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

  interface{}

  own as the *empty 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
```

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.

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering