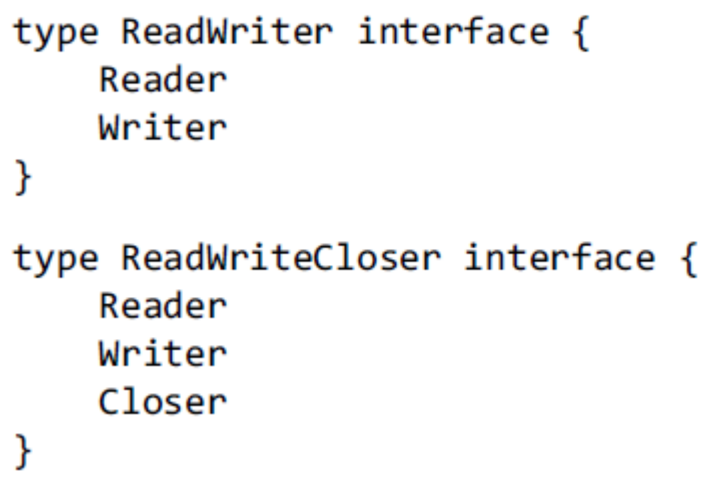
- An interface type specifies a set of methods that a concrete type must possess to be considered an instance of that interface.
- The io.Writer type is one of the most widely used interfaces because it provides an abstraction of all the types to which bytes can be written, which includes files, memory buffers, network connections, HTTP clients, archivers, hashers, and so on.
- The io package defines many other useful interfaces.
- A Reader represents any type from which you can read bytes
- A Closer is any value that you can close, such as a file or a network connection

```
package io

type Reader interface {
    Read(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}
```

Go Programming Language

Interfaces Types

- New interface types may be combinations of existing ones.



```
type ReadWriter interface {
    Reader
    Writer
}

type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}
```

- The interface own as the *empty interface*:

  interface{}

Go Programming Language

Interface satisfaction

- A type satisfies an interface if it possesses all the methods the interface requires.
- For example, an *os.File satisfies io.Reader, Writer, Closer, and ReadWriter.
- A *bytes.Buffer satisfies Reader, Writer, and ReadWriter, but does not satisfy Closer because it does not have a Close method.
- An expression may be assigned to an interface only if its type satisfies the interface

```
var w io.Writer
w = os.Stdout       // OK: *os.File has Write method
```

Go Programming Language

Interface values

- The value of an interface type, or interface value, has two components, a concrete type and a value of that type. These are called the interface's dynamic type and dynamic value.
- The zero value for an interface has both its type and value components set to nil
- You can test whether an interface value is nil using w == nil or w != nil.

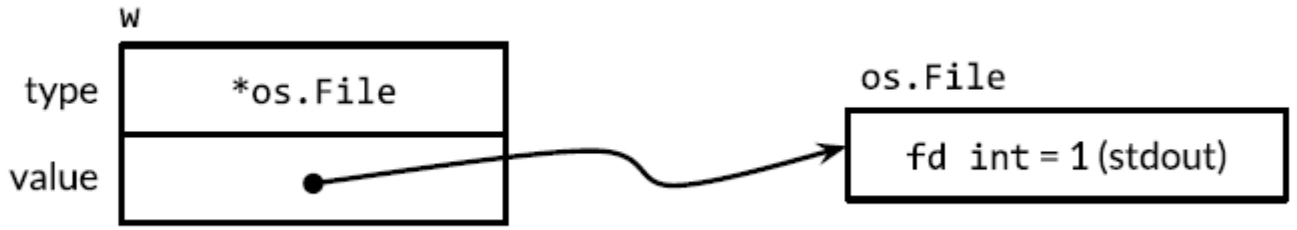


Figure 7.2. An interface value containing an *os.File pointer.

- Calling the Write method on an interface value containing an *os.File pointer causes the (*os.File).Write method to be called

Go Programming Language
Type assertion

- A type assertion is an operation applied to an interface value.
- Syntactically, it looks like x.(T), where x is an expression of an interface type and T is a type, called the ‘‘asserted’’ type.
- A type assertion checks that the dynamic type of its operand matches the asserted type.
- A type assertion to a concrete type extracts the concrete value from its operand
- A type assertion to an interface type changes the type of the expression, making a different (and usually larger) set of methods accessible, but it preserves the dynamic type and value components inside the interface value

