

Types, Methods, and Interfaces

As you saw in earlier chapters, Go is a statically typed language with both built-in types and user-defined types. Like most modern languages, Go allows you to attach methods to types. It also has type abstraction, allowing you to write code that invokes methods without explicitly specifying the implementation.

However, Go's approach to methods, interfaces, and types is very different from that of most other languages in common use today. Go is designed to encourage the best practices advocated by software engineers, avoiding inheritance while encouraging composition. In this chapter, you'll take a look at types, methods, and interfaces, and see how to use them to build testable and maintainable programs.

Types in Go

Back in “[Structs](#)” on page 61, you saw how to define a struct type:

```
type Person struct {
    FirstName string
    LastName  string
    Age       int
}
```

This should be read as declaring a user-defined type with the name `Person` to have the *underlying type* of the struct literal that follows. In addition to struct literals, you can use any primitive type or compound type literal to define a concrete type. Here are a few examples:

```
type Score int
type Converter func(string)Score
type TeamScores map[string]Score
```

Go allows you to declare a type at any block level, from the package block down. However, you can access the type only from within its scope. The only exceptions are types exported from other packages. I'll talk more about those in [Chapter 10](#).



To make it easier to talk about types, I'll define a couple of terms. An *abstract type* is one that specifies *what* a type should do but not *how* it is done. A *concrete type* specifies what and how. This means that the type has a specified way to store its data and provides an implementation of any methods declared on the type. While all types in Go are either abstract or concrete, some languages allow hybrid types, such as abstract classes or interfaces with default methods in Java.

Methods

Like most modern languages, Go supports methods on user-defined types.

The methods for a type are defined at the package block level:

```
type Person struct {
    FirstName string
    LastName  string
    Age       int
}

func (p Person) String() string {
    return fmt.Sprintf("%s %s, age %d", p.FirstName, p.LastName, p.Age)
}
```

Method declarations look like function declarations, with one addition: the *receiver* specification. The receiver appears between the keyword `func` and the name of the method. Like all other variable declarations, the receiver name appears before the type. By convention, the receiver name is a short abbreviation of the type's name, usually its first letter. It is nonidiomatic to use `this` or `self`.

There is one key difference between declaring methods and functions: methods can be defined *only* at the package block level, while functions can be defined inside any block.

Just like functions, method names cannot be overloaded. You can use the same method names for different types, but you can't use the same method name for two different methods on the same type. While this philosophy feels limiting when coming from languages that have method overloading, not reusing names is part of Go's philosophy of making clear what your code is doing.

I'll talk more about packages in [Chapter 10](#), but be aware that methods must be declared in the same package as their associated type; Go doesn't allow you to add methods to types you don't control. While you can define a method in a different file within the same package as the type declaration, it is best to keep your type definition and its associated methods together so that it's easy to follow the implementation.

Method invocations should look familiar to those who have used methods in other languages:

```
p := Person{  
    FirstName: "Fred",  
    LastName:  "Fredson",  
    Age:       52,  
}  
output := p.String()
```

Pointer Receivers and Value Receivers

As I covered in [Chapter 6](#), Go uses parameters of pointer type to indicate that a parameter might be modified by the function. The same rules apply for method receivers too. They can be *pointer receivers* (the type is a pointer) or *value receivers* (the type is a value type). The following rules help you determine when to use each kind of receiver:

- If your method modifies the receiver, you *must* use a pointer receiver.
- If your method needs to handle `nil` instances (see “[Code Your Methods for nil Instances](#)” on page 148), then it *must* use a pointer receiver.
- If your method doesn't modify the receiver, you *can* use a value receiver.

Whether you use a value receiver for a method that doesn't modify the receiver depends on the other methods declared on the type. When a type has *any* pointer receiver methods, a common practice is to be consistent and use pointer receivers for *all* methods, even the ones that don't modify the receiver.

Here's some simple code to demonstrate pointer and value receivers. It starts with a type that has two methods on it, one using a value receiver, the other with a pointer receiver:

```
type Counter struct {  
    total      int  
    lastUpdated time.Time  
}  
  
func (c *Counter) Increment() {  
    c.total++  
    c.lastUpdated = time.Now()  
}
```

```
func (c Counter) String() string {
    return fmt.Sprintf("total: %d, last updated: %v", c.total, c.lastUpdated)
}
```

You can then try out these methods with the following code. You can run it yourself on [The Go Playground](#) or use the code in the `sample_code/pointer_value` directory in the [Chapter 7 repository](#):

```
var c Counter
fmt.Println(c.String())
c.Increment()
fmt.Println(c.String())
```

You should see the following output:

```
total: 0, last updated: 0001-01-01 00:00:00 +0000 UTC
total: 1, last updated: 2009-11-10 23:00:00 +0000 UTC m=+0.000000001
```

One thing you might notice is that you were able to call the pointer receiver method even though `c` is a value type. When you use a pointer receiver with a local variable that's a value type, Go automatically takes the address of the local variable when calling the method. In this case, `c.Increment()` is converted to `(&c).Increment()`.

If you call a value receiver on a pointer variable, Go automatically dereferences the pointer when calling the method. In the code:

```
c := &Counter{}
fmt.Println(c.String())
c.Increment()
fmt.Println(c.String())
```

the call `c.String()` is silently converted to `(*c).String()`.



If you call a value receiver method with pointer instance whose value is `nil`, your code will compile but will panic at runtime (I discuss panics in “[panic and recover](#)” on page 218).

Be aware that the rules for passing values to functions still apply. If you pass a value type to a function and call a pointer receiver method on the passed value, you are invoking the method on a *copy*. You can try out the following code on [The Go Playground](#) or use the code in the `sample_code/update_wrong` directory in the [Chapter 7 repository](#):

```
func doUpdateWrong(c Counter) {
    c.Increment()
    fmt.Println("in doUpdateWrong:", c.String())
}

func doUpdateRight(c *Counter) {
```

```

    c.Increment()
    fmt.Println("in doUpdateRight:", c.String())
}

func main() {
    var c Counter
    doUpdateWrong(c)
    fmt.Println("in main:", c.String())
    doUpdateRight(&c)
    fmt.Println("in main:", c.String())
}

```

When you run this code, you'll get the following output:

```

in doUpdateWrong: total: 1, last updated: 2009-11-10 23:00:00 +0000 UTC
    m+=0.000000001
in main: total: 0, last updated: 0001-01-01 00:00:00 +0000 UTC
in doUpdateRight: total: 1, last updated: 2009-11-10 23:00:00 +0000 UTC
    m+=0.000000001
in main: total: 1, last updated: 2009-11-10 23:00:00 +0000 UTC m+=0.000000001

```

The parameter in `doUpdateRight` is of type `*Counter`, which is a pointer instance. As you can see, you can call both `Increment` and `String` on it. Go considers both pointer and value receiver methods to be in the *method set* for a pointer instance. For a value instance, only the value receiver methods are in the method set. This seems like a pedantic detail right now, but I'll come back to it when talking about interfaces in just a bit.



This can be confusing to new Go programmers (and, to be honest, to not-so-new Go programmers), but Go's automatic conversion from pointer types to value types, and vice versa, is purely syntactic sugar. It is independent of the method set concept. Alexey Gronskiy has written a [blog post](#) that explores in detail why the method set of pointer instances have both pointer and value receiver methods, but the method set of value instances has only value receiver methods.

One final note: do not write getter and setter methods for Go structs unless you need them to meet an interface (I'll start covering interfaces in [“A Quick Lesson on Interfaces” on page 157](#)). Go encourages you to directly access a field. Reserve methods for business logic. The exceptions are when you need to update multiple fields as a single operation or when the update isn't a straightforward assignment of a new value. The `Increment` method defined earlier demonstrates both of these properties.

Code Your Methods for nil Instances

The previous section covered pointer receivers, which might make you wonder what happens when you call a method on a `nil` instance. In most languages, this produces some sort of error. (Objective-C allows you to call a method on a `nil` instance, but it always does nothing.)

Go does something a little different. It actually tries to invoke the method. As mentioned earlier, if it's a method with a value receiver, you'll get a panic, since there is no value being pointed to by the pointer. If it's a method with a pointer receiver, it can work if the method is written to handle the possibility of a `nil` instance.

In some cases, expecting a `nil` receiver makes the code simpler. Here's an implementation of a binary tree that takes advantage of `nil` values for the receiver:

```
type IntTree struct {
    val      int
    left, right *IntTree
}

func (it *IntTree) Insert(val int) *IntTree {
    if it == nil {
        return &IntTree{val: val}
    }
    if val < it.val {
        it.left = it.left.Insert(val)
    } else if val > it.val {
        it.right = it.right.Insert(val)
    }
    return it
}

func (it *IntTree) Contains(val int) bool {
    switch {
    case it == nil:
        return false
    case val < it.val:
        return it.left.Contains(val)
    case val > it.val:
        return it.right.Contains(val)
    default:
        return true
    }
}
```



The `Contains` method doesn't modify the `*IntTree`, but it is declared with a pointer receiver. This demonstrates the rule mentioned previously about supporting a `nil` receiver. A method with a value receiver can't check for `nil` and, as mentioned earlier, panics if invoked with a `nil` receiver.

The following code uses the tree. You can try it out on [The Go Playground](#) or use the code in the `sample_code/tree` directory in the [Chapter 7](#) repository:

```
func main() {
    var it *IntTree
    it = it.Insert(5)
    it = it.Insert(3)
    it = it.Insert(10)
    it = it.Insert(2)
    fmt.Println(it.Contains(2)) // true
    fmt.Println(it.Contains(12)) // false
}
```

It's very clever that Go allows you to call a method on a `nil` receiver, and there are situations where it is useful, like the previous tree node example. However, most of the time it's not very useful. Pointer receivers work like pointer function parameters; it's a copy of the pointer that's passed into the method. Just like `nil` parameters passed to functions, if you change the copy of the pointer, you haven't changed the original. This means you can't write a pointer receiver method that handles `nil` and makes the original pointer non-`nil`.

If your method has a pointer receiver and won't work for a `nil` receiver, you have to decide how your method should handle a `nil` receiver. One choice is to treat it as a fatal flaw, like trying to access a position in a slice beyond its length. In that case, don't do anything and let the code panic. (Also make sure you write good tests, as discussed in [Chapter 15](#).) If a `nil` receiver is something that is recoverable, check for `nil` and return an error (I discuss errors in [Chapter 9](#)).

Methods Are Functions Too

Methods in Go are so much like functions that you can use a method as a replacement for a function anytime there's a variable or parameter of a function type.

Let's start with this simple type:

```
type Adder struct {
    start int
}

func (a Adder) AddTo(val int) int {
    return a.start + val
}
```

You create an instance of the type in the usual way and invoke its method:

```
myAdder := Adder{start: 10}
fmt.Println(myAdder.AddTo(5)) // prints 15
```

You can also assign the method to a variable or pass it to a parameter of type `func(int)int`. This is called a *method value*:

```
f1 := myAdder.AddTo  
fmt.Println(f1(10))           // prints 20
```

A method value is a bit like a closure, since it can access the values in the fields of the instance from which it was created.

You can also create a function from the type itself. This is called a *method expression*:

```
f2 := Adder.AddTo  
fmt.Println(f2(myAdder, 15)) // prints 25
```

In a method expression, the first parameter is the receiver for the method; the function signature is `func(Adder, int) int`.

Method values and method expressions aren't clever corner cases. You'll see one way to use them when you look at dependency injection in "[Implicit Interfaces Make Dependency Injection Easier](#)" on page 174.

Functions Versus Methods

Since you can use a method as a function, you might wonder when you should declare a function and when you should use a method.

The differentiator is whether your function depends on other data. As I've covered several times, package-level state should be effectively immutable. Anytime your logic depends on values that are configured at startup or changed while your program is running, those values should be stored in a struct, and that logic should be implemented as a method. If your logic depends only on the input parameters, it should be a function.

Type Declarations Aren't Inheritance

In addition to declaring types based on built-in Go types and struct literals, you can also declare a user-defined type based on another user-defined type:

```
type HighScore Score  
type Employee Person
```

Many concepts can be considered "object-oriented," but one stands out: *inheritance*. With inheritance, the state and methods of a *parent* type are declared to be available on a *child* type, and values of the child type can be substituted for the parent type.¹

¹ For the computer scientists in the audience, I realize that subtyping is not inheritance. However, most programming languages use inheritance to implement subtyping, so the definitions are often conflated in popular usage.

Declaring a type based on another type looks a bit like inheritance but isn't. The two types have the same underlying type, but that's all. The types have no hierarchy. In languages with inheritance, a child instance can be used anywhere the parent instance is used. The child instance also has all the methods and data structures of the parent instance. That's not the case in Go. You can't assign an instance of type `HighScore` to a variable of type `Score`, or vice versa, without a type conversion, nor can you assign either of them to a variable of type `int` without a type conversion. Furthermore, any methods defined on `Score` aren't defined on `HighScore`:

```
// assigning untyped constants is valid
var i int = 300
var s Score = 100
var hs HighScore = 200
hs = s           // compilation error!
s = i           // compilation error!
s = Score(i)    // ok
hs = HighScore(s) // ok
```

User-defined types whose underlying types are built-in types can be assigned literals and constants compatible with the underlying type. They can also be used with the operators for those types:

```
var s Score = 50
scoreWithBonus := s + 100 // type of scoreWithBonus is Score
```



A type conversion between types that share an underlying type keeps the same underlying storage but associates different methods.

Types Are Executable Documentation

While it's well understood that you should declare a struct type to hold a set of related data, it's less clear when you should declare a user-defined type based on other built-in types or one user-defined type that's based on another user-defined type. The short answer is that types are documentation. They make code clearer by providing a name for a concept and describing the kind of data that is expected. It's clearer for someone reading your code when a method has a parameter of type `Percentage` than of type `int`, and it's harder for it to be invoked with an invalid value.

The same logic applies when declaring one user-defined type based on another user-defined type. When you have the same underlying data, but different sets of operations to perform, make two types. Declaring one as being based on the other avoids some repetition and makes it clear that the two types are related.

iota Is for Enumerations—Sometimes

Many programming languages have the concept of enumerations, which allow you to specify that a type can have only a limited set of values. Go doesn't have an enumeration type. Instead, it has `iota`, which lets you assign an increasing value to a set of constants.



The concept of `iota` comes from the programming language APL (which stood for "A Programming Language"). To generate a list of the first three positive integers in APL, you write `⍳3`, where `⍳` is the lowercase Greek letter iota.

APL is famous for being so reliant on its own custom notation that it required computers with a special keyboard. For example, `(~R∈R∘.×R)/R←1↓iR` is an APL program to find all the prime numbers up to the value of the variable `R`.

It may seem ironic that a language as focused on readability as Go would borrow a concept from a language that is concise to a fault, but this is why you should learn multiple programming languages: you can find inspiration everywhere.

When using `iota`, the best practice is to first define a type based on `int` that will represent all the valid values:

```
type MailCategory int
```

Next, use a `const` block to define a set of values for your type:

```
const (
    Uncategorized MailCategory = iota
    Personal
    Spam
    Social
    Advertisements
)
```

The first constant in the `const` block has the type specified, and its value is set to `iota`. Every subsequent line has neither the type nor a value assigned to it. When the Go compiler sees this, it repeats the type and the assignment to all the subsequent constants in the block, which is `iota`. The value of `iota` increments for each constant defined in the `const` block, starting with 0. This means that 0 is assigned to the first constant (`Uncategorized`), 1 to the second constant (`Personal`), and so on. When a new `const` block is created, `iota` is set back to 0.

The value of `iota` increments for each constant in the `const` block, whether or not `iota` is used to define the value of a constant. The following code demonstrates what happens when `iota` is used intermittently in a `const` block:

```
const (
    Field1 = 0
    Field2 = 1 + iota
    Field3 = 20
    Field4
    Field5 = iota
)

func main() {
    fmt.Println(Field1, Field2, Field3, Field4, Field5)
}
```

You can run this code on [The Go Playground](#) and see the (perhaps unexpected) result:

```
0 2 20 20 4
```

`Field2` is assigned 2 because `iota` has a value of 1 on the second line in the `const` block. `Field4` is assigned 20 because it has no type or value explicitly assigned, so it gets the value of the previous line with a type and assignment. Finally, `Field5` gets the value 4 because it is the fifth line and `iota` starts counting from 0.

This is the best advice I've seen on `iota`:

Don't use *iota* for defining constants where its values are explicitly defined (elsewhere). For example, when implementing parts of a specification and the specification says which values are assigned to which constants, you should explicitly write the constant values. Use *iota* for "internal" purposes only. That is, where the constants are referred to by name rather than by value. That way, you can optimally enjoy *iota* by inserting new constants at any moment in time / location in the list without the risk of breaking everything.

—Danny van Heumen

The important thing to understand is that nothing in Go will stop you (or anyone else) from creating additional values of your type. Furthermore, if you insert a new identifier in the middle of your list of literals, all subsequent ones will be renumbered. This will break your application in a subtle way if those constants represented values in another system or in a database. Given these two limitations, `iota`-based enumerations make sense only when you care about being able to differentiate between a set of values and don't particularly care what the value is behind the scenes. If the actual value matters, specify it explicitly.



Because you can assign a literal expression to a constant, you'll see sample code that suggests you should use `iota` for cases like this:

```
type BitField int

const (
    Field1 BitField = 1 << iota // assigned 1
    Field2                  // assigned 2
    Field3                  // assigned 4
    Field4                  // assigned 8
)
```

While this is clever, be careful when using this pattern. If you do so, document what you are doing. As mentioned previously, using `iota` with constants is fragile when you care about the value. You don't want a future maintainer to insert a new constant in the middle of the list and break your code.

Be aware that `iota` starts numbering from 0. If you are using your set of constants to represent different configuration states, the zero value might be useful. You saw this earlier in the `MailCategory` type. When mail first arrives, it is uncategorized, so the zero value makes sense. If there isn't a sensible default value for your constants, a common pattern is to assign the first `iota` value in the constant block to `_` or to a constant that indicates the value is invalid. This makes it easy to detect when a variable has not been properly initialized.

Use Embedding for Composition

The software engineering advice “Favor object composition over class inheritance” dates back to at least the 1994 book *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley), better known as the Gang of Four book. While Go doesn't have inheritance, it encourages code reuse via built-in support for composition and promotion:

```
type Employee struct {
    Name      string
    ID       string
}

func (e Employee) Description() string {
    return fmt.Sprintf("%s (%s)", e.Name, e.ID)
}

type Manager struct {
    Employee
    Reports []Employee
}

func (m Manager) FindNewEmployees() []Employee {
```

```
// do business logic  
}
```

Note that `Manager` contains a field of type `Employee`, but no name is assigned to that field. This makes `Employee` an *embedded field*. Any fields or methods declared on an embedded field are *promoted* to the containing struct and can be invoked directly on it. That makes the following code valid:

```
m := Manager{  
    Employee: Employee{  
        Name: "Bob Bobson",  
        ID:   "12345",  
    },  
    Reports: []Employee{},  
}  
fmt.Println(m.ID)           // prints 12345  
fmt.Println(m.Description()) // prints Bob Bobson (12345)
```



You can embed any type within a struct, not just another struct. This promotes the methods on the embedded type to the containing struct.

If the containing struct has fields or methods with the same name as an embedded field, you need to use the embedded field's type to refer to the obscured fields or methods. If you have types defined like this:

```
type Inner struct {  
    X int  
}  
  
type Outer struct {  
    Inner  
    X int  
}
```

you can access the `X` on `Inner` only by specifying `Inner` explicitly:

```
o := Outer{  
    Inner: Inner{  
        X: 10,  
    },  
    X: 20,  
}  
fmt.Println(o.X)           // prints 20  
fmt.Println(o.Inner.X) // prints 10
```

Embedding Is Not Inheritance

Built-in embedding support is rare in programming languages (I'm not aware of another popular language that supports it). Many developers who are familiar with inheritance (which is available in many languages) try to understand embedding by treating it as inheritance. That way lies tears. You cannot assign a variable of type `Manager` to a variable of type `Employee`. If you want to access the `Employee` field in `Manager`, you must do so explicitly. You can run the following code on [The Go Playground](#) or use the code in the `sample_code/embedding` directory in the [Chapter 7](#) repository:

```
var eFail Employee = m      // compilation error!
var eOK Employee = m.Employee // ok!
```

You'll get the error:

```
cannot use m (type Manager) as type Employee in assignment
```

Furthermore, Go has no *dynamic dispatch* for concrete types. The methods on the embedded field have no idea they are embedded. If you have a method on an embedded field that calls another method on the embedded field, and the containing struct has a method of the same name, the method on the embedded field is invoked, not the method on the containing struct. This behavior is demonstrated in the following code, which you can run on [The Go Playground](#), or use the code in the `sample_code/no_dispatch` directory in the [Chapter 7](#) repository:

```
type Inner struct {
    A int
}

func (i Inner) IntPrinter(val int) string {
    return fmt.Sprintf("Inner: %d", val)
}

func (i Inner) Double() string {
    return i.IntPrinter(i.A * 2)
}

type Outer struct {
    Inner
    S string
}

func (o Outer) IntPrinter(val int) string {
    return fmt.Sprintf("Outer: %d", val)
}

func main() {
    o := Outer{
        Inner: Inner{
```

```

        A: 10,
    },
    S: "Hello",
}
fmt.Println(o.Double())
}

```

Running this code produces the following output:

```
Inner: 20
```

While embedding one concrete type inside another won't allow you to treat the outer type as the inner type, the methods on an embedded field do count toward the *method set* of the containing struct. This means they can make the containing struct implement an interface.

A Quick Lesson on Interfaces

While Go's concurrency model (which I cover in [Chapter 12](#)) gets all the publicity, the real star of Go's design is its implicit interfaces, the only abstract type in Go. Let's see what makes them so great.

Let's start by taking a quick look at how to declare interfaces. At their core, interfaces are simple. Like other user-defined types, you use the `type` keyword.

Here's the definition of the `Stringer` interface in the `fmt` package:

```
type Stringer interface {
    String() string
}
```

In an interface declaration, an interface literal appears after the name of the interface type. It lists the methods that must be implemented by a concrete type to meet the interface. The methods defined by an interface are the method set of the interface. As I covered in [“Pointer Receivers and Value Receivers” on page 145](#), the method set of a pointer instance contains the methods defined with both pointer and value receivers, while the method set of a value instance contains only the methods with value receivers. Here's a quick example using the `Counter` struct defined previously:

```
type Incrementer interface {
    Increment()
}

var myStringer fmt.Stringer
var myIncrementer Incrementer
pointerCounter := &Counter{}
valueCounter := Counter{}

myStringer = pointerCounter // ok
myStringer = valueCounter // ok
```

```
myIncrementer = pointerCounter // ok
myIncrementer = valueCounter // compile-time error!
```

Trying to compile this code results in the error `cannot use valueCounter` (variable of type `Counter`) as `Incrementer` value in assignment: `Counter` does not implement `Incrementer` (method `Increment` has pointer receiver).

You can try this code on [The Go Playground](#) or use the code in the `sample_code/method_set` directory in the [Chapter 7 repository](#).

Like other types, interfaces can be declared in any block.

Interfaces are usually named with “er” endings. You’ve already seen `fmt.Stringer`, but there are many more, including `io.Reader`, `io.Closer`, `io.ReadCloser`, `json.Marshaler`, and `http.Handler`.

Interfaces Are Type-Safe Duck Typing

So far, nothing that’s been said about the Go interface is much different from interfaces in other languages. What makes Go’s interfaces special is that they are implemented *implicitly*. As you’ve seen with the `Counter` struct type and the `Incrementer` interface type that you’ve used in previous examples, a concrete type does not declare that it implements an interface. If the method set for a concrete type contains all the methods in the method set for an interface, the concrete type implements the interface. Therefore, that the concrete type can be assigned to a variable or field declared to be of the type of the interface.

This implicit behavior makes interfaces the most interesting thing about types in Go, because they enable both type safety and decoupling, bridging the functionality in both static and dynamic languages.

To understand why, let’s talk about why languages have interfaces. Earlier I mentioned that *Design Patterns* taught developers to favor composition over inheritance. Another piece of advice from the book is “Program to an interface, not an implementation.” Doing so allows you to depend on behavior, not on implementation, allowing you to swap implementations as needed. This allows your code to evolve over time, as requirements inevitably change.

Dynamically typed languages like Python, Ruby, and JavaScript don’t have interfaces. Instead, those developers use *duck typing*, which is based on the expression “If it walks like a duck and quacks like a duck, it’s a duck.” The concept is that you can pass an instance of a type as a parameter to a function as long as the function can find a method to invoke that it expects:

```
class Logic:
    def process(self, data):
        # business logic
```

```
def program(logic):
    # get data from somewhere
    logic.process(data)

logicToUse = Logic()
program(logicToUse)
```

Duck typing might sound weird at first, but it's been used to build large and successful systems. If you program in a statically typed language, this sounds like utter chaos. Without an explicit type being specified, it's hard to know exactly what functionality should be expected. As new developers move on to a project or the existing developers forget what the code is doing, they have to trace through the code to identify the actual dependencies.

Java developers use a different pattern. They define an interface, create an implementation of the interface, but refer to the interface only in the client code:

```
public interface Logic {
    String process(String data);
}

public class LogicImpl implements Logic {
    public String process(String data) {
        // business logic
    }
}

public class Client {
    private final Logic logic;
    // this type is the interface, not the implementation

    public Client(Logic logic) {
        this.logic = logic;
    }

    public void program() {
        // get data from somewhere
        this.logic.process(data);
    }
}

public static void main(String[] args) {
    Logic logic = new LogicImpl();
    Client client = new Client(logic);
    client.program();
}
```

Dynamic language developers look at the explicit interfaces in Java and don't see how you can possibly refactor your code over time when you have explicit dependencies.

Switching to a new implementation from a different provider means rewriting your code to depend on a new interface.

Go's developers decided that both groups are right. If your application is going to grow and change over time, you need flexibility to change implementation. However, in order for people to understand what your code is doing (as new people work on the same code over time), you also need to specify what the code depends on. That's where implicit interfaces come in. Go code is a blend of the previous two styles:

```
type LogicProvider struct {}

func (lp LogicProvider) Process(data string) string {
    // business logic
}

type Logic interface {
    Process(data string) string
}

type Client struct{
    L Logic
}

func(c Client) Program() {
    // get data from somewhere
    c.L.Process(data)
}

main() {
    c := Client{
        L: LogicProvider{},
    }
    c.Program()
}
```

The Go code provides an interface, but only the caller (`Client`) knows about it; nothing is declared on `LogicProvider` to indicate that it meets the interface. This is sufficient to both allow a new logic provider in the future and provide executable documentation to ensure that any type passed into the client will match the client's need.



Interfaces specify what callers need. The client code defines the interface to specify what functionality it requires.

This doesn't mean that interfaces can't be shared. You've already seen several interfaces in the standard library that are used for input and output. Having a standard

interface is powerful; if you write your code to work with `io.Reader` and `io.Writer`, it will function correctly whether it is writing to a file on local disk or a value in memory.

Furthermore, using standard interfaces encourages the *decorator pattern*. It is common in Go to write factory functions that take in an instance of an interface and return another type that implements the same interface. For example, say you have a function with the following definition:

```
func process(r io.Reader) error
```

You can process data from a file with the following code:

```
r, err := os.Open(fileName)
if err != nil {
    return err
}
defer r.Close()
return process(r)
```

The `os.File` instance returned by `os.Open` meets the `io.Reader` interface and can be used in any code that reads in data. If the file is gzip-compressed, you can wrap the `io.Reader` in another `io.Reader`:

```
r, err := os.Open(fileName)
if err != nil {
    return err
}
defer r.Close()
gz, err := gzip.NewReader(r)
if err != nil {
    return err
}
defer gz.Close()
return process(gz)
```

Now the exact same code that was reading from an uncompressed file is reading from a compressed file instead.



If an interface in the standard library describes what your code needs, use it! Commonly used interfaces include `io.Reader`, `io.Writer`, and `io.Closer`.

It's perfectly fine for a type that meets an interface to specify additional methods that aren't part of the interface. One set of client code may not care about those methods, but others do. For example, the `io.File` type also meets the `io.Writer` interface. If your code cares only about reading from a file, use the `io.Reader` interface to refer to the file instance and ignore the other methods.

Embedding and Interfaces

Embedding is not only for structs. You can also embed an interface in an interface. For example, the `io.ReadCloser` interface is built out of an `io.Reader` and an `io.Closer`:

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

type Closer interface {
    Close() error
}

type ReadCloser interface {
    Reader
    Closer
}
```



Just as you can embed a concrete type in a struct, you can also embed an interface in a struct. You'll see a use for this in “[Using Stubs in Go](#)” on page 397.

Accept Interfaces, Return Structs

You'll often hear experienced Go developers say that your code should “Accept interfaces, return structs.” This phrase was most likely coined by Jack Lindamood in his 2016 blog post “[Preemptive Interface Anti-Pattern in Go](#)”. It means that the business logic invoked by your functions should be invoked via interfaces, but the output of your functions should be a concrete type. I've already covered why functions should accept interfaces: they make your code more flexible and explicitly declare the exact functionality being used.

The primary reason your functions should return concrete types is they make it easier to gradually update a function's return values in new versions of your code. When a concrete type is returned by a function, new methods and fields can be added without breaking existing code that calls the function, because the new fields and methods are ignored. The same is not true for an interface. Adding a new method to an interface means that all existing implementations of that interface must be updated, or your code breaks. In semantic versioning terms, this is the difference between a minor release that is backward compatible and a major release, which is backward breaking. If you are exposing an API that's consumed by other people (either inside your organization or as part of an open source project), avoiding backward-breaking changes keeps your users happy.

In some rare situations, the least bad option is to have your functions return interfaces. For example, the `database/sql/driver` package in the standard library defines a set of interfaces that define what a database driver must provide. It is the responsibility of the database driver author to provide concrete implementations of these interfaces, so almost all methods on all interfaces defined in `database/sql/driver` return interfaces. Starting in Go 1.8, database drivers are expected to support additional features. The standard library has a compatibility promise, so the existing interfaces can't be updated with new methods, and the existing methods on these interfaces can't be updated to return different types. The solution is to leave the existing interfaces alone, define new interfaces that describe the new functionality, and tell database driver authors that they should implement both the old and new methods on their concrete types.

This leads to the question of how to check whether these new methods are present and how to access them if they are. You'll learn how in [“Type Assertions and Type Switches” on page 167](#).

Rather than writing a single factory function that returns different instances behind an interface based on input parameters, try to write separate factory functions for each concrete type. In some situations (such as a parser that can return one or more kinds of tokens), it's unavoidable and you have no choice but to return an interface.

Errors are an exception to this rule. As you'll see in [Chapter 9](#), Go functions and methods can declare a return parameter of the `error` interface type. In the case of `error`, it's quite likely that different implementations of the interface could be returned, so you need to use an interface to handle all possible options, as interfaces are the only abstract type in Go.

This pattern has one potential drawback. As I discussed in [“Reducing the Garbage Collector’s Workload” on page 136](#), reducing heap allocations improves performance by reducing the amount of work for the garbage collector. Returning a struct avoids a heap allocation, which is good. However, when invoking a function with parameters of interface types, a heap allocation occurs for each interface parameter. Figuring out the trade-off between better abstraction and better performance should be done over the life of your program. Write your code so that it is readable and maintainable. If you find that your program is too slow *and* you have profiled it *and* you have determined that the performance problems are due to a heap allocation caused by an interface parameter, then you should rewrite the function to use a concrete type parameter. If multiple implementations of an interface are passed into the function, this will mean creating multiple functions with repeated logic.



Developers who come from a C++ or Rust background might try using generics as a way to get the compiler to generate specialized functions. As of Go 1.21, this will probably not produce faster code. I'll cover why in “[Idiomatic Go and Generics](#)” on page 199.

Interfaces and nil

When discussing pointers in [Chapter 6](#), I also talked about `nil`, the zero value for pointer types. You can also use `nil` to represent the zero value for an interface instance, but it's not as simple as it is for concrete types.

Understanding the relationship between interfaces and `nil` requires understanding a little bit about how interfaces are implemented. In the Go runtime, interfaces are implemented as a struct with two pointer fields, one for the value and one for the type of the value. As long as the type field is non-`nil`, the interface is non-`nil`. (Since you cannot have a variable without a type, if the value pointer is non-`nil`, the type pointer is always non-`nil`.)

In order for an interface to be considered `nil`, *both* the type and the value must be `nil`. The following code prints out `true` on the first two lines and `false` on the last:

```
var pointerCounter *Counter
fmt.Println(pointerCounter == nil) // prints true
var incrementer Incrementer
fmt.Println(incrementer == nil) // prints true
incrementer = pointerCounter
fmt.Println(incrementer == nil) // prints false
```

You can run this code for yourself on [The Go Playground](#) or use the code in the `sample_code/interface_nil` directory in the [Chapter 7 repository](#).

What `nil` indicates for a variable with an interface type is whether you can invoke methods on it. As I covered earlier, you can invoke methods on `nil` concrete instances, so it makes sense that you can invoke methods on an interface variable that was assigned a `nil` concrete instance. If an interface variable is `nil`, invoking any methods on it triggers a panic (which I'll discuss in “[panic and recover](#)” on page 218). If an interface variable is non-`nil`, you can invoke methods on it. (But note that if the value is `nil` and the methods of the assigned type don't properly handle `nil`, you could still trigger a panic.)

Since an interface instance with a non-`nil` type is not equal to `nil`, it is not straightforward to tell whether the value associated with the interface is `nil` when the type is non-`nil`. You must use reflection (which I'll discuss in “[Use Reflection to Check If an Interface's Value Is nil](#)” on page 417) to find out.

Interfaces Are Comparable

In [Chapter 3](#), you learned about comparable types, the ones that can be checked for equality with `==`. You might be surprised to learn that interfaces are comparable. Just as an interface is equal to `nil` only if its type and value fields are both `nil`, two instances of an interface type are equal only if their types are equal and their values are equal. This suggests a question: what happens if the type isn't comparable? Let's use a simple example to explore this concept. Start with an interface definition and a couple of implementations of that interface:

```
type Doubler interface {
    Double()
}

type DoubleInt int

func (d *DoubleInt) Double() {
    *d = *d * 2
}

type DoubleIntSlice []int

func (d DoubleIntSlice) Double() {
    for i := range d {
        d[i] = d[i] * 2
    }
}
```

The `Double` method on `DoubleInt` is declared with a pointer receiver because you are modifying the value of the `int`. You can use a value receiver for the `Double` method on `DoubleIntSlice` because, as covered in [“The Difference Between Maps and Slices” on page 131](#), you can change the value of an item in a parameter that is a slice type. The `*DoubleInt` type is comparable (all pointer types are), and the `DoubleIntSlice` type is not comparable (slices aren't comparable).

You also have a function that takes in two parameters of type `Doubler` and prints if they are equal:

```
func DoublerCompare(d1, d2 Doubler) {
    fmt.Println(d1 == d2)
}
```

You now define four variables:

```
var di DoubleInt = 10
var di2 DoubleInt = 10
var dis = DoubleIntSlice{1, 2, 3}
var dis2 = DoubleIntSlice{1, 2, 3}
```

Now, you're going to call this function three times. The first call is as follows:

```
DoublerCompare(&di, &di2)
```

This prints out `false`. The types match (both are `*DoubleInt`), but you are comparing pointers, not values, and the pointers point to different instances.

Next, you compare a `*DoubleInt` with a `DoubleIntSlice`:

```
DoublerCompare(&di, dis)
```

This prints out `false`, because the types do not match.

Finally, you get to the problematic case:

```
DoublerCompare(dis, dis2)
```

This code compiles without issue, but triggers a panic at runtime:

```
panic: runtime error: comparing uncomparable type main.DoubleIntSlice
```

The entire program is available in the `sample_code/comparable` directory in the [Chapter 7 repository](#).

Also be aware that the key of a map must be comparable, so a map can be defined to have an interface as a key:

```
m := map[Doubler]int{}
```

If you add a key-value pair to this map and the key isn't comparable, that will also trigger a panic.

Given this behavior, be careful when using `==` or `!=` with interfaces or using an interface as a map key, as it's easy to accidentally generate a panic that will crash your program. Even if all your interface implementations are currently comparable, you don't know what will happen when someone else uses or modifies your code, and there's no way to specify that an interface can be implemented only by comparable types. If you want to be extra safe, you can use the `Comparable` method on `reflect.Value` to inspect an interface before using it with `==` or `!=`. (You'll learn more about reflection in [“Reflection Lets You Work with Types at Runtime” on page 410](#)).

The Empty Interface Says Nothing

Sometimes in a statically typed language, you need a way to say that a variable could store a value of any type. Go uses an *empty interface*, `interface{}`, to represent this:

```
var i interface{}
i = 20
i = "hello"
i = struct {
    FirstName string
    LastName string
} {"Fred", "Fredson"}
```



`interface{}` isn't special case syntax. An empty interface type simply states that the variable can store any value whose type implements zero or more methods. This just happens to match every type in Go.

To improve readability, Go added `any` as a type alias for `interface{}`. Legacy code (written before `any` was added in Go 1.18) used `interface{}`, but stick with `any` for new code.

Because an empty interface doesn't tell you anything about the value it represents, you can't do a lot with it. One common use of `any` is as a placeholder for data of uncertain schema that's read from an external source, like a JSON file:

```
data := map[string]any{}
contents, err := os.ReadFile("testdata/sample.json")
if err != nil {
    return err
}
json.Unmarshal(contents, &data)
// the contents are now in the data map
```



User-created data containers that were written before generics were added to Go use an empty interface to store a value. (I'll talk about generics in [Chapter 8](#).) An example in the standard library is [container/list](#). Now that generics are part of Go, please use them for any newly created data containers.

If you see a function that takes in an empty interface, it's likely using reflection (which I'll talk about in [Chapter 16](#)) to either populate or read the value. In the preceding example, the second parameter of the `json.Unmarshal` function is declared to be of type `any`.

These situations should be relatively rare. Avoid using `any`. As you've seen, Go is designed as a strongly typed language and attempts to work around this are unidiomatic.

If you find yourself in a situation where you had to store a value into an empty interface, you might be wondering how to read the value back again. To do that, you need to learn about type assertions and type switches.

Type Assertions and Type Switches

Go provides two ways to see if a variable of an interface type has a specific concrete type or if the concrete type implements another interface. Let's start by looking at type assertions. A *type assertion* names the concrete type that implemented the interface,

or names another interface that is also implemented by the concrete type whose value is stored in the interface. You can try it out on [The Go Playground](#) or in the `typeAssert` function in `main.go` in the `sample_code/type_assertions` directory in the [Chapter 7 repository](#):

```
type MyInt int

func main() {
    var i any
    var mine MyInt = 20
    i = mine
    i2 := i.(MyInt)
    fmt.Println(i2 + 1)
}
```

In the preceding code, the variable `i2` is of type `MyInt`.

You might wonder what happens if a type assertion is wrong. In that case, your code panics. You can try it out on [The Go Playground](#) or in the `typeAssertPanicWrongType` function in `main.go` in the `sample_code/type_assertions` directory in the [Chapter 7 repository](#):

```
i2 := i.(string)
fmt.Println(i2)
```

Running this code produces the following panic:

```
panic: interface conversion: interface {} is main.MyInt, not string
```

As you've already seen, Go is very careful about concrete types. Even if two types share an underlying type, a type assertion must match the type of the value stored in the interface. The following code panics. You can try it out on [The Go Playground](#) or in the `typeAssertPanicTypeNotIdentical` function in `main.go` in the `sample_code/type_assertions` directory in the [Chapter 7 repository](#):

```
i2 := i.(int)
fmt.Println(i2 + 1)
```

Obviously, crashing is not desired behavior. You avoid this by using the comma ok idiom, just as you saw in “[The comma ok Idiom](#)” on page 58 when detecting whether a zero value was in a map. You can see this in the `typeAssertCommaOK` function in `main.go` in the `sample_code/type_assertions` directory in the [Chapter 7 repository](#):

```
i2, ok := i.(int)
if !ok {
    return fmt.Errorf("unexpected type for %v", i)
}
fmt.Println(i2 + 1)
```

The boolean `ok` is set to `true` if the type conversion was successful. If it was not, `ok` is set to `false` and the other variable (in this case `i2`) is set to its zero value. You then

handle the unexpected condition within an `if` statement. I'll talk more about error handling in [Chapter 9](#).



A type assertion is very different from a type conversion. Conversions change a value to a new type, while assertions reveal the type of the value stored in the interface. Type conversions can be applied to both concrete types and interfaces. Type assertions can be applied only to interface types. All type assertions are checked at runtime, so they can fail at runtime with a panic if you don't use the comma ok idiom. Most type conversions are checked at compile time, so if they are invalid, your code won't compile. (Type conversions between slices and array pointers can fail at runtime and don't support the comma ok idiom, so be careful when using them!)

Even if you are absolutely certain that your type assertion is valid, use the comma ok idiom version. You don't know how other people (or you in six months) will reuse your code. Sooner or later, your unvalidated type assertions will fail at runtime.

When an interface could be one of multiple possible types, use a *type switch* instead:

```
func doThings(i any) {
    switch j := i.(type) {
    case nil:
        // i is nil, type of j is any
    case int:
        // j is of type int
    case MyInt:
        // j is of type MyInt
    case io.Reader:
        // j is of type io.Reader
    case string:
        // j is a string
    case bool, rune:
        // i is either a bool or rune, so j is of type any
    default:
        // no idea what i is, so j is of type any
    }
}
```

A type switch looks a lot like the `switch` statement that you saw way back in [“switch” on page 84](#). Instead of specifying a boolean operation, you specify a variable of an interface type and follow it with `.(type)`. Usually, you assign the variable being checked to another variable that's valid only within the `switch`.



Since the purpose of a type switch is to derive a new variable from an existing one, it is idiomatic to assign the variable being switched on to a variable of the same name (`i := i.(type)`), making this one of the few places where shadowing is a good idea. To make the comments more readable, this example doesn't use shadowing.

The type of the new variable depends on which case matches. You can use `nil` for one case to see if the interface has no associated type. If you list more than one type on a case, the new variable is of type `any`. Just as with a `switch` statement, you can have a `default` case that matches when no specified type does. Otherwise, the new variable has the type of the case that matches.

While the examples so far have used the `any` interface with type assertions and type switches, you can uncover the concrete type from all interface types.



If you *don't* know the type of the value stored in an interface, you need to use reflection. I'll talk more about reflection in [Chapter 16](#).

Use Type Assertions and Type Switches Sparingly

While being able to extract the concrete implementation from an interface variable might seem handy, you should use these techniques infrequently. For the most part, treat a parameter or return value as the type that was supplied and not what else it could be. Otherwise, your function's API isn't accurately declaring the types it needs to perform its task. If you needed a different type, it should be specified.

That said, type assertions and type switches are useful in some use cases. One common use of a type assertion is to see if the concrete type behind the interface also implements another interface. This allows you to specify optional interfaces. For example, the standard library uses this technique to allow more efficient copies when the `io.Copy` function is called. This function has two parameters of types `io.Writer` and `io.Reader` and calls the `io.copyBuffer` function to do its work. If the `io.Writer` parameter also implements `io.WriterTo`, or the `io.Reader` parameter also implements `io.ReaderFrom`, most of the work in the function can be skipped:

```
// copyBuffer is the actual implementation of Copy and CopyBuffer.  
// if buf is nil, one is allocated.  
func copyBuffer(dst Writer, src Reader, buf []byte) (written int64, err error) {  
    // If the reader has a WriteTo method, use it to do the copy.  
    // Avoids an allocation and a copy.  
    if wt, ok := src.(WriterTo); ok {  
        return wt.WriteTo(dst)  
    }  
}
```

```

// Similarly, if the writer has a ReadFrom method, use it to do the copy.
if rt, ok := dst.(ReaderFrom); ok {
    return rt.ReadFrom(src)
}
// function continues...
}

```

Another place optional interfaces are used is when evolving an API. As was covered in “Accept Interfaces, Return Structs” on page 162, the API for the database drivers has changed over time. One of the reasons for this change is the addition of the context, which is discussed in Chapter 14. Context is a parameter that’s passed to functions that provides, among other things, a standard way to manage cancellation. It was added to Go in version 1.7, which means older code doesn’t support it. This includes older database drivers.

In Go 1.8, new context-aware analogues of existing interfaces were defined in the `database/sql/driver` package. For example, the `StmtExecContext` interface defines a method called `ExecContext`, which is a context-aware replacement for the `Exec` method in `Stmt`. When an implementation of `Stmt` is passed into standard library database code, it checks whether it also implements `StmtExecContext`. If it does, `ExecContext` is invoked. If not, the Go standard library provides a fallback implementation of the cancellation support provided by newer code:

```

func ctxDriverStmtExec(ctx context.Context, si driver.Stmt,
    nvdargs []driver.NamedValue) (driver.Result, error) {
    if siCtx, is := si.(driver.StmtExecContext); is {
        return siCtx.ExecContext(ctx, nvdargs)
    }
    // fallback code is here
}

```

This optional interface technique has one drawback. You saw earlier that it is common for implementations of interfaces to use the decorator pattern to wrap other implementations of the same interface to layer behavior. The problem is that if an optional interface is implemented by one of the wrapped implementations, you cannot detect it with a type assertion or type switch. For example, the standard library includes a `bufio` package that provides a buffered reader. You can buffer any other `io.Reader` implementation by passing it to the `bufio.NewReader` function and using the returned `*bufio.Reader`. If the passed-in `io.Reader` also implemented `io.ReaderFrom`, wrapping it in a buffered reader prevents the optimization.

You also see this when handling errors. As mentioned earlier, they implement the `error` interface. Errors can include additional information by wrapping other errors. A type switch or type assertion cannot detect or match wrapped errors. If you want different behaviors to handle different concrete implementations of a returned error, use the `errors.Is` and `errors.As` functions to test for and access the wrapped error.

Type switch statements provide the ability to differentiate between multiple implementations of an interface that require different processing. They are most useful when only certain possible valid types can be supplied for an interface. Be sure to include a `default` case in the type switch to handle implementations that aren't known at development time. This protects you if you forget to update your type switch statements when adding new interface implementations:

```
func walkTree(t *treeNode) (int, error) {
    switch val := t.val.(type) {
    case nil:
        return 0, errors.New("invalid expression")
    case number:
        // we know that t.val is of type number, so return the
        // int value
        return int(val), nil
    case operator:
        // we know that t.val is of type operator, so
        // find the values of the left and right children, then
        // call the process() method on operator to return the
        // result of processing their values.
        left, err := walkTree(t.lchild)
        if err != nil {
            return 0, err
        }
        right, err := walkTree(t.rchild)
        if err != nil {
            return 0, err
        }
        return val.process(left, right), nil
    default:
        // if a new treeVal type is defined, but walkTree wasn't updated
        // to process it, this detects it
        return 0, errors.New("unknown node type")
    }
}
```

You can see the complete implementation on [The Go Playground](#) or in the `sample_code/type_switch` directory in the [Chapter 7](#) repository.



You can further protect yourself from unexpected interface implementations by making the interface unexported and at least one method unexported. If the interface is exported, it can be embedded in a struct in another package, making the struct implement the interface. I'll talk more about packages and exporting identifiers in [Chapter 10](#).

Function Types Are a Bridge to Interfaces

I haven't talked about one last thing with type declarations. Once you understand the concept of declaring a method on a struct, you can start to see how a user-defined type with an underlying type of `int` or `string` can have a method as well. After all, a method provides business logic that interacts with the state of an instance, and integers and strings have state as well.

Go, however, allows methods on *any* user-defined type, including user-defined function types. This sounds like an academic corner case, but they are actually very useful. They allow functions to implement interfaces. The most common usage is for HTTP handlers. An HTTP handler processes an HTTP server request. It's defined by an interface:

```
type Handler interface {
    ServeHTTP(w http.ResponseWriter, r *http.Request)
}
```

By using a type conversion to `http.HandlerFunc`, any function that has the signature `func(http.ResponseWriter, *http.Request)` can be used as an `http.Handler`:

```
type HandlerFunc func(w http.ResponseWriter, r *http.Request)

func (f HandlerFunc) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    f(w, r)
}
```

This lets you implement HTTP handlers using functions, methods, or closures using the exact same code path as the one used for other types that meet the `http.Handler` interface.

Functions in Go are first-class concepts, and as such, they are often passed as parameters into functions. Meanwhile, Go encourages small interfaces, and an interface of only one method could easily replace a parameter of a function type. The question becomes: when should your function or method specify an input parameter of a function type, and when should you use an interface?

If your single function is likely to depend on many other functions or other state that's not specified in its input parameters, use an interface parameter and define a function type to bridge a function to the interface. That's what's done in the `http` package; a `Handler` is likely just the entry point for a chain of calls that needs to be configured. However, if it's a simple function (like the one used in `sort.Slice`), then a parameter of a function type is a good choice.

Implicit Interfaces Make Dependency Injection Easier

In the preface, I compared writing software to building bridges. One of the things that software has in common with physical infrastructure is that any program used for a lengthy period of time by multiple people will need maintenance. While programs don't wear out, developers are often asked to update programs to fix bugs, add features, and run in new environments. Therefore, you should structure your programs in ways that make them easier to modify. Software engineers talk about *decoupling* code, so that changes to different parts of a program have no effect on one another.

One of the techniques that has been developed to ease decoupling is called *dependency injection*. Dependency injection is the concept that your code should explicitly specify the functionality it needs to perform its task. It's quite a bit older than you might think; in 1996, Robert Martin wrote an article called "[The Dependency Inversion Principle](#)".

One of the surprising benefits of Go's implicit interfaces is that they make dependency injection an excellent way to decouple your code. While developers in other languages often use large, complicated frameworks to inject their dependencies, the truth is that it is easy to implement dependency injection in Go without any additional libraries. Let's work through a simple example to see how to use implicit interfaces to compose applications via dependency injection.

To understand this concept better and see how to implement dependency injection in Go, you'll build a very simple web application. (I'll talk more about Go's built-in HTTP server support in "[The Server](#)" on page 337; consider this a preview.) Start by writing a small utility function, a logger:

```
func LogOutput(message string) {
    fmt.Println(message)
}
```

Another thing your app needs is a data store. Let's create a simple one:

```
type SimpleDataStore struct {
    userData map[string]string
}

func (sds SimpleDataStore) UserNameForID(userID string) (string, bool) {
    name, ok := sds.userData(userID]
    return name, ok
}
```

Also define a factory function to create an instance of a `SimpleDataStore`:

```
func NewSimpleDataStore() SimpleDataStore {
    return SimpleDataStore{
        userData: map[string]string{
```

```

        "1": "Fred",
        "2": "Mary",
        "3": "Pat",
    },
}

```

Next, write some business logic that looks up a user and says hello or goodbye. Your business logic needs some data to work with, so it requires a data store. You also want your business logic to log when it is invoked, so it depends on a logger. However, you don't want to force it to depend on `LogOutput` or `SimpleDataStore`, because you might want to use a different logger or data store later. What your business logic needs are interfaces to describe what it depends on:

```

type DataStore interface {
    UserNameForID(userID string) (string, bool)
}

type Logger interface {
    Log(message string)
}

```

To make your `LogOutput` function meet this interface, you define a function type with a method on it:

```

type LoggerAdapter func(message string)

func (lg LoggerAdapter) Log(message string) {
    lg(message)
}

```

By a stunning coincidence, `LoggerAdapter` and `SimpleDataStore` happen to meet the interfaces needed by your business logic, but neither type has any idea that it does.

Now that you have the dependencies defined, let's look at the implementation of your business logic:

```

type SimpleLogic struct {
    l  Logger
    ds DataStore
}

func (sl SimpleLogic) SayHello(userID string) (string, error) {
    sl.l.Log("in SayHello for " + userID)
    name, ok := sl.ds.UserNameForID(userID)
    if !ok {
        return "", errors.New("unknown user")
    }
    return "Hello, " + name, nil
}

func (sl SimpleLogic) SayGoodbye(userID string) (string, error) {

```

```

sl.l.Log("in SayGoodbye for " + userID)
name, ok := sl.ds.UserNameForID(userID)
if !ok {
    return "", errors.New("unknown user")
}
return "Goodbye, " + name, nil
}

```

You have a `struct` with two fields: one a `Logger`, the other a `DataStore`. Nothing in `SimpleLogic` mentions the concrete types, so there's no dependency on them. There's no problem if you later swap in new implementations from an entirely different provider, because the provider has nothing to do with your interface. This is very different from explicit interfaces in languages like Java. Even though Java uses an interface to decouple the implementation from the interface, the explicit interfaces bind the client and the provider together. This makes replacing a dependency in Java (and other languages with explicit interfaces) far more difficult than it is in Go.

When you want a `SimpleLogic` instance, you call a factory function, passing in interfaces and returning a struct:

```

func NewSimpleLogic(l Logger, ds DataStore) SimpleLogic {
    return SimpleLogic{
        l: l,
        ds: ds,
    }
}

```



The fields in `SimpleLogic` are unexported. This means they can be accessed only by code within the same package as `SimpleLogic`. You can't enforce immutability in Go, but limiting which code can access these fields makes their accidental modification less likely. I'll talk more about exported and unexported identifiers in [Chapter 10](#).

Now you get to your API. You're going to have only a single endpoint, `/hello`, which says hello to the person whose user ID is supplied. (Please do not use query parameters in your real applications for authentication information; this is just a quick sample.) Your controller needs business logic that says hello, so you define an interface for that:

```

type Logic interface {
    SayHello(userID string) (string, error)
}

```

This method is available on your `SimpleLogic` struct, but once again, the concrete type is not aware of the interface. Furthermore, the other method on `SimpleLogic`, `SayGoodbye`, is not in the interface because your controller doesn't care about it. The

interface is owned by the client code, so its method set is customized to the needs of the client code:

```
type Controller struct {
    l    Logger
    logic Logic
}

func (c Controller) SayHello(w http.ResponseWriter, r *http.Request) {
    c.l.Log("In SayHello")
    userID := r.URL.Query().Get("user_id")
    message, err := c.logic.SayHello(userID)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte(err.Error()))
        return
    }
    w.Write([]byte(message))
}
```

Just as you have factory functions for your other types, let's write one for the Controller:

```
func NewController(l Logger, logic Logic) Controller {
    return Controller{
        l:    l,
        logic: logic,
    }
}
```

Again, you accept interfaces and return structs.

Finally, you wire up all your components in your `main` function and start your server:

```
func main() {
    l := LoggerAdapter(LogOutput)
    ds := NewSimpleDataStore()
    logic := NewSimpleLogic(l, ds)
    c := NewController(l, logic)
    http.HandleFunc("/hello", c.SayHello)
    http.ListenAndServe(":8080", nil)
}
```

You can find the complete code for this application in the `sample_code/dependency_injection` directory in the [Chapter 7 repository](#).

The `main` function is the only part of the code that knows what all the concrete types actually are. If you want to swap in different implementations, this is the only place that needs to change. Externalizing the dependencies via dependency injection means that you limit the changes that are needed to evolve your code over time.

Dependency injection is also a great pattern for making testing easier. It shouldn't be surprising, since writing unit tests is effectively reusing your code in a different

environment, one that constrains the inputs and outputs to validate functionality. For example, you can validate the logging output in a test by injecting a type that captures the log output and meets the `Logger` interface. I'll talk about this more in [Chapter 15](#).



The line `http.HandleFunc("/hello", c.SayHello)` demonstrates two points I talked about earlier.

First, you are treating the `SayHello` method as a function.

Second, the `http.HandleFunc` function takes in a function and converts it to an `http.HandlerFunc` function type, which declares a method to meet the `http.Handler` interface, which is the type used to represent a request handler in Go. You took a method from one type and converted it into another type with its own method. That's pretty neat.

Wire

If you feel that writing dependency injection code by hand is too much work, you can use [Wire](#), a dependency injection helper written by Google. Wire uses code generation to automatically create the concrete type declarations that you wrote yourself in `main`.

Go Isn't Particularly Object-Oriented (and That's Great)

Now that you've taken a look at the idiomatic use of types in Go, you can see that categorizing Go as a particular style of language is difficult. It clearly isn't a strictly procedural language. At the same time, Go's lack of method overriding, inheritance, or, well, objects means that it is also not a particularly object-oriented language. Go has function types and closures, but it isn't a functional language either. If you attempt to shoehorn Go into one of these categories, the result is nonidiomatic code.

If you had to label Go's style, the best word to use is *practical*. It borrows concepts from many places with the overriding goal of creating a language that is simple, readable, and maintainable by large teams for many years.

Exercises

In these exercises, you're going to build a program that uses what you've learned about types, methods, and interfaces. Answers are available in the `exercise_solutions` directory in the [Chapter 7 repository](#).

1. You have been asked to manage a basketball league and are going to write a program to help you. Define two types. The first one, called `Team`, has a field for

the name of the team and a field for the player names. The second type is called `League` and has a field called `Teams` for the teams in the league and a field called `Wins` that maps a team's name to its number of wins.

2. Add two methods to `League`. The first method is called `MatchResult`. It takes four parameters: the name of the first team, its score in the game, the name of the second team, and its score in the game. This method should update the `Wins` field in `League`. Add a second method to `League` called `Ranking` that returns a slice of the team names in order of wins. Build your data structures and call these methods from the `main` function in your program using some sample data.
3. Define an interface called `Ranker` that has a single method called `Ranking` that returns a slice of strings. Write a function called `RankPrinter` with two parameters, the first of type `Ranker` and the second of type `io.Writer`. Use the `io.WriteString` function to write the values returned by `Ranker` to the `io.Writer`, with a newline separating each result. Call this function from `main`.

Wrapping Up

In this chapter, you learned about types, methods, interfaces, and their best practices. In the next chapter, you'll learn about generics, which improve readability and maintainability by allowing you to reuse logic and custom-written containers with different types.