Go Programming Language

Type assertion - switch

- 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
}
```

Go Programming Language

Interface - error

- **error** is a built-in interface type with a single method that returns an error message:

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

- Functions often return an error value, and calling code should handle errors by testing whether the error equals nil.
- A nil error denotes success; a non-nil error denotes failure.

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

Go Programming Language

Interface - Reader

- 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.

PES
UNIVERSITY
CELEBRATING 50 YEARS

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

PES UNIVERSITY

CELEBRATING 50 YEARS

Go Programming Language

Goroutines and Channels

Department of Computer Science and Engineering

Go Programming Language
Goroutines

- In Go, each concurrently executing activity is called a **goroutine**.
- Consider a program that has two functions, one that does some computation and one that writes some output, and assume that neither function calls the other.
- A sequential program may call one function and then call the other, but in a concurrent program with two or more goroutines, calls to both functions can be active at the same time.
- A goroutine is similar to a thread.
- A goroutine is an independent function that executes simultaneously in some separate lightweight threads managed by Go
- The differences between threads and goroutines are essentially quantitative, not qualitative
- When a program starts, its only goroutine is the one that calls the main function, so we call it the main goroutine.

Go Programming Language

Goroutines

- New goroutines are created by the go statement
- Syntactically, a go statement is an ordinary function or method call prefixed by the key word **go**.
- A go statement causes the function to be called in a newly created goroutine.
- The go statement itself completes immediately
- **f()** // call f(); wait for it to return
- **go f()** // create a new goroutine that calls f(); don't wait

- Goroutines run in the same address space, so access to shared memory must be synchronized.

Go Programming Language
Channels

- A channel is a communication mechanism that lets one goroutine send values to another goroutine.
- Each channel is a conduit for values of a particular type, called the channel’s element type.
- The type of a channel whose elements have type int is written **chan int**.
- To create a channel, we use the built-in make function:
- **ch := make(chan int)** // ch has type 'chan int‘
- Channel is a reference to the data structure created by make.
- When we copy a channel or pass one as an argument to a function, we are copying a reference, so caller and callee refer to the same data struc ture.
- The zero value of a channel is nil.
- Two channels of the same type may be compared using ==. The comparison is true if both are references to the same channel data structure. A channel may also be compared to nil.

Go Programming Language
Channels - Operations

- A channel has two principal operations, **send** and **receive**, collectively known as communications.
- A send statement transmits a value from one goroutine, through the channel, to another goroutine executing a corresponding receive expression.
- Both operations are written using the **<-** operator.
- In a send statement, the <- separates the channel and value operands.
- In a receive expression, <- precedes the channel operand.
- A receive expression whose result is not used is a valid statement.
- **ch <- x** // a send statement
- **x = <- ch** // a receive expression in an assignment statement
- **<- ch** // a receive statement; result is discarded

Go Programming Language

Channels - Operations

- Channels support a third operation, close, which sets a flag indicating that no more values will ever be sent on this channel
- Receive operations on a closed channel yield the values that have been sent until no more values are left
- To close a channel, we call the built-in close function: **close(ch)**

Go Programming Language
Unbuffered Channels

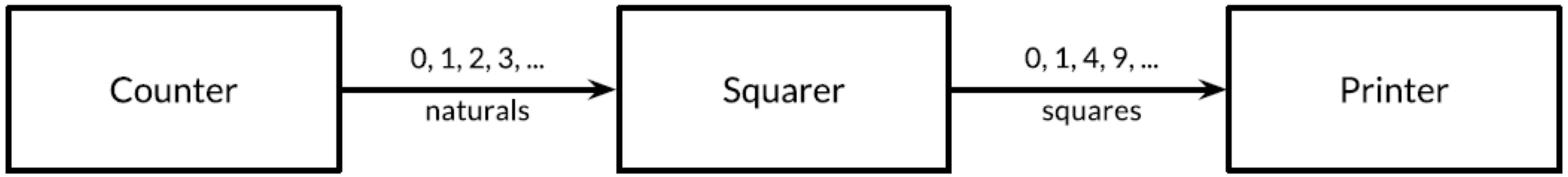
- A send operation on an unbuffered channel blocks the sending goroutine until another goroutine executes a corresponding receive on the same channel, at which point the value is transmitted and both goroutines may continue.
- If the receive operation was attempted first, the receiving goroutine is blocked until another goroutine performs a send on the same channel.
- Communication over an unbuffered channel causes the sending and receiving goroutines to synchronize. Because of this, unbuffered channels are sometimes called synchronous channels

Go Programming Language
Buffered Channels

- A buffered channel has a queue of elements.
- The queue’s maximum size is determined when it is created, by the capacity argument to make.
- **ch = make(chan string, 3)**
- We can send up to three values on this channel without the goroutine blocking
- A fourth send statement would block.

Go Programming Language
Pipelines

- Channels can be used to connect goroutines together so that the output of one is the input to another. This is called a pipeline.



- The first goroutine, counter, generates the integers 0, 1, 2, ..., and sends them over a channel to the second goroutine, squarer, which receives each value, squares it, and sends the result over another channel to the third goroutine, printer, which receives the squared values and prints them

Go Programming Language
Unidirectional Channels

- To document the intent and prevent misuse, the Go type system provides unidirectional channel types that expose only one or the other of the send and receive operations.
- The type chan <- int, a send-only channel of int, allows sends but not receives.
- The type <- chan int, a receive-only channel of int, allows receives but not sends.

```
func counter(out chan int)
func squarer(out, in chan int)
func printer(in chan int)
```

PES UNIVERSITY

CELEBRATING 50 YEARS

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

PES UNIVERSITY

CELEBRATING 50 YEARS

Go Programming Language

Looping in parallel & Multiplexing with select

Department of Computer Science and Engineering

Go Programming Language

Looping in parallel

- Problems that consist entirely of sub-problems that are completely independent of each other are described as **embarrassingly parallel**.
- Embarrassingly parallel problems are the easiest kind to implement concurrently and enjoy performance that scales linearly with the amount of parallelism.
- Looping in parallel is implemented as go routines

Go Programming Language

Example – Web Crawler

```
func main() {
    worklist := make(chan []string)

    // Start with the command-line arguments.
    go func() { worklist <- os.Args[1:] }()

    // Crawl the web concurrently.
    seen := make(map[string]bool)
    for list := range worklist {
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

```
func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

Example – Web Crawler – limiting parallelism

- We can limit parallelism using a buffered channel of capacity **n** to model a concurrency primitive called a counting semaphore.
- Conceptually, each of the n vacant slots in the channel buffer represents a token entitling the holder to proceed.
- Sending a value into the channel acquires a token, and receiving a value from the channel releases a token, creating a new vacant slot.
- This ensures that at most n sends can occur without an intervening receive

Go Programming Language

Example – Web Crawler – limiting parallelism

```go
func main() {
    worklist := make(chan []string)
    var n int // number of pending sends to worklist

    // Start with the command-line arguments.
    n++
    go func() { worklist <- os.Args[1:] }()

    // Crawl the web concurrently.
    seen := make(map[string]bool)
    for ; n > 0; n-- {
        list := <-worklist
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                n++
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

```go
// tokens is a counting semaphore used to
// enforce a limit of 20 concurrent requests.
var tokens = make(chan struct{}, 20)

func crawl(url string) []string {
    fmt.Println(url)
    tokens <- struct{}{} // acquire a token
    list, err := links.Extract(url)
    <-tokens // release the token

    if err != nil {
        log.Print(err)
    }
    return list
}
```

Multiplexing with select

- One of the most powerful tools for managing multiple channels is the **select** statement
- The select statement provides another way to handle multiple channels.
- It is like a switch statement, but for channels
- A select waits until a communication for some case is ready to proceed.
- It then performs that communication and executes the case's associated statements
- The other communications do not happen.
- A select with no cases, select{}, waits forever.

Go Programming Language

Multiplexing with select

```
func main() {
    // ...create abort channel...

    fmt.Println("Commencing countdown.  Press return to abort.")
    select {
    case <-time.After(10 * time.Second):
        // Do nothing.
    case <-abort:
        fmt.Println("Launch aborted!")
        return
    }
    launch()
}


abort := make(chan struct{})
go func() {
    os.Stdin.Read(make([]byte, 1)) // read a single byte
    abort <- struct{}{}
}()
```

Cancellation

- Sometimes we need to instruct a goroutine to stop what it is doing
- There is no way for one goroutine to terminate another directly, since that would leave all its shared variables in undefined states.
- What if we need to cancel two goroutines, or an arbitrary number?
- In general, it's hard to know how many goroutines are working at any given moment.
- For cancellation, what we need is a reliable mechanism to broadcast an event over a channel so that many goroutines can see it as it occurs and can later see that it has occurred.
- First, we create a cancellation channel on which no values are ever sent, but whose closure indicates that it is time for the program to stop what it is doing.
- We also define a utility function, cancelled, that checks or polls the cancellation state at the instant it is called.

PES UNIVERSITY
CELEBRATING 50 YEARS

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering