CHAPTER 4

Object Types

In the preceding chapter, I discussed some built-in object types. But I have not yet explained object types themselves. As I mentioned in Chapter 1, Swift object types come in three flavors: enum, struct, and class. What are the differences between them? And how would you create your own object type?

In this chapter, I'll describe first object types generally and then each of the three flavors. Then I'll explain three Swift ways of giving an object type greater flexibility: protocols, generics, and extensions. Finally, I'll complete the survey of Swift's main built-in types with three umbrella types and three collection types.

Object Type Declarations and Features

Object types are declared with the flavor of the object type (`enum`, `struct`, or `class`), the name of the object type (which should start with a capital letter), and curly braces:

```
class Manny {
}
struct Moe {
}
enum Jack {
}
```

The visibility of an object type to other code — its scope — depends upon where its declaration appears (compare "Variable Scope and Lifetime" on page 71):

Top level
: Object types declared at the top level of a file will, by default, be visible to all files in the same module. This is the usual place for object type declarations.

Inside another type declaration

Sometimes it's useful to declare a type inside the declaration of another type, giving it a namespace. This is called a *nested type*.

Function body

An object type declared within the body of a function will exist only inside the scope of the curly braces that surround it; such declarations are legal but rare.

Declarations for any object type may contain within their curly braces the following things:

Initializers

An object type is merely the *type* of an object. The purpose of declaring an object type will usually (though not always) be so that you can make an actual object — an *instance* — that *has* this type. An *initializer* is a function, declared and called in a special way, allowing you to do that.

Properties

A variable declared at the top level of an object type declaration is a *property*.

By default, a property is an *instance property*. An instance property is scoped to an instance: it is accessed through a particular instance of this type, and its value can be different for every instance of this type.

Alternatively, a property can be a *static/class property*. For an enum or struct, it is declared with the keyword `static`; for a class, it may instead be declared with the keyword `class`. It belongs to the object type itself: it is accessed through the type, and it has just one value, associated with the type.

Methods

A function declared at the top level of an object type declaration is a *method*.

By default, a method is an *instance method*: it is called by sending a message to a particular instance of this type. Inside an instance method, `self` is the instance.

Alternatively, a method can be a *static/class method*. For an enum or struct, it is declared with the keyword `static`; for a class, it may be declared instead with the keyword `class`. It is called by sending a message to the type. Inside a static/class method, `self` is the type.

Subscripts

A subscript is a special kind of method, called by appending square brackets to an instance reference or type name.

Object type declarations

An object type declaration can contain an object type declaration — a nested type. From inside the containing object type, the nested type is in scope; from outside the containing object type, the nested type must be referred to through the containing object type. The containing object type is a namespace for the nested type.

Initializers

An *initializer* is a function for producing an instance of an object type. Strictly speaking, it is a static/class method, because it is called by talking to the object type. It is usually called by means of special syntax: the name of the type is followed directly by parentheses, as if the type itself were a function. When an initializer is called, a new instance is created and returned as a result. You will usually do something with the returned instance, such as assigning it to a variable, in order to preserve it and work with it in subsequent code.

Suppose we have a Dog class:

```
class Dog {
}
```

Then we can make a Dog instance like this:

```
Dog()
```

That code, however, though legal, is silly — so silly that it warrants a warning from the compiler. We have created a Dog instance, but there is no reference to that instance. Without such a reference, the Dog instance comes into existence and then immediately vanishes in a puff of smoke. The usual sort of thing is more like this:

```
let fido = Dog()
```

Now our Dog instance will persist as long as the variable `fido` persists (see Chapter 3) and the variable `fido` gives us a reference to our Dog instance so that we can use it.

Observe that `Dog()` calls an initializer even though our Dog class doesn't declare any initializers! The reason is that object types may have *implicit initializers*. These are a convenience that save you the trouble of writing your own initializers. But you *can* write your own initializers, and you will often do so.

How to write an initializer

An initializer is a kind of function, but its declaration syntax doesn't involve the keyword `func` or a return type. Instead, you use the keyword `init` with a parameter list, followed by curly braces containing the code. An object type can have multiple initializers, distinguished by their parameters. A frequent use of the parameters is to set the values of instance properties.

Here's a Dog class with two instance properties, `name` (a String) and `license` (an Int). We give these instance properties default values that are effectively placeholders — an empty string and the number zero. Then we declare three initializers, so that the caller can create a Dog instance in three different ways: by supplying a name, by supplying a license number, or by supplying both. In each initializer, the parameters that are supplied are used to set the values of the corresponding properties:

```
class Dog {
    var name = ""
    var license = 0
    init(name:String) {
        self.name = name
    }
    init(license:Int) {
        self.license = license
    }
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
```

In that code, in each initializer, I've given each parameter the same name as the property to which it corresponds. There's no reason to do that apart from stylistic clarity. In the initializer function body, I can distinguish the parameter from the property by using `self` explicitly to access the property.

The result of that declaration is that I can create a Dog in three different ways:

```
let fido = Dog(name:"Fido")
let rover = Dog(license:1234)
let spot = Dog(name:"Spot", license:1357)
```

But now I can't create a Dog with *no* initializer parameters. I wrote initializers, so my implicit initializer went away. This code is no longer legal:

```
let puff = Dog() // compile error
```

I could *make* that code legal by declaring an initializer with no parameters:

```
class Dog {
    var name = ""
    var license = 0
    init() {
    }
    init(name:String) {
        self.name = name
    }
    init(license:Int) {
        self.license = license
    }
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
```

Now, the truth is that we don't need those four initializers, because an initializer is a function, and a function's parameters can have default values. I can condense all that code into a single initializer, like this:

```
class Dog {
    var name = ""
    var license = 0
    init(name:String = "", license:Int = 0) {
        self.name = name
        self.license = license
    }
}
```

I can still make an actual Dog instance in four different ways:

```
let fido = Dog(name:"Fido")
let rover = Dog(license:1234)
let spot = Dog(name:"Spot", license:1357)
let puff = Dog()
```

Now comes the really interesting part. In my property declarations, I can *eliminate* the assignment of default initial values (as long as I declare explicitly the *type* of each property):

```
class Dog {
    var name : String // no default value!
    var license : Int // no default value!
    init(name:String = "", license:Int = 0) {
        self.name = name
        self.license = license
    }
}
```

That code is legal (and common) — because an initializer initializes! In other words, I don't have to give my properties initial values in their declarations, *provided I give them initial values in all initializers*. That way, I am guaranteed that all my instance properties have values when the instance comes into existence, which is what matters. Conversely, an instance property without an initial value when the instance comes into existence *is illegal*. A property *must* be initialized either as part of its declaration or by every initializer, and the compiler will stop you otherwise.

The Swift compiler's insistence that all instance properties be properly initialized is a valuable feature of Swift. (Contrast Objective-C, where instance properties can go uninitialized — and often do, leading to mysterious errors later.) Don't fight the compiler; work with it. The compiler will help you by giving you an error message ("Return from initializer without initializing all stored properties") until *all* your initializers initialize *all* your instance properties:

```
class Dog {
    var name : String
    var license : Int
    init(name:String = "") {
        self.name = name // compile error (do you see why?)
    }
}
```

Because setting an instance property in an initializer counts as initialization, it is legal even if the instance property is a constant declared with `let`:

```
class Dog {
    let name : String
    let license : Int
    init(name:String = "", license:Int = 0) {
        self.name = name
        self.license = license
    }
}
```

In our artificial examples, we have been very generous with our initializers: we are letting the caller instantiate a Dog without supplying a `name:` argument or a `license:` argument. Usually, however, the purpose of an initializer is just the opposite: we want to *force* the caller to supply *all* needed information at instantiation time. In real life, it is much more likely that our Dog class would look like this:

```
class Dog {
    let name : String
    let license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
```

In that code, our Dog has a `name` property and a `license` property, and values for these *must* be supplied at instantiation time (there are no default values), and those values can never be changed thereafter (the properties are constants). In this way, we enforce a rule that every Dog must have a meaningful name and license. There is now only *one* way to make a Dog:

```
let spot = Dog(name:"Spot", license:1357)
```

Deferred initialization of properties

Sometimes there is no meaningful value that can be assigned to an instance property during initialization. Perhaps the initial value of this property will not be obtained until some time has elapsed *after* this instance has come into existence. This situation conflicts with the requirement that all instance properties be initialized either in their declaration or through an initializer. You could circumvent the problem by assigning a default initial value anyway; but this fails to communicate to your own code the fact that this isn't a "real" value.

A common solution, as I explained in Chapter 3, is to declare your instance property as a `var` having an Optional type. An Optional has a value, namely `nil`, signifying that no "real" value has been supplied; and an Optional `var` is initialized to `nil` automatically. Your code can test this instance property against `nil` and, if it is `nil`, it won't use the property. Later, the property will be given its "real" value. Of course, that value is now wrapped in an Optional; but if you declare this property as an implicitly unwrapped Optional, you can use the wrapped value directly, without explicitly unwrapping it — as if this weren't an Optional at all — once you're sure it is safe to do so:

```
// this property will be set automatically when the nib loads
@IBOutlet var myButton: UIButton!
// this property will be set after time-consuming gathering of data
var albums : [MPMediaItemCollection]?
```

Referring to self

An initializer may refer to an already initialized instance property, and may refer to an uninitialized instance property in order to initialize it. Otherwise, an initializer *may not refer to `self`*, explicitly or implicitly, until *all* instance properties have been initialized. This rule guarantees that the instance is fully formed before it is used. This code is illegal:

```
struct Cat {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        meow() // too soon - compile error
        self.license = license
    }
    func meow() {
        print("meow")
    }
}
```

The call to the instance method `meow` is implicitly a reference to `self` — it means `self.meow()`. The initializer can say that, but not until it has fulfilled its primary contract of initializing all uninitialized properties. The call to the instance method `meow` simply needs to be moved down one line, so that it comes *after* both `name` and `license` have been initialized.

Delegating initializers

Initializers within an object type can call one another by using the syntax `self.init(...)`. An initializer that calls another initializer is called a *delegating initializer*. When an initializer delegates, the other initializer — the one that it delegates to — must completely initialize the instance first, and then the delegating initializer can work with the fully initialized instance, possibly setting again a `var` property that was already set by the initializer that it delegated to.

A delegating initializer appears to be an exception to the rule against saying `self` too early. But it isn't, because it is saying `self` in order to delegate — and delegating will cause all instance properties to be initialized. In fact, the rules about a delegating initializer saying `self` are even more stringent: a delegating initializer *cannot refer to `self` at all*, not even to set a property, until *after* the call to the other initializer:

```
struct Digit {
    var number : Int
    var meaningOfLife : Bool
    init(number:Int) {
        self.number = number
        self.meaningOfLife = false
    }
    init() { // this is a delegating initializer
        self.init(number:42)
        self.meaningOfLife = true // legal
    }
}
```

A delegating initializer *cannot set a constant property* (a `let` variable). That is because it cannot refer to the property until after it has called the other initializer, and at that point the instance is fully formed — initialization proper is over, and the door for initialization of properties has closed. This property is a constant, it has been initialized, and that's that. The preceding code would be illegal if `meaningOfLife` were declared with `let`, because the second initializer is a delegating initializer and cannot set a constant property.

Be careful not to delegate recursively! If you tell an initializer to delegate to itself, or if you create a vicious circle of delegating initializers, the compiler won't stop you, but your running app will hang. Don't say this:

```
struct Digit {
    var number : Int = 100
    init(value:Int) {
        self.init(number:value) // do not do this!
    }
    init(number:Int) {
        self.init(value:number) // do not do this!
    }
}
```

Failable initializers

An initializer can return an Optional wrapping the new instance. In this way, `nil` can be returned to signal failure. An initializer that behaves this way is a *failable initializer*. To mark an initializer as failable when declaring it, put a question mark after the keyword `init`. If your failable initializer needs to return `nil`, explicitly write `return nil`; that's legal even if you haven't fulfilled the initializer contract of initializing all stored properties.

Here's a Dog with an initializer that returns an Optional, returning `nil` if the `name:` argument is invalid:

```
class Dog {
    let name : String
    init?(name:String) {
        if name.isEmpty {
            return nil
        }
        self.name = name
    }
}
```

The resulting value is typed as an Optional wrapping a Dog, and the caller will need to unwrap that Optional (if isn't `nil`) before sending any messages to it.

To exit early from a failable initializer while signalling success, say simply `return`. Here's a rewrite of the previous example:

```
class Dog {
    let name : String
    init?(name:String) {
        if !name.isEmpty {
            self.name = name
            return
        }
        return nil
    }
}
```

Any Objective-C initializer can return `nil` to signal failure. For example, the UIImage initializer `init?(named:)` is failable because there might be no image with the given name. But most Objective-C initializers are not *bridged* as failable initializers; this is essentially the same hand-tweaking policy I described in "Why Optionals?" on page 121.)

Properties

A *property* is a variable — one that happens to be declared at the top level of an object type declaration. This means that everything said about variables in Chapter 3 applies. A property has a fixed type; it can be declared with `var` or `let`; it can be stored or computed; it can have setter observers. An instance property can also be declared `lazy`.

A stored instance property must be given an initial value. As I explained a moment ago, this doesn't have to happen through assignment in the declaration; it can happen through initializer functions instead. Setter observers are not called during initialization of properties.

How properties are accessed

If a property is an instance property (the default), it can be accessed only through an instance, and its value is separate for each instance. To illustrate, let's start once again with a Dog class:

```
class Dog {
    let name : String
    let license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
```

Our Dog class has a `name` instance property. Then we can make two different Dog instances with two different `name` values, and we can access each Dog instance's `name` through the instance:

```
let fido = Dog(name:"Fido", license:1234)
let spot = Dog(name:"Spot", license:1357)
let aName = fido.name // "Fido"
let anotherName = spot.name // "Spot"
```

A static/class property, on the other hand, is accessed through the type, and is scoped to the type, which usually means that it is global and unique. I'll use a struct as an example:

```
struct Greeting {
    static let friendly = "hello there"
    static let hostile = "go away"
}
```

Now code elsewhere can fetch the values of `Greeting.friendly` and `Greeting.hostile`. That example is neither artificial nor trivial; immutable static properties are a convenient and effective way to supply your code with nicely namespaced constants.

Property initialization and self

A property declaration that assigns an initial value to the property *cannot fetch an instance property or call an instance method*. Such behavior would require a reference, explicit or implicit, to `self`; and during initialization, there is no `self` yet — `self` is exactly what we are in the process of initializing. Making this mistake can result in some of Swift's most perplexing compile error messages. This is illegal (and removing the explicit references to `self` doesn't make it legal):

```
class Moi {
    let first = "Matt"
    let last = "Neuburg"
    let whole = self.first + " " + self.last // compile error
}
```

There are two common solutions in that situation:

Make this a computed property
: A computed property can refer to `self` in a getter or setter function because the function won't run until the property is accessed, and that won't happen until after `self` has been fully initialized:

```
    class Moi {
        let first = "Matt"
        let last = "Neuburg"
        var whole : String {
            self.first + " " + self.last
        }
    }
```

Declare this property `lazy`
: A `lazy` property can refer to `self` in its initializer because the initializer won't be evaluated until the property is accessed, and that won't happen until after `self` has been fully initialized:

```
    class Moi {
        let first = "Matt"
        let last = "Neuburg"
        lazy var whole = self.first + " " + self.last
    }
```

As I demonstrated in Chapter 3, a variable can be initialized as part of its declaration using multiple lines of code by means of a define-and-call anonymous function. If this variable is an instance property, and if the function code refers to `self`, the variable must also be declared `lazy`:

```
class Moi {
    let first = "Matt"
    let last = "Neuburg"
    lazy var whole : String = {
        var s = self.first
        s.append(" ")
```

```
            s.append(self.last)
            return s
        }()
    }
```

Unlike instance properties, static properties *can* be initialized with reference to one another; the reason is that static property initializers *are* lazy:

```
    struct Greeting {
        static let friendly = "hello there"
        static let hostile = "go away"
        static let ambivalent = friendly + " but " + hostile
    }
```

Notice the lack of `self` in that code. In static/class code, `self` means the type itself. I like to use `self` explicitly wherever it would be implicit, but here I can't use it without arousing the ire of the compiler (I regard this as a bug). To clarify the status of the terms `friendly` and `hostile`, I can use the type name (or the term `Self`, as I'll explain later in this chapter):

```
    struct Greeting {
        static let friendly = "hello there"
        static let hostile = "go away"
        static let ambivalent = Greeting.friendly + " but " + Greeting.hostile
    }
```

On the other hand, if I write `ambivalent` as a computed property, I *can* use `self`:

```
    struct Greeting {
        static let friendly = "hello there"
        static let hostile = "go away"
        static var ambivalent : String {
            self.friendly + " but " + self.hostile
        }
    }
```

On the other other hand, I'm not allowed to use `self` when the initial value is set by a define-and-call anonymous function (again, I regard this as a bug):

```
    struct Greeting {
        static let friendly = "hello there"
        static let hostile = "go away"
        static var ambivalent : String = {
            self.friendly + " but " + self.hostile // compile error
        }()
    }
```

Methods

A *method* is a function — one that happens to be declared at the top level of an object type declaration. This means that everything said about functions in Chapter 2 applies.

By default, a method is an instance method. This means that it can be accessed only through an instance. Within the body of an instance method, `self` is the instance. To illustrate, let's continue to develop our Dog class:

```
class Dog {
    let name : String
    let license : Int
    let whatDogsSay = "woof"
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    func bark() {
        print(self.whatDogsSay)
    }
    func speak() {
        self.bark()
        print("I'm \(self.name)")
    }
}
```

Now I can make a Dog instance and tell it to speak:

```
let fido = Dog(name:"Fido", license:1234)
fido.speak() // woof I'm Fido
```

In my Dog class, the `speak` method calls the instance method `bark` by way of `self`, and obtains the value of the instance property `name` by way of `self`; and the `bark` instance method obtains the value of the instance property `whatDogsSay` by way of `self`. This is because instance code can use `self` to refer to this instance. Such code can omit `self` if the reference is unambiguous; I could have written this:

```
func speak() {
    bark()
    print("I'm \(name)")
}
```

A static/class method is accessed through the type. Within the body of a static/class method, `self` means the type:

```
struct Greeting {
    static let friendly = "hello there"
    static func beFriendly() {
        print(self.friendly)
    }
}
```

And here's how to call the static `beFriendly` method:

```
Greeting.beFriendly() // hello there
```

There is a kind of conceptual wall between static/class members, on the one hand, and instance members on the other; even though they may be declared within the

The Secret Life of Instance Methods

Here's a secret: instance methods are actually static/class methods. This is legal (but strange):

```
class MyClass {
    var s = ""
    func store(_ s:String) {
        self.s = s
    }
}
let m = MyClass()
let f = MyClass.store(m) // what just happened!?
```

Even though `store` is an instance method, we are able to call it as a class method — with a parameter that is an instance of this class! The reason is that an instance method is actually a curried static/class method composed of two functions — one function that takes an instance, and another function that takes the parameters of the instance method. After that code, `f` is the *second* of those functions, and can be called as a way of passing a parameter to the `store` method *of the instance `m`*:

```
f("howdy")
print(m.s) // howdy
```

same object type declaration, they inhabit different worlds. A static/class method can't refer to "the instance" because there is no instance; thus, a static/class method cannot directly refer to any instance properties or call any instance methods. An instance method, on the other hand, can refer to the type, and can thus access static/class properties and can call static/class methods.

Let's return to our Dog class and grapple with the question of what dogs say. Presume that all dogs say the same thing. We'd prefer, therefore, to express `whatDogsSay` not at instance level but at class level. This would be a good use of a static property. Here's a simplified Dog class that illustrates:

```
class Dog {
    static var whatDogsSay = "woof"
    func bark() {
        print(Dog.whatDogsSay)
    }
}
```

(Instead of `Dog.whatDogsSay`, a Dog instance method can say `Self.whatDogsSay`, as I'll explain later in this chapter.) Now we can make a Dog instance and tell it to bark:

```
let fido = Dog()
fido.bark() // woof
```

Subscripts

A *subscript* is a method that is called by appending square brackets containing arguments directly to a reference. You can use this feature for whatever you like, but it is suitable particularly for situations where this is an object type with *elements* that can be appropriately accessed by key or by index number. I have already described (in Chapter 3) the use of this syntax with strings, and it is familiar also from dictionaries and arrays; you can use square brackets with strings and dictionaries and arrays exactly because Swift's String and Dictionary and Array types declare subscript methods.

The syntax for declaring a subscript method is somewhat like a function declaration and somewhat like a computed property declaration. That's no coincidence. A subscript is like a function in that it can take parameters: arguments can appear in the square brackets when a subscript method is called. A subscript is like a computed property in that the call is used like a reference to a property: you can fetch its value or you can assign into it.

To illustrate the syntax, here's a struct that treats an integer as if it were a digit sequence, returning a digit that can be specified by an index number in square brackets; for simplicity, I'm deliberately omitting any error-checking:

```
struct Digit {
    var number : Int
    init(_ n:Int) {
        self.number = n
    }
    subscript(ix:Int) -> Int { ❶ ❷
        get { ❸
            let s = String(self.number)
            return Int(String(s[s.index(s.startIndex, offsetBy:ix)]))!
        }
    }
}
```

❶ After the keyword `subscript` we have a parameter list stating what parameters are to appear inside the square brackets. By default, *parameter names are not externalized*; if you want a parameter name to be externalized, your declaration must include an external name before the internal name, even if they are the same name — for example, `subscript(ix ix:Int)`. This is different from how external names work everywhere else in Swift (and therefore I regard it as a bug in the language).

❷ Then we have the type of value that is passed out (when the getter is called) or in (when the setter is called); this is parallel to the type declared for a computed property, except that (oddly) the type is preceded by the arrow operator instead of a colon.

❸ Finally, we have curly braces whose contents are exactly like those of a computed property. You can have `get` and curly braces for the getter, and `set` and curly braces for the setter. The setter can be omitted (as here); in that case, the word `get` and its curly braces can be omitted. If the getter consists of a single statement, the keyword `return` can be omitted. The setter receives the new value as `newValue`, but you can change that name by supplying a different name in parentheses after the word `set`.

Here's an example of calling the getter; the instance with appended square brackets containing the arguments is used just as if you were getting a property value:

```
let d = Digit(1234)
let aDigit = d[1] // 2
```

Now I'll expand my Digit struct so that its subscript method includes a setter (and again I'll omit error-checking):

```
struct Digit {
    var number : Int
    init(_ n:Int) {
        self.number = n
    }
    subscript(ix:Int) -> Int {
        get {
            let s = String(self.number)
            return Int(String(s[s.index(s.startIndex, offsetBy:ix)]))!
        }
        set {
            var s = String(self.number)
            let i = s.index(s.startIndex, offsetBy:ix)
            s.replaceSubrange(i...i, with: String(newValue))
            self.number = Int(s)!
        }
    }
}
```

And here's an example of calling the setter; the instance with appended square brackets containing the arguments is used just as if you were setting a property value:

```
var d = Digit(1234)
d[0] = 2 // now d.number is 2234
```

An object type can declare multiple subscript methods, distinguished by their parameters.

Starting in Swift 5.1, a subscript can be a static/class method. I'll demonstrate later, when we talk about enums.

Starting in Swift 5.2, a subscript can have default parameter values. I can declare my Digit subscript method like this (though I can't think why I'd want to):

```
    subscript(ix:Int = 0) -> Int {
```

And then I can call it like this:

```
    var d = Digit(1234)
    let aDigit = d[] // 1
```

Nested Object Types

An object type may be declared inside an object type declaration, forming a nested type:

```
    class Dog {
        struct Noise {
            static var noise = "woof"
        }
        func bark() {
            print(Dog.Noise.noise)
        }
    }
```

A nested object type is no different from any other object type, but the rules for referring to it from the outside are changed; the surrounding object type acts as a namespace, and must be referred to explicitly in order to access the nested object type:

```
    Dog.Noise.noise = "arf"
```

Here, the Noise struct is namespaced inside the Dog class. This namespacing provides clarity: the name Noise does not float free, but is explicitly associated with the Dog class to which it belongs. Namespacing also allows more than one Noise type to exist, without any clash of names. Swift built-in object types often take advantage of namespacing; for example, the String struct is one of several structs that contain an Index struct, with no clash of names.

A nested type can't refer directly to the surrounding type's instance members, but it can refer directly to the surrounding type's static/class members:

```
    class Dog {
        static let sound = "ruff"
        struct Noise {
            static var noise = "woof"
            func barkTheDog() { bark() } // compile error
            var othernoise = sound // fine!
        }
        func bark() {
            print(Dog.Noise.noise)
        }
    }
```

In that example, code inside Noise cannot refer directly to Dog's `bark` method, because it's an instance method, which can be referred to only by way of some specific instance of Dog. But code inside Noise *can* refer directly to Dog's `sound` static property. Moreover, it can do so *without explicit namespacing* — that is, it doesn't have to say `Dog.sound`. In effect, the term `sound` is global in scope to the nested type.

Enums

An *enum* is an object type whose instances represent *distinct predefined alternative values*. Think of it as a list of known possibilities. An enum is the Swift way to express a set of constants that are alternatives to one another. An enum declaration includes case statements. Each case is the name of one of the alternatives. An instance of an enum will represent exactly one alternative — one case.

In my Albumen app, different instances of the same view controller can list any of four different sorts of music library contents: albums, playlists, podcasts, or audiobooks. The view controller's behavior is slightly different in each case. So I need a sort of four-way switch that I can set once when the view controller is instantiated, saying which sort of contents this view controller is to display. That sounds like an enum!

Here's the basic declaration for that enum; I call it Filter, because each case represents a different way of filtering the contents of the music library:

```
enum Filter {
    case albums
    case playlists
    case podcasts
    case books
}
```

That enum doesn't have an initializer. You *can* write an initializer for an enum, as I'll demonstrate in a moment; but there is a default mode of initialization that you'll probably use most of the time — the name of the enum followed by dot-notation and one of the cases. Here's how to make an instance of Filter representing the `albums` case:

```
let type = Filter.albums
```

If the type is known in advance, you can omit the name of the enum; the bare case must still be preceded by a dot:

```
let type : Filter = .albums
```

You can't say `.albums` just anywhere out of the blue, because Swift doesn't know what enum it belongs to. But in that code, the variable is explicitly declared as a Filter, so Swift knows what `.albums` means. A similar thing happens when passing an enum instance as an argument in a function call:

```
func filterExpecter(_ type:Filter) {}
filterExpecter(.albums)
```

In the second line, I create an instance of Filter and pass it, all in one move, without having to include the name of the enum. That's because Swift knows from the function declaration that a Filter is expected here.

In real life, the space savings when omitting the enum name can be considerable — especially because, when talking to Cocoa, the enum type names are often long:

```
let v = UIView()
v.contentMode = .center
```

A UIView's `contentMode` property is typed as a UIView.ContentMode enum. Our code is neater and simpler because we don't have to include the type name explicitly here; `.center` is nicer than `UIView.ContentMode.center`. But either is legal.

Instances of an enum with the same case are regarded as equal. You can compare an enum instance for equality against a case. Again, the type of enum is known from the first term in the comparison, so the second term can omit the enum name:

```
func filterExpecter(_ type:Filter) {
    if type == .albums {
        print("it is albums")
    }
}
filterExpecter(.albums) // "it is albums"
```

Raw Values

Optionally, when you declare an enum, you can add a type declaration. Every case then carries with it a fixed (constant) value of that type. The types attached to an enum's cases in this way are limited to numbers and strings, and the values assigned must be literals.

If the type is an integer numeric type, the values can be implicitly assigned, and will start at zero by default:

```
enum PepBoy : Int {
    case manny
    case moe
    case jack
}
```

In that code, `.manny` carries a value of `0`, `.moe` carries of a value of `1`, and so on.

If the type is String, the implicitly assigned values are the string equivalents of the case names:

```
enum Filter : String {
    case albums
    case playlists
    case podcasts
    case books
}
```

In that code, `.albums` carries a value of `"albums"`, and so on.

Regardless of the type, you can assign values explicitly as part of the case declarations, like this:

```
enum Normal : Double {
    case fahrenheit = 98.6
    case centigrade = 37
}
enum PepBoy : Int {
    case manny = 1
    case moe // 2 implicitly
    case jack = 4
}
enum Filter : String {
    case albums = "Albums"
    case playlists = "Playlists"
    case podcasts = "Podcasts"
    case books = "Audiobooks"
}
```

The values carried by the cases are called their *raw values*. An enum with a type declaration implicitly adopts the RawRepresentable protocol, meaning that it implicitly has an `init(rawValue:)` initializer and a `rawValue` property. (I'll explain later what a protocol is.) So you can retrieve a case's assigned value as its `rawValue`:

```
let type = Filter.albums
print(type.rawValue) // Albums
```

Having each case carry a fixed raw value can be quite useful. In my Albumen app, the Filter cases really do have those String values, and `type` is a Filter instance property of the view controller; when the view controller wants to know what title string to put at the top of the screen, it simply retrieves `self.type.rawValue`.

The raw value associated with each case must be unique within this enum; the compiler will enforce this rule. Therefore, the mapping works the other way: given a raw value, you can derive the case; in particular, you can instantiate an enum that has raw values by using its `init(rawValue:)` initializer:

```
let type = Filter(rawValue:"Albums")
```

However, the attempt to instantiate the enum in this way still might fail, because you might supply a raw value corresponding to *no* case; therefore, this is a failable initializer, and the value returned is an Optional. In that code, `type` is not a Filter; it's an Optional wrapping a Filter. This might not be terribly important, however, because the thing you are most likely to want to do with an enum is to compare it for equality with a case of the enum; you can do that with an Optional without unwrapping it. This code is legal and works correctly:

```
let type = Filter(rawValue:"Albums")
if type == .albums { // ...
```

Associated Values

The raw values discussed in the preceding section are fixed in the enum's declaration: a given case carries with it a certain raw value, and that's that. But there's also a way to construct a case whose constant value can be set *when the instance is created*. The attached value here is called an *associated value*.

To write an enum with one or more cases taking an associated value, do not declare any raw value type for the enum as a whole; instead, you append to the name of the case an expression that looks very much like a tuple declaration — that is, parentheses containing a list of possibly labeled types. Unlike a raw value, your choice of type is not limited. Most often, a single value will be attached to a case, so you'll write parentheses containing a single type name. Here's an example:

```
enum MyError {
    case number(Int)
    case message(String)
    case fatal
}
```

That code means that, at instantiation time, a MyError instance with the `.number` case must be assigned an Int value, a MyError instance with the `.message` case must be assigned a String value, and a MyError instance with the `.fatal` case can't be assigned any value. Instantiation with assignment of a value is really a way of calling an initialization function, so to supply the value, you pass it as an argument in parentheses:

```
let err : MyError = .number(4)
```

This is an ordinary function call, so the argument doesn't have to be a literal:

```
let num = 4
let err : MyError = .number(num)
```

At the risk of sounding like a magician explaining his best trick, I can now reveal how an Optional works. An Optional is simply an enum with two cases: `.none` and `.some`. If it is `.none`, it carries no associated value, and it equates to `nil`. If it is `.some`, it carries the wrapped value as its associated value.

Inference of Type Name with Static/Class Members

Just as you can use a dot and the name of an enum case where an instance of that enum is expected, you can do the same thing when referring to a type's static/class member whose value is an instance of that type. For example, UIColor has many class properties that produce a UIColor instance, so you can omit `UIColor` where a UIColor is expected:

```
p.trackTintColor = .red // instead of UIColor.red
```

Starting in Swift 5.4, you can append a property or method call that also yields the expected type:

```
p.trackTintColor = .red.withAlphaComponent(0.5)
```

Similarly, suppose we have a struct Thing with static constants whose values are Thing instances:

```
struct Thing : RawRepresentable {
    let rawValue : Int
    static let one : Thing = Thing(rawValue:1)
    static let two : Thing = Thing(rawValue:2)
}
```

Then we can refer to `Thing.one` as `.one` where a Thing instance is expected:

```
let thing : Thing = .one
```

Many Objective-C enums are bridged to Swift as that kind of struct, as I'll explain later in the chapter.

By the same rule, when a certain type of instance is expected, that type's initializer can be used without saying the name of the type, by saying `.init` instead. Suppose Dog has an initializer that expects a `name:` parameter, and `dogExpecter` is a function that takes a Dog as its parameter:

```
struct Dog {
    let name: String
}
func dogExpecter(_ dog: Dog) {
    print(dog.name)
}
```

Then we can create and pass a Dog to `dogExpecter` without using the term Dog, and this is regarded as good Swift style:

```
dogExpecter(.init(name:"Fido"))
```

If a case's associated value type has a label, that label must be used at initialization time:

```
enum MyError2 {
    case number(Int)
    case message(String)
    case fatal(n:Int, s:String)
}
let err : MyError2 = .fatal(n:-12, s:"Oh the horror")
```

By default, the == operator cannot be used to compare cases of an enum if any case of that enum has an associated value:

```
if err == MyError.fatal { // compile error
```

But if you declare this enum explicitly as adopting the Equatable protocol (discussed later in this chapter and in Chapter 5), the == operator starts working:

```
enum MyError : Equatable { // *
    case number(Int)
    case message(String)
    case fatal
}
```

That code won't compile, however, unless all the associated types are themselves Equatable. That makes sense; if we declare `case pet(Dog)` and there is no way to know whether any two Dogs are equal, there is obviously no way to know whether any two `pet` cases are equal.

I'll explain in Chapter 5 how to *check the case* of an instance of an enum that has an associated value case, as well as how to *extract* the associated value from an enum instance that has one.

Enum Case Iteration

It is often useful to have a list — that is, an array — of all the cases of an enum. You could define this list manually as a static property of the enum:

```
enum Filter : String {
    case albums = "Albums"
    case playlists = "Playlists"
    case podcasts = "Podcasts"
    case books = "Audiobooks"
    static let cases : [Filter] = [.albums, .playlists, .podcasts, .books]
}
```

That, however, is error-prone and hard to maintain; if, as you develop your program, you modify the enum's cases, you must remember to modify the `cases` property to match. Instead, the list of cases can be generated for you *automatically*. Simply have your enum adopt the CaseIterable protocol (adoption of protocols is explained later in this chapter); now the list of cases springs to life as a static property called `all-Cases`:

```
enum Filter : String, CaseIterable {
    case albums = "Albums"
    case playlists = "Playlists"
    case podcasts = "Podcasts"
    case books = "Audiobooks"
    // static allCases is now [.albums, .playlists, .podcasts, .books]
}
```

I'll put this feature to use in the next section.

Automatic generation of `allCases` is impossible if any of the enum's cases has an associated value, as it would then be unclear how that case should be defined in the list.

Enum Initializers

An explicit enum initializer must do what default initialization does: it must return a particular case of this enum. To do so, set `self` to the case. In this example, I'll expand my Filter enum so that it can be initialized with a numeric argument:

```
enum Filter : String, CaseIterable {
    case albums = "Albums"
    case playlists = "Playlists"
    case podcasts = "Podcasts"
    case books = "Audiobooks"
    init(_ ix:Int) {
        self = Filter.allCases[ix]
    }
}
```

Now there are three ways to make a Filter instance:

```
let type1 = Filter.albums
let type2 = Filter(rawValue:"Playlists")!
let type3 = Filter(2) // .podcasts
```

In that example, we'll crash in the third line if the caller passes a number that's out of range (less than 0 or greater than 3). If we want to avoid that, we can make this a failable initializer and return `nil` if the number is out of range:

```
enum Filter : String, CaseIterable {
    case albums = "Albums"
    case playlists = "Playlists"
    case podcasts = "Podcasts"
    case books = "Audiobooks"
    init?(_ ix:Int) {
        if !Filter.allCases.indices.contains(ix) {
            return nil
        }
        self = Filter.allCases[ix]
    }
}
```

An enum can have multiple initializers. Enum initializers can delegate to one another by saying `self.init(...)`. The only requirement is that, at some point in the chain of calls, `self` must be set to a case; if that doesn't happen, your enum won't compile.

In this example, I improve my Filter enum so that it can be initialized with a String raw value without having to say `rawValue:` in the call. To do so, I declare a failable initializer with a string parameter that delegates to the built-in failable `rawValue:` initializer:

```
enum Filter : String, CaseIterable {
    case albums = "Albums"
    case playlists = "Playlists"
    case podcasts = "Podcasts"
    case books = "Audiobooks"
    init?(_ ix:Int) {
        if !Filter.allCases.indices.contains(ix) {
            return nil
        }
        self = Filter.allCases[ix]
    }
    init?(_ rawValue:String) {
        self.init(rawValue:rawValue)
    }
}
```

Now there are four ways to make a Filter instance:

```
let type1 = Filter.albums
let type2 = Filter(rawValue:"Playlists")!
let type3 = Filter(2)
let type4 = Filter("Audiobooks")!
```

Enum Properties

An enum can have instance properties and static properties, but there's a limitation: an enum instance property can't be a stored property. Computed instance properties are fine, however, and the value of the property can depend on the case of `self`. In this example from my real code, I've associated an MPMediaQuery (obtained by calling an MPMediaQuery factory class method) with each case of my Filter enum, suitable for fetching the songs of that type from the music library:

```
enum Filter : String {
    case albums = "Albums"
    case playlists = "Playlists"
    case podcasts = "Podcasts"
    case books = "Audiobooks"
    var query : MPMediaQuery {
        switch self {
        case .albums:
            return .albums()
        case .playlists:
            return .playlists()
        case .podcasts:
            return .podcasts()
        case .books:
            return .audiobooks()
        }
    }
}
```

If an enum instance property is a computed variable with a setter, other code can assign to this property. However, that code's reference to the enum instance itself must be a variable (`var`), not a constant (`let`). If you try to assign to an enum instance property through a `let` reference to the enum, you'll get a compile error.

For example, here's a silly enum:

```
enum Silly {
    case one
    var sillyProperty : String {
        get { "Howdy" }
        set {} // do nothing
    }
}
```

It is then legal to say this:

```
var silly = Silly.one
silly.sillyProperty = "silly"
```

But if `silly` were declared with `let` instead of `var`, trying to set `silly.silly-Property` would cause a compile error.

An enum static property can have a property wrapper, but an enum instance property can't, because that would imply storage of an instance of the underlying `@propertyWrapper` type — and enums have no stored instance properties.

Enum Methods

An enum can have instance methods (including subscripts) and static methods. Writing an enum method is straightforward. Here's an example from my own code. In a card game, the cards draw themselves as rectangles, ellipses, or diamonds. I've abstracted the drawing code into an enum that draws itself as a rectangle, an ellipse, or a diamond, depending on its case:

```
enum Shape {
    case rectangle
    case ellipse
    case diamond
    func addShape (to p: CGMutablePath, in r: CGRect) -> () {
        switch self {
        case .rectangle:
            p.addRect(r)
        case .ellipse:
            p.addEllipse(in:r)
        case .diamond:
            p.move(to: CGPoint(x:r.minX, y:r.midY))
            p.addLine(to: CGPoint(x: r.midX, y: r.minY))
            p.addLine(to: CGPoint(x: r.maxX, y: r.midY))
            p.addLine(to: CGPoint(x: r.midX, y: r.maxY))
            p.closeSubpath()
        }
    }
}
```

Earlier, I mentioned that a subscript can be a static method. That gives me an idea for yet another way to make a Filter instance by number:

```
enum Filter : String, CaseIterable {
    case albums = "Albums"
    case playlists = "Playlists"
    case podcasts = "Podcasts"
    case books = "Audiobooks"
    static subscript(ix: Int) -> Filter {
        Filter.allCases[ix] // warning, no range checking
    }
}
```

And now we can say:

```
let type = Filter[2] // podcasts
```

An enum instance method that modifies the enum itself must be marked as `mutating`. For example, an enum instance method might assign to an instance property of `self`; even though this is a computed property, such assignment is illegal unless the method is marked as `mutating`. The caller of a mutating instance method must have a variable reference to the instance (`var`), not a constant reference (`let`).

A mutating enum instance method can replace this instance with another instance, by assigning another case to `self`. In this example, I add an `advance` method to my Filter enum. The idea is that the cases constitute a sequence, and the sequence can cycle. By calling `advance`, I transform a Filter instance into an instance of the next case in the sequence:

```
enum Filter : String, CaseIterable {
    case albums = "Albums"
    case playlists = "Playlists"
    case podcasts = "Podcasts"
    case books = "Audiobooks"
    mutating func advance() {
        let cases = Filter.allCases
        var ix = cases.firstIndex(of:self)!
```

```
            ix = (ix + 1) % cases.count
            self = cases[ix]
        }
    }
```

And here's how to call it:

```
    var type = Filter.books
    type.advance() // type is now Filter.albums
```

Observe that `type` is declared with `var`; if it were declared with `let`, we'd get a compile error.

A subscript or computed property setter is considered mutating by default and does not have to be specially marked. However, if a computed property getter sets another property as a side effect, it must be marked `mutating get`.

Why Enums?

An enum is a switch whose states have names. There are many situations where that's a desirable thing. You could implement a multistate value yourself; if there are five possible states, you could use an Int whose values can be 0 through 4. But then you would have to provide a lot of additional overhead, interpreting those numeric values correctly and making sure that no other values are used. A list of five named cases is much better!

Even when there are only *two* states, an enum is often better than, say, a mere Bool, because the enum's states have names. With a Bool, you have to know what `true` and `false` signify in a particular usage; with an enum, the name of the enum and the names of its cases *tell* you its significance. Moreover, you can store extra information in an enum's associated value or raw value.

In my LinkSame app, the user can play a real game with a timer or a practice game without a timer. At various places in the code, I need to know which type of game this is. The game types are the cases of an enum:

```
    enum InterfaceMode : Int {
        case timed = 0
        case practice = 1
    }
```

The current game type is stored in an instance property `interfaceMode`, whose value is an InterfaceMode. It's easy to set the game type by case name:

```
    // ... initialize new game ...
    self.interfaceMode = .timed
```

And it's easy to examine the game type by case name:

```
    // notify of high score only if user is not just practicing
    if self.interfaceMode == .timed { // ...
```

And what are my InterfaceMode enum's raw value integers for? That's the really clever part. They correspond to the segment indexes of a UISegmentedControl in the interface! Whenever I change the `interfaceMode` property, a setter observer also selects the corresponding segment of the UISegmentedControl (`self.timedPractice`), simply by fetching the `rawValue` of the current enum case:

```
var interfaceMode : InterfaceMode = .timed {
    willSet (mode) {
        self.timedPractice?.selectedSegmentIndex = mode.rawValue
    }
}
```

Structs

A *struct* is the Swift object type *par excellence*. An enum, with its fixed set of cases, is a reduced, specialized kind of object. A class, at the other extreme, will often turn out to be overkill; it has some features that a struct lacks (I'll talk later about what they are), but if you don't need those features, a struct may be preferable.

Of more than two hundred object types declared in the Swift header, maybe half a dozen are classes. A couple of dozen are enums. All the rest are structs. A String is a struct. An Int is a struct. A Range is a struct. An Array is a struct. And so on. That shows how powerful a struct can be.

Struct Initializers

A struct that doesn't have an explicit initializer and that doesn't *need* an explicit initializer — because it has no stored properties, or because all its stored properties are assigned default values as part of their declaration — automatically gets an implicit initializer with no parameters, `init()`. For example:

```
struct Digit {
    var number = 42
}
```

That struct can be initialized by saying `Digit()`. But if you add any explicit initializers of your own, you lose that implicit initializer:

```
struct Digit {
    var number = 42
    init(number:Int) {
        self.number = number
    }
}
```

Now you can say `Digit(number:42)`, but you can't say `Digit()` any longer. Of course, you can add an explicit initializer that does the same thing:

```
struct Digit {
    var number = 42
    init() {}
    init(number:Int) {
        self.number = number
    }
}
```

Now you can say `Digit()` once again, as well as `Digit(number:42)`.

A struct that has stored properties and that doesn't have an explicit initializer automatically gets an implicit initializer derived from its instance properties. This is called the *memberwise initializer*. For example:

```
struct Test {
    var number = 42
    var name : String
    let age : Int
    let greeting = "Hello"
}
```

That struct is legal, even though it seems we have not fulfilled the contract requiring us to initialize all stored properties in their declaration or in an initializer. The reason is that this struct automatically has a memberwise initializer which *does* initialize all its properties. Given that declaration, there are two ways to make a Test instance:

```
let t1 = Test(number: 42, name: "matt", age: 67)
let t2 = Test(name: "matt", age: 67)
```

The memberwise initializer includes `number`, `name`, and `age`, but not `greeting`, because `greeting` is a `let` property that has already been initialized before the initializer is called. `number` has been initialized too, but it is a `var` property, so the memberwise initializer includes it — but you can also omit it from your call to the memberwise initializer, because it has been initialized already and therefore has a default value.

But if you add any explicit initializers of your own, or if any of the properties involved are declared `private`, you lose the implicit memberwise initializer (though of course you can write an explicit initializer that does the same thing).

All stored properties must be initialized either by direct initialization in the declaration or by all initializers. If a struct has multiple explicit initializers, they can delegate to one another by saying `self.init(...)`.

Struct Properties

A struct can have instance properties and static properties, which can be stored or computed variables. If other code wants to set a property of a struct instance, its reference to that instance must be a variable (`var`), not a constant (`let`).

Here's a Digit struct with a `var number` instance property:

```
struct Digit {
    var number : Int
    init(_ n:Int) {
        self.number = n
    }
}
```

Then this is legal:

```
var d = Digit(123)
d.number = 42
```

But if `d` were declared with `let`, trying to set `d.number` would cause a compile error.

Struct Methods

A struct can have instance methods and static methods, including subscripts. If an instance method sets a property, it must be marked as `mutating`, and the caller's reference to the struct instance must be a variable (`var`), not a constant (`let`).

Here's a new version of our Digit struct:

```
struct Digit {
    private var number : Int
    init(_ n:Int) {
        self.number = n
    }
    mutating func changeNumberTo(_ n:Int) {
        self.number = n
        // or: self = Digit(n)
    }
}
```

Here we have a private `number` property, along with a public method for setting it. We can then say this:

```
var d = Digit(123)
d.changeNumberTo(42)
```

The `changeNumberTo` method must be declared `mutating`, and if `d` were declared with `let`, trying to call `d.changeNumberTo` would cause a compile error.

Fetching the value of a `lazy` instance property may cause the property's initializer to run, setting the value of the property. That's a mutation! Therefore, an instance method must be declared `mutating` if it fetches the value of a `lazy` instance property:

Constant Namespaces

I often use a degenerate struct or enum as a handy namespace for constants. I call such a type “degenerate” because it consists entirely of static members; I don’t intend to use this object type to make any instances. (A reason for preferring an enum over a struct is that a caseless enum *cannot* be instantiated.)

Let’s say I’m storing user preference information in Cocoa’s UserDefaults. UserDefaults is a kind of dictionary: each item is accessed through a key. The keys are typically strings. A common programmer mistake is to write out these string keys literally every time a key is used; if you misspell a key name, there’s no penalty at compile time, but your code will mysteriously fail to work correctly. A better approach is to embody those keys as constant strings with names; if you make a mistake typing a name, the compiler can catch you. An enum with static members is a great way to define constant strings and clump their names into a namespace. Here’s a real-life example:

```
enum Default {
    static let rows = "CardMatrixRows"
    static let columns = "CardMatrixColumns"
    static let hazyStripy = "HazyStripy"
}
```

I can now refer to a UserDefaults key with a name, such as `Default.hazyStripy`.

```
struct Person {
    private lazy var name = "matt"
    mutating func getName() -> String {
        return self.name
    }
}
```

A subscript, setter observer, or computed property setter is considered mutating by default and does not have to be specially marked. However, if a computed property getter sets another property as a side effect, it must be marked `mutating get`.

A mutating instance method can replace this instance with another instance, by setting `self` to a different instance of the same struct.

Classes

A *class* is similar to a struct, with the following key differences:

Reference type
: Classes are reference types. This means, among other things, that a class instance has two remarkable features that are not true of struct or enum instances:

Mutability

A class instance is mutable in place. Even if your reference to an instance of a class is a constant (`let`), you can change the value of an instance property through that reference. An instance method of a class never has to be marked `mutating` (and cannot be).

Multiple references

When a given instance of a class is assigned to multiple variables or passed as argument to a function, you get multiple references to *one and the same object*.

Inheritance

A class can have a superclass. A class that has a superclass is a *subclass* of that superclass, and inherits its superclass's members. Class types can form a hierarchical tree.

In Objective-C, classes are the only object type. Some built-in Swift struct types are magically bridged to Objective-C class types, but your custom struct types don't have that magic. Thus, when programming iOS with Swift, one reason for declaring a class, rather than a struct, is as a form of interchange with Objective-C and Cocoa.

Value Types and Reference Types

A major difference between enums and structs, on the one hand, and classes, on the other, is that enums and structs are *value types*, whereas classes are *reference types*. I will now explain what that means.

Class instances are mutable

A value type is *not mutable in place*, even though it seems to be. A struct is a value type. Here is a Digit struct:

```
struct Digit {
    var number : Int
    init(_ n:Int) {
        self.number = n
    }
}
```

Now, Swift's syntax of assignment would lead us to believe that changing a Digit's `number` is possible:

```
var d = Digit(123)
d.number = 42
```

But in reality, when you apparently mutate an instance of a value type, you are actually *replacing* that instance with a *different* instance. To see that this is true, add a setter observer:

```
var d : Digit = Digit(123) { // Digit is a struct
    didSet {
        print("d was set")
    }
}
d.number = 42 // "d was set"
```

That explains why it is impossible to mutate a value type instance if the reference to that instance is declared with `let`:

```
let d = Digit(123) // Digit is a struct
d.number = 42 // compile error
```

Under the hood, this change would require us to *replace* the Digit instance pointed to by `d` with another Digit instance — and we can't do that, because it would mean assigning into `d`, which is exactly what the `let` declaration forbids.

That also explains why an instance method of a struct or enum that sets a property of the instance must be marked explicitly with the `mutating` keyword. Such a method can potentially replace this object with another, so the reference to the object must be `var`, not `let`.

But classes are *not* value types. They are reference types. A reference to a class instance does *not* have to be declared with `var` in order to set a `var` property through that reference:

```
class Dog {
    var name : String = "Fido"
}
let rover = Dog()
rover.name = "Rover" // fine
```

In the last line of that code, the class instance pointed to by `rover` is being *mutated in place*. No implicit assignment to `rover` is involved, and so the `let` declaration is powerless to prevent the mutation. A setter observer on a Dog variable is *not* called when a property is set:

```
var rover : Dog = Dog() { // Dog is a class
    didSet {
        print("did set rover")
    }
}
rover.name = "Rover" // nothing in console
```

The setter observer would be called if we were to set `rover` explicitly (to another Dog instance), but it is not called merely because we change a property of the Dog instance already pointed to by `rover`.

Exactly the same difference between a value type and a reference type may be seen with a parameter of a function call. When we receive an instance of a value type as a

Mutating Captured Self

Here's a Digit struct with some `mutating` methods:

```
struct Digit {
    var number : Int
    init(_ n:Int) {
        self.number = n
    }
    mutating func changeNumberTo(_ n:Int) {
        self.number = n
    }
    func otherFunction(_ f: ()->()) {
    }
    mutating func callAnotherFunction() {
        otherFunction {
            self.changeNumberTo(345) // *
        }
    }
}
```

Whether that's legal depends on whether `otherFunction` declares its function parameter `@escaping` ("Escaping Closures" on page 59). If it does, the compiler will stop us:

```
func otherFunction(_ f: @escaping ()->()) {
}
```

That change causes a compile error at the starred line: "Escaping closure captures mutating `self` parameter." Now that `otherFunction` is escaping, we are threatening to mutate a persisting captured `self` *at some later time*. Digit is a struct, so that would involve *replacing* the captured `self` with a different Digit — and that's incoherent. No such problem arises if Digit is a class, because the persistent captured `self` can be mutated in place.

parameter into a function body, the compiler will stop us in our tracks if we try to assign to its instance property. This doesn't compile:

```
func digitChanger(_ d:Digit) { // Digit is a struct
    d.number = 42 // compile error
}
```

But this does compile:

```
func dogChanger(_ d:Dog) { // Dog is a class
    d.name = "Rover"
}
```

Class instance references are pointers

With a reference type, there is a concealed level of indirection between your reference to the instance and the instance itself; the reference actually holds a *pointer* to the instance. This means that when a class instance is assigned to a variable or passed as an argument to a function or as the result of a function, you can wind up with *multiple references to the same object*. That is not true of structs and enums:

Struct or enum instance (value type)
: When an enum instance or a struct instance is assigned or passed, what is assigned or passed is essentially a *new copy* of that instance.

Class instance (reference type)
: When a class instance is assigned or passed, what is assigned or passed is a reference to the *same* instance.

To prove it, I'll assign one reference to another, and then mutate the second reference — and then I'll examine what happened to the first reference. Let's start with the struct:

```
var d = Digit(123) // Digit is a struct
print(d.number) // 123
var d2 = d // assignment!
d2.number = 42
print(d.number) // 123
```

In that code, we changed the `number` property of `d2`, a struct instance; but nothing happened to the `number` property of `d`. Now let's try the class:

```
var fido = Dog() // Dog is a class
print(fido.name) // Fido
var rover = fido // assignment!
rover.name = "Rover"
print(fido.name) // Rover
```

In that code, we changed the `name` property of `rover`, a class instance — and the `name` property of `fido` was changed as well! That's because, after the assignment in the third line, `fido` and `rover` refer to *one and the same instance*.

The same thing is true of parameter passing. With a class instance, what is passed is a reference to the *same* instance:

```
func dogChanger(_ d:Dog) { // Dog is a class
    d.name = "Rover"
}
var fido = Dog()
print(fido.name) // "Fido"
dogChanger(fido)
print(fido.name) // "Rover"
```

The change made to `d` inside the function `dogChanger` affected *our* Dog instance `fido`! You can't do that with an enum or struct instance parameter — unless it's an `inout` parameter — because the instance is effectively *copied* as it is passed. But handing a class instance to a function does *not* copy that instance; it is more like *lending* that instance to the function.

Advantages of value types vs. reference types

The ability to generate multiple references to the same instance is significant particularly in a world of object-based programming, where objects persist and can have properties that persist along with them. If object A and object B are both long-lived, and if they both have a Dog property where Dog is a class, and if they have each been handed a reference to one and the same Dog instance, then either object A or object B can mutate its Dog, and this mutation will affect the other's Dog. You can be holding on to an object, only to discover that it has been mutated by someone else behind your back. If that happens unexpectedly, it can put your program into an invalid state.

Class instances are also more complicated behind the scenes. Swift has to manage their memory (as I'll explain in detail in Chapter 13), precisely because there can be multiple references to the same object; this management can involve quite a bit of overhead.

On the whole, therefore, you should prefer a value type (such as a struct) to a reference type (a class) wherever possible. Struct instances are not shared between references, and so you are relieved from any worry about such an instance being mutated behind your back; moreover, under the hood, storage and memory management are far simpler as well. Apple likes to say that value types are *easier to reason about*. The Swift language itself will help you by imposing value types in front of many Cocoa Foundation reference types. Objective-C NSDate and NSData are classes, but Swift will steer you toward using struct types Date and Data instead (Chapter 11).

But don't get the wrong idea. Classes are not bad; they're good! For one thing, a class instance is very efficient to pass around, because all you're passing is a pointer. No matter how big and complicated a class instance may be, no matter how many properties it may have containing vast amounts of data, passing the instance is incredibly fast and efficient.

And although a class may be a reference type, a *particular* class can be implemented in such a way as to exhibit value *semantics*. Simply put, a class's API can refuse to mutate that class in place. Cocoa NSString, NSArray, NSDictionary, NSDate, NSIndexSet, NSParagraphStyle, and many more behave like this; they are *immutable* by design. Two objects may hold a reference to the same NSArray without fear that it will be mutated behind their backs, not because it's a value type (it isn't) but because it's immutable. In effect, this architecture combines the ease of use of a value type with the pointer efficiency of a reference type.

Moreover, there are many situations where the independent identity of a class instance, no matter how many times it is referred to, is exactly what you want. The extended lifetime of a class instance, as it is passed around, can be crucial to its functionality and integrity. In particular, only a class instance can successfully represent an *independent reality*. UIView is a class so that an individual UIView instance keeps representing the same single real and persistent view in your running app's interface.

Still another reason for preferring a class over a struct or enum is when you need recursive references. A value type cannot be structurally recursive: a stored instance property of a value type cannot be an instance of the same type. This code won't compile:

```
struct Dog { // compile error
    var puppy : Dog?
}
```

(Oddly, you can work around this restriction by typing `puppy` as an *array* of Dog.) More complex circular chains, such as a Dog with a Puppy property and a Puppy with a Dog property, are similarly illegal. But if Dog is a class instead of a struct, there's no error. This is a consequence of the nature of memory management of value types as opposed to reference types.

An enum case's associated value *can* be an instance of that enum, provided the case (or the entire enum) is marked `indirect`:

```
enum Node {
    case none(Int)
    indirect case left(Int, Node)
    indirect case right(Int, Node)
    indirect case both(Int, Node, Node)
}
```

Subclass and Superclass

Two classes can be *subclass* and *superclass* of one another. For example, we might have a class Quadruped and a class Dog, with Quadruped as the superclass of Dog. A class may have many subclasses, but a class can have only one immediate superclass. I say "immediate" because that superclass might itself have a superclass, and so on until we get to the ultimate superclass, called the *base class*, or *root class*. Thus there is a hierarchical tree of subclasses, each group of subclasses branching from its superclass, and so on, with a single class, the base class, at the top.

As far as the Swift language itself is concerned, there is no requirement that a class should have any superclass, or, if it does have a superclass, that it should ultimately be descended from any particular base class. A Swift program can have many classes

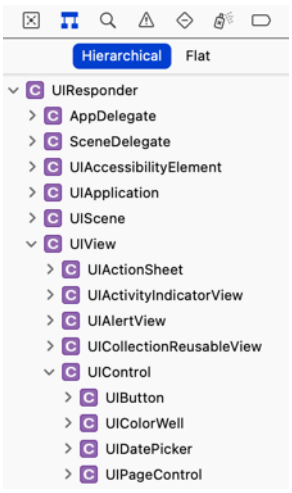

Figure 4-1. Part of the Cocoa class hierarchy as shown in Xcode

that have no superclass, and it can have many independent hierarchical subclass trees, each descended from a different base class.

Cocoa, however, doesn't work that way. In Cocoa, there is effectively just one base class, NSObject, which embodies all the functionality necessary for a class to *be* a class in the first place — and all other classes are subclasses, at some level, of that one base class. Cocoa thus consists of one huge tree of hierarchically arranged classes, even before you write a single line of code or create any classes of your own.

We can imagine diagramming this tree as an outline. And in fact Xcode will *show* you this outline (Figure 4-1): in an iOS project window, choose View → Navigators → Show Symbol Navigator and click Hierarchical, with the first and third icons in the filter bar selected (filled). Now locate NSObject in the list; the Cocoa classes are the part of the tree descending from it.

Inheritance

The reason for having a superclass–subclass relationship in the first place is to allow related classes to *share functionality*. Suppose we have a Dog class and a Cat class, and we are considering declaring a `walk` method for both of them. We might reason that both a dog and a cat walk in pretty much the same way, by virtue of both being quadrupeds. So it might make sense to declare `walk` as a method of the Quadruped class, and make both Dog and Cat subclasses of Quadruped. When we do that, both Dog and Cat can be sent the `walk` message, even if neither of them has a `walk` method, because each of them has a superclass that *does* have a `walk` method. We say that a subclass *inherits* the methods of its superclass.

To declare that a certain class is a subclass of a certain superclass, add a colon and the superclass name after the class's name in its declaration:

```
class Quadruped {
    func walk () {
        print("walk walk walk")
    }
}
class Dog : Quadruped {}
class Cat : Quadruped {}
```

Now let's prove that Dog has indeed inherited `walk` from Quadruped:

```
let fido = Dog()
fido.walk() // walk walk walk
```

The `walk` message can be sent to a Dog instance just as if the `walk` instance method were declared in the Dog class, even though the `walk` instance method is in fact declared in a superclass of Dog. That's inheritance at work.

A class declaration can *prevent* the class from being subclassed by preceding the class declaration with the `final` keyword.

Additional functionality

The purpose of subclassing is not *merely* so that a class can inherit another class's methods; it's so that it can also declare methods *of its own*. Typically, a subclass consists of the methods inherited from its superclass *and then some*. For example, dogs can bark, but quadrupeds in general can't. If we declare `bark` in the Dog class, and `walk` in the Quadruped class, and make Dog a subclass of Quadruped, then Dog inherits the ability to walk from the Quadruped class *and also* knows how to bark:

```
class Quadruped {
    func walk () {
        print("walk walk walk")
    }
}
class Dog : Quadruped {
    func bark () {
        print("woof")
    }
}
```

Again, let's prove that it works:

```
let fido = Dog()
fido.walk() // walk walk walk
fido.bark() // woof
```

Within a class, it is a matter of indifference whether that class has an instance method because that method is declared in that class or because the method is declared in a superclass and inherited. A message to `self` works equally well either way. In this code, we have declared a `barkAndWalk` instance method that sends two messages to `self`, without regard to where the corresponding methods are declared (one is native to the subclass, one is inherited from the superclass):

```
class Quadruped {
    func walk () {
        print("walk walk walk")
    }
}
class Dog : Quadruped {
    func bark () {
        print("woof")
    }
    func barkAndWalk() {
        self.bark()
        self.walk()
    }
}
```

And here's proof that it works:

```
let fido = Dog()
fido.barkAndWalk() // woof walk walk walk
```

Overriding

It is also permitted for a subclass to *redefine* a method inherited from its superclass. For example, perhaps some dogs bark differently from other dogs. We might have a class NoisyDog, for instance, that is a subclass of Dog. Dog declares `bark`, but NoisyDog also declares `bark`, and defines it differently from how Dog defines it. This is called *overriding*. The very natural rule is that if a subclass overrides a method inherited from its superclass, then when the corresponding message is sent to an instance of that subclass, it is the subclass's version of that method that is called.

In Swift, when you override something inherited from a superclass, you must explicitly acknowledge this fact by preceding its declaration with the keyword `override`:

```
class Quadruped {
    func walk () {
        print("walk walk walk")
    }
}
class Dog : Quadruped {
    func bark () {
        print("woof")
    }
}
```

```
class NoisyDog : Dog {
    override func bark () {
        print("woof woof woof")
    }
}
```

And let's try it:

```
let fido = Dog()
fido.bark() // woof
let rover = NoisyDog()
rover.bark() // woof woof woof
```

Observe that a subclass method by the same *name* as a superclass's method is not necessarily, of itself, an override. Recall that Swift can distinguish two functions with the same name, provided they have different *signatures*. Those are different functions, and so an implementation of one in a subclass is not an override of the other in a superclass. An override situation exists only when the subclass redefines the *same* method that it inherits from a superclass — using the same name, including the external parameter names, and the same signature.

However, a method override need not have *exactly* the same signature as the overridden method. In particular, in a method override, the type of a parameter may be replaced with a superclass, or with an Optional wrapping the superclass. If we have a Cat class and its Kitten subclass, the following is legal:

```
class Dog {
    func barkAt(cat:Kitten) {}
}
class NoisyDog : Dog {
    override func barkAt(cat:Cat) {}
    // or: override func barkAt(cat:Cat?)
}
```

Moreover, a parameter may be overridden with an Optional wrapping its own type, and an Optional parameter may be overridden with an Optional wrapping its wrapped type's superclass:

```
class Dog {
    func barkAt(cat:Cat) {}
    // or: func barkAt(cat:Kitten)
    // or: func barkAt(cat:Kitten?)
}
class NoisyDog : Dog {
    override func barkAt(cat:Cat?) {}
}
```

There are further rules along the same lines, but I won't try to list them all here; you probably won't need to take advantage of them, and in any case the compiler will tell you if your override is illegal.

Along with methods, a subclass also inherits its superclass's properties. Naturally, the subclass may also declare additional properties of its own. It is possible to override an inherited property (with some restrictions that I'll talk about later).

I'll have more to say about the implications of overriding when I talk about polymorphism, later in this chapter.

A class declaration can *prevent* a class member from being overridden by a subclass by preceding the member's declaration with the `final` keyword.

The keyword super

It often happens that we want to override something in a subclass and yet access the thing overridden in the superclass. This is done by sending a message to the keyword `super`. Our `bark` implementation in NoisyDog is a case in point. What NoisyDog really does when it barks is the same thing Dog does when *it* barks, but more times. We'd like to express that relationship in our implementation of NoisyDog's `bark`. To do so, we have NoisyDog's `bark` implementation send the `bark` message, not to `self` (which would be circular), but to `super`; this causes the search for a `bark` instance method implementation to start in the superclass rather than in our own class:

```
class Dog : Quadruped {
    func bark () {
        print("woof")
    }
}
class NoisyDog : Dog {
    override func bark () {
        for _ in 1...3 {
            super.bark()
        }
    }
}
```

And it works:

```
let fido = Dog()
fido.bark() // woof
let rover = NoisyDog()
rover.bark() // woof woof woof
```

A subscript function is a method. If a superclass declares a subscript, the subclass can declare a subscript with the same signature, provided it designates it with the `override` keyword. To call the superclass subscript implementation, the subclass can use square brackets after the keyword `super` (e.g. `super[3]`).

Class Initializers

Initialization of a class instance is considerably more complicated than initialization of a struct or enum instance, because of class inheritance. The chief task of an initializer is to ensure that all properties have an initial value, making the instance well-formed as it comes into existence; and an initializer may have other tasks to perform that are essential to the initial state and integrity of this instance. A class, however, may have a superclass, which may have properties and initializers of its own. Thus we must somehow ensure that when a subclass is initialized, its *superclass's* properties are initialized and the tasks of *its* initializers are performed in good order, in addition to those of the subclass itself.

Swift solves this problem coherently and reliably — and ingeniously — by enforcing some clear and well-defined rules about what a class initializer must do.

Kinds of class initializer

The rules begin with a distinction between the kinds of initializer that a class can have:

Designated initializer

A class initializer, by default, is a *designated* initializer. A class can be instantiated only through a call to one of its designated initializers. A designated initializer must see to it that all stored properties are initialized. It may not delegate to another initializer in the same class; it is illegal for a designated initializer to use the phrase `self.init(...)`.

A class with any stored properties that are not initialized as part of their declaration must have at least one explicit designated initializer. A class with no stored properties, or with stored properties all of which are initialized as part of their declaration, and that has *no* explicit designated initializers, has an *implicit* designated initializer `init()`.

Convenience initializer

A *convenience* initializer is marked with the keyword `convenience`. A convenience initializer is not how a class is instantiated; it is merely a façade for a designated initializer. A convenience initializer is a delegating initializer; it *must* contain the phrase `self.init(...)`, which must call a *designated* initializer in the same class — or, if it calls another convenience initializer in the same class, the chain of convenience initializers must end by calling a designated initializer in the same class.

Here are some examples. This class has no stored properties, so it has an implicit `init()` designated initializer:

```
class Dog {
}
let d = Dog()
```

This class's stored properties have default values, so it has an implicit `init()` designated initializer too:

```
class Dog {
    var name = "Fido"
}
let d = Dog()
```

This class's stored properties have default values, but it has no implicit `init()` initializer because it has an explicit designated initializer:

```
class Dog {
    var name = "Fido"
    init(name:String) {self.name = name}
}
let d = Dog(name:"Rover") // ok
let d2 = Dog() // compile error
```

This class's stored properties have default values, and it has an explicit initializer, but it also has an implicit `init()` initializer because its explicit initializer is a convenience initializer. Moreover, the implicit `init()` initializer is a designated initializer, so the convenience initializer can delegate to it:

```
class Dog {
    var name = "Fido"
    convenience init(name:String) {
        self.init()
        self.name = name
    }
}
let d = Dog(name:"Rover")
let d2 = Dog()
```

This class has stored properties without default values; it has an explicit designated initializer, and all of those properties are initialized in that designated initializer:

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
let d = Dog(name:"Rover", license:42)
```

This class is similar to the previous example, but it also has convenience initializers forming a chain that ends with a designated initializer:

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    convenience init(license:Int) {
        self.init(name:"Fido", license:license)
    }
    convenience init() {
        self.init(license:1)
    }
}
let d = Dog()
```

Note that the rules about what else an initializer can say and when it can say it, as I described them earlier in this chapter, are still in force:

- A designated initializer cannot, except in order to initialize a property (or to fetch the value of a property that is already initialized), say `self`, implicitly or explicitly, until *all* of this class's properties have been initialized.
- A convenience initializer is a delegating initializer, so it cannot say `self` for *any* purpose until after it has called, directly or indirectly, a designated initializer (and cannot set a constant property at all).

Subclass initializers

Having defined and distinguished between designated initializers and convenience initializers, we are ready for the rules about a subclass's initializers:

No declared initializers
: If a subclass doesn't have to have any initializers of its own, and if it declares no initializers of its own, then its initializers consist of the initializers inherited from its superclass. (A subclass thus has no implicit `init()` initializer unless it inherits it from its superclass.)

Convenience initializers only
: If a subclass doesn't have to have any initializers of its own, it is eligible to declare convenience initializers, and these work exactly as convenience initializers always do, because inheritance supplies the designated initializers that the convenience initializers must call by saying `self.init(...)`.

Designated initializers
: If a subclass declares any designated initializers of its own, the entire game changes drastically. Now, *no initializers are inherited!* The existence of an explicit designated initializer *blocks initializer inheritance*. The only initializers the sub-class now has are the initializers that you explicitly write (with one exception that

I'll mention later). This rule may seem surprising, but I'll justify it in an example later on.

Moreover, every designated initializer in the subclass now has an extra requirement: it must call one of the *superclass's designated initializers*, by saying `super.init(...)`. If it fails to do this, then `super.init()` is called implicitly if possible, but I disapprove of this feature (in my view, Swift should not indulge in secret behavior, even if that behavior might be considered "helpful").

At the same time, the rules about saying `self` continue to apply.

Thus, a subclass designated initializer must do these things *in this order:*

1. It must ensure that all properties of *this* class (the subclass) are initialized.
2. It must call `super.init(...)`, and the initializer that it calls must be a designated initializer.
3. Only then may this initializer say `self` for such purposes as to call an instance method or to access an inherited property.

Designated and convenience initializers

If a subclass declares both designated and convenience initializers, the convenience initializers in the subclass are still subject to the rules I've already outlined. They must call `self.init(...)`, calling a designated initializer directly or through a chain of convenience initializers. There are no inherited initializers, so the designated initializer must be explicitly declared in the subclass.

Override initializers

A subclass may override initializers from its superclass:

- An initializer whose parameters match a *convenience* initializer of the superclass can be a designated initializer or a convenience initializer, and is *not* marked `override`.
- An initializer whose parameters match a *designated* initializer of the superclass can be a designated initializer or a convenience initializer, and *must* be marked `override`. An `override` designated initializer must still call some superclass designated initializer (possibly even the one that it overrides) with `super.init(...)`.

If a subclass overrides *all* of its superclass's *designated* initializers, then the subclass inherits the superclass's *convenience* initializers. (This is the exception to the rule that if a subclass has any designated initializers, no initializers are inherited.)

Failable initializers

If an initializer called by a failable initializer is failable, the calling syntax does not change, and no additional test is needed — if a failable initializer fails, the whole initialization process will fail (and will be aborted) immediately.

There are some additional restrictions on failable initializers:

- `init` can override `init?`, but not vice versa.
- `init?` can call `init`.
- `init` can call `init?` by saying `init` and unwrapping the result with an exclamation mark (and if the `init?` fails, you'll crash).

At no time can a subclass initializer set a constant (`let`) property of a superclass. This is because, by the time the subclass is allowed to do anything other than initialize its own properties and call another initializer, the superclass has finished its own initialization and the door for initializing its constants has closed.

Subclass initializer examples

Your eyes may glaze over reading the subclass initializer rules, but the most important rules are very easy to understand with the help of some basic examples. We start with a subclass that has no explicit initializers of its own:

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    convenience init(license:Int) {
        self.init(name:"Fido", license:license)
    }
}
class NoisyDog : Dog {
}
```

Given that code, you can make a NoisyDog like this:

```
let nd1 = NoisyDog(name:"Fido", license:1)
let nd2 = NoisyDog(license:2)
```

That code is legal, because NoisyDog inherits its superclass's initializers. However, you can't make a NoisyDog like this:

```
let nd3 = NoisyDog() // compile error
```

That code is illegal. Even though a NoisyDog has no properties of its own, it has no implicit `init()` initializer; its initializers are its inherited initializers, and its superclass, Dog, has no implicit `init()` initializer to inherit.

Now here is a subclass whose only explicit initializer is a convenience initializer:

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    convenience init(license:Int) {
        self.init(name:"Fido", license:license)
    }
}
class NoisyDog : Dog {
    convenience init(name:String) {
        self.init(name:name, license:1)
    }
}
```

Observe how NoisyDog's convenience initializer fulfills its contract by calling `self.init(...)` to call a designated initializer — which it happens to have inherited. Given that code, there are three ways to make a NoisyDog, just as you would expect:

```
let nd1 = NoisyDog(name:"Fido", license:1)
let nd2 = NoisyDog(license:2)
let nd3 = NoisyDog(name:"Rover")
```

Next, here is a subclass that declares a designated initializer:

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    convenience init(license:Int) {
        self.init(name:"Fido", license:license)
    }
}
class NoisyDog : Dog {
    init(name:String) {
        super.init(name:name, license:1)
    }
}
```

NoisyDog's explicit initializer is now a designated initializer. It fulfills its contract by calling a designated initializer in `super`. NoisyDog has now *cut off inheritance* of all initializers; the *only* way to make a NoisyDog is like this:

```
let nd1 = NoisyDog(name:"Rover")
```

Earlier, I promised to justify the rule that adding a designated initializer to a subclass cuts off initializer inheritance. That example is a case in point. It would be terrible if the caller could bypass NoisyDog's designated initializer by using an inherited Dog initializer instead. NoisyDog's initializer enforces a rule that a NoisyDog can only have a `license` value of `1`; if you could say `NoisyDog(license:2)`, you'd bypass that rule.

Here's another example that makes the same point a little more realistically:

```
class Dog {
    let name : String
    init(name:String) {
        self.name = name
    }
}
class RoverDog : Dog {
    init() {
        super.init(name:"Rover")
    }
}
let fido = RoverDog(name:"Fido") // compile error
```

Clearly that last line *needs* to be an error; otherwise, a RoverDog could be named Fido, undermining the point of the subclass.

Finally, here is a subclass that overrides its designated initializers:

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    convenience init(license:Int) {
        self.init(name:"Fido", license:license)
    }
}
class NoisyDog : Dog {
    override init(name:String, license:Int) {
        super.init(name:name, license:license)
    }
}
```

NoisyDog has overridden *all* of its superclass's designated initializers, so it inherits its superclass's convenience initializers. There are thus two ways to make a NoisyDog:

```
let nd1 = NoisyDog(name:"Rover", license:1)
let nd2 = NoisyDog(license:2)
```

Those examples illustrate the main rules that you should keep in your head. You probably don't need to memorize the remaining rules, because the compiler will enforce them, and will keep slapping you down until you get them right.

Required initializers

A class initializer may be preceded by the keyword `required`. This means that a subclass may not lack this initializer. This, in turn, means that if a subclass implements designated initializers, thus blocking inheritance, it *must* override this initializer and mark the override `required`. Here's a (rather pointless) example:

```
class Dog {
    var name : String
    required init(name:String) {
        self.name = name
    }
}
class NoisyDog : Dog {
    var obedient = false
    init(obedient:Bool) {
        self.obedient = obedient
        super.init(name:"Fido")
    }
} // compile error
```

That code won't compile. Dog's `init(name:)` is marked `required`; our code won't compile unless we inherit or override `init(name:)` in NoisyDog. But we cannot inherit it, because, by implementing the NoisyDog designated initializer `init(obedient:)`, we have blocked inheritance. Therefore we must override it:

```
class Dog {
    var name : String
    required init(name:String) {
        self.name = name
    }
}
class NoisyDog : Dog {
    var obedient = false
    init(obedient:Bool) {
        self.obedient = obedient
        super.init(name:"Fido")
    }
    required init(name:String) {
        super.init(name:name)
    }
}
```

Observe that our overridden required initializer is not marked with `override`, but *is* marked with `required`, guaranteeing that the requirement continues drilling down to any further subclasses.

I have explained what declaring an initializer as `required` does, but I have not explained *why* you'd need to do it. That's another matter! I'll discuss it later in this chapter.

Class Deinitializer

A class can have a deinitializer. This is a function declared with the keyword `deinit` followed by curly braces containing the function body. You never call this function yourself; it is called by the runtime when an instance of this class goes out of existence. If a class has a superclass, the subclass's deinitializer (if any) is called before the superclass's deinitializer (if any).

A deinitializer is a class feature only; a struct or enum has no deinitializer. That's because a class is a reference type (as I explained earlier in this chapter). The idea is that you might want to perform some internal cleanup. Another good use of a class's `deinit` is to log to the console to prove to yourself that your instance is going out of existence in good order; I'll take advantage of that when I discuss memory management issues in Chapter 5.

Property observers are not called during `deinit`.

Class Properties

A subclass can override its inherited properties. The override must have the same name and type as the inherited property, and must be marked with `override`. (A property cannot have the same name as an inherited property but a different type, as there is no way to distinguish them.)

The chief restriction here is that an `override` property *cannot be a stored property*. More specifically:

- If the superclass property is writable (a stored property or a computed property with a setter), the subclass's override may consist of adding setter observers to this property.
- Alternatively, the subclass's override may be a computed property. In that case:
    - If the superclass property is stored, it must be writable and the subclass's computed property override must have both a getter and a setter.
    - If the superclass property is computed, the subclass's computed property override must have at least a getter, and:
        - If the superclass property has a setter, the override must have a setter.
        - If the superclass property has no setter, the override can add one.

The overriding property's functions may refer to — and may read from and write to — the inherited property, through the `super` keyword.

Static/Class Members

A class can have static members, marked `static`, just like a struct or an enum. It can also have class members, marked `class`. Both static and class members are inherited by subclasses.

Static methods vs. class methods

The chief difference between static and class methods, from the programmer's point of view, is that a static method *cannot be overridden;* it is as if `static` were a synonym for `class final`.

Here, I'll use a static method to express what dogs say:

```
class Dog {
    static func whatDogsSay() -> String {
        return "woof"
    }
    func bark() {
        print(Dog.whatDogsSay())
    }
}
```

A subclass now inherits `whatDogsSay`, but can't override it. No subclass of Dog may contain any implementation of a class method or a static method `whatDogsSay` with this same signature.

Now I'll use a class method to express what dogs say:

```
class Dog {
    class func whatDogsSay() -> String {
        return "woof"
    }
    func bark() {
        print(Dog.whatDogsSay())
    }
}
```

A subclass inherits `whatDogsSay`, and *can* override it, either as a class method or as a static method:

```
class NoisyDog : Dog {
    override class func whatDogsSay() -> String {
        return "WOOF"
    }
}
```

Static properties vs. class properties

The difference between static and class properties is similar to the difference between static and class methods, but with an additional, rather dramatic qualification: a static property can be stored, but a class property *must be a computed property*.

Here, I'll use a static class property to express what dogs say:

```
class Dog {
    static var whatDogsSay = "woof"
    func bark() {
        print(Dog.whatDogsSay)
    }
}
```

A subclass inherits `whatDogsSay`, but can't override it; no subclass of Dog can declare a class or static property `whatDogsSay`.

Now I'll use a class property to express what dogs say. It cannot be a stored property, so I'll have to use a computed property instead:

```
class Dog {
    class var whatDogsSay : String {
        return "woof"
    }
    func bark() {
        print(Dog.whatDogsSay)
    }
}
```

A subclass inherits `whatDogsSay` and can override it either as a class property or as a static property. But the rule about property overrides not being stored is still in force, even if the override is a static property:

```
class NoisyDog : Dog {
    override static var whatDogsSay : String {
        return "WOOF"
    }
}
```

Polymorphism

When a computer language has a hierarchy of types and subtypes, it must resolve the question of what such a hierarchy means for the relationship between the type of an *object* and the declared type of a *reference* to that object. Swift obeys the principles of *polymorphism*. In my view, it is polymorphism that turns an object-based language into a full-fledged object-oriented language. We may summarize Swift's polymorphism principles:

Substitution
: Wherever a certain type of object is expected, the actual object may be a subtype of that type.

Internal identity
: An object's real type is a matter of its internal nature, regardless of how that object is referred to.

To see what these principles mean in practice, imagine we have a Dog class, along with its subclass, NoisyDog:

```
class Dog {
}
class NoisyDog : Dog {
}
let d : Dog = NoisyDog()
```

In that code:

- The substitution rule says that the last line is legal: we can assign a NoisyDog instance to a reference, `d`, that is typed as Dog.
- The internal identity rule says that, under the hood, even though `d` is typed as Dog, the instance that it refers to is a NoisyDog.

You may be asking: How is the internal identity rule manifested? If a reference to a NoisyDog is typed as Dog, in what sense is this "really" a NoisyDog? To illustrate, let's examine what happens when a subclass overrides an inherited method. I'll redefine Dog and NoisyDog to demonstrate:

```
class Dog {
    func bark() {
        print("woof")
    }
}
class NoisyDog : Dog {
    override func bark() {
        for _ in 1...3 {
            super.bark()
        }
    }
}
```

Now consider the following code:

```
func tellToBark(_ d:Dog) {
    d.bark()
}
var nd = NoisyDog()
tellToBark(nd) // what will happen??????
```

That code is legal, because, by the substitution principle, we can pass `nd`, typed as NoisyDog, where a Dog is expected. Now, inside the `tellToBark` function, `d` is typed as Dog. How will it react to being told to `bark`? On the one hand, `d` is *typed* as Dog, and a Dog barks by saying `"woof"` once. On the other hand, in our code, when `tellToBark` is called, what is *really* passed is a NoisyDog instance, and a NoisyDog barks by saying `"woof"` three times. *What will happen?* Let's find out:

```
func tellToBark(_ d:Dog) {
    d.bark()
}
var nd = NoisyDog()
tellToBark(nd) // woof woof woof
```

The result is `"woof woof woof"`. The internal identity rule says that what matters when a message is sent is not how the recipient of that message is *typed* through this or that *reference*, but what that recipient actually *is*. What arrives inside `tellToBark` is a NoisyDog, regardless of the type of variable that holds it; thus, the `bark` message causes this object to say `"woof"` three times.

Here's another important consequence of polymorphism — the meaning of the keyword `self`. Its meaning depends upon the type of the actual instance — even if the word `self` *appears* in a superclass's code. For example:

```
class Dog {
    func bark() {
        print("woof")
    }
    func speak() {
        self.bark()
    }
}
class NoisyDog : Dog {
    override func bark() {
        for _ in 1...3 {
            super.bark()
        }
    }
}
```

What happens when we tell a NoisyDog to `speak`? The `speak` method is declared in Dog, the superclass — not in NoisyDog. The `speak` method calls the `bark` method. It does this by way of the keyword `self`. (I could have omitted the explicit reference to `self` here, but `self` would still be involved implicitly, so I'm not cheating by making `self` explicit.) There's a `bark` method in Dog, and an override of the `bark` method in NoisyDog. *Which `bark` method will be called?* Let's find out:

```
let nd = NoisyDog()
nd.speak() // woof woof woof
```

The word `self` is encountered within the Dog class's implementation of `speak`. But what matters is not where the word `self` *appears* but what it *means*. It means *this instance*. And the internal identity principle tells us that this instance is a NoisyDog! Thus, it is NoisyDog's override of `bark` that is called when Dog's `speak` says `self.bark()`.

Polymorphism applies to Optional types in the same way that it applies to the type of thing wrapped by the Optional. Suppose we have a reference typed as an Optional wrapping a Dog. You already know that you can assign a Dog to it. Well, you can also assign a NoisyDog, or an Optional wrapping a NoisyDog, and the underlying wrapped object will maintain its integrity:

```
var d : Dog?
d = Dog()
d = NoisyDog()
d = Optional(NoisyDog())
```

(The applicability of polymorphism to Optionals derives from a special dispensation of the Swift language: Optionals are *covariant*. I'll talk more about that later in this chapter.)

Thanks to polymorphism, you can take advantage of subclasses to add power and customization to existing classes. This is important particularly in the world of iOS programming, where most of the classes are defined by Cocoa and don't belong to you. The UIViewController class, for example, is defined by Cocoa; it has lots of built-in methods that Cocoa will call, and these methods perform various important tasks — but in a generic way. In real life, you'll make a UIViewController *subclass*, and you'll *override* those methods to do the tasks appropriate to your particular app. When you do that:

- It won't bother Cocoa in the slightest, because (substitution principle) wherever Cocoa expects to receive or to be talking to a UIViewController, it will accept without question an instance of your UIViewController subclass.
- The substituted UIViewController subclass will also work as expected, because (internal identity principle) whenever Cocoa calls one of those UIViewController methods on your subclass, it is your subclass's override that will be called, and wherever a UIViewController method refers to `self`, that will mean your subclass.

I'll talk more about subclassing Cocoa classes in Chapter 11.

Polymorphism is cool, but in the grand scheme of things it is also relatively slow. It requires *dynamic dispatch*, meaning that the compiler can't perform certain optimizations, and that the runtime has to think about what a message to a class instance means. You can reduce the need for dynamic dispatch by declaring a class or a class member `final` or `private`. Or use a struct, if appropriate; structs don't need dynamic dispatch.

Casting

Here's a conundrum. The Swift compiler, with its strict typing, imposes severe restrictions on what messages can be sent to an object reference. The messages that the compiler will permit to be sent to an object reference depend upon the reference's *declared* type. But the internal identity principle of polymorphism says that, under the hood, an object may have a *real* type that is different from its reference's declared type. Such an object may be *capable* of receiving certain messages, but the compiler *won't permit us* to send them.

To illustrate the problem, let's give NoisyDog a method that Dog doesn't have:

```
class Dog {
    func bark() {
        print("woof")
    }
}
class NoisyDog : Dog {
    override func bark() {
        super.bark(); super.bark()
    }
    func beQuiet() {
        self.bark()
    }
}
```

In that code, we configure a NoisyDog so that we can tell it to `beQuiet`. Now look at what happens when we try to tell an object typed as Dog to be quiet:

```
func tellToHush(_ d:Dog) {
    d.beQuiet() // compile error
}
let nd = NoisyDog()
tellToHush(nd)
```

Our code doesn't compile. We can't send the `beQuiet` message to the reference `d` inside the function body, because it is typed as Dog — and a Dog has no `beQuiet` method. But there is a certain irony here: for once, we happen to know more than the compiler does — namely, that this object is *in fact* a NoisyDog and *does* have a `beQuiet` method! Our code would run correctly — because `d` really is a NoisyDog — if only we could get our code to compile in the first place. We need a way to say to the compiler, “Look, compiler, just trust me: this thing is going to turn out to be a NoisyDog when the program actually runs, so let me send it this message.”

There is in fact a way to do this — *casting*. To cast, you use a form of the keyword `as` followed by the name of the type you claim something really is.

Casting Down

Swift will not let you cast just any old type to any old other type — you can’t cast a String to an Int — but it will let you cast a superclass to a subclass. This is called *casting down*. When you cast down, the form of the keyword `as` that you use is `as!` with an exclamation mark. The exclamation mark reminds you that you are *forcing* the compiler to do something it would rather not do:

```
func tellToHush(_ d:Dog) {
    (d as! NoisyDog).beQuiet()
}
let nd = NoisyDog()
tellToHush(nd)
```

That code compiles, and works. A useful way to rewrite the example is like this:

```
func tellToHush(_ d:Dog) {
    let d = d as! NoisyDog
    d.beQuiet()
    // ... other NoisyDog messages to d can go here ...
}
let nd = NoisyDog()
tellToHush(nd)
```

The reason that way of rewriting the code is useful is in case we have other NoisyDog messages to send to this object. Instead of casting every time we want to send a message to it, we cast the object once to its internal identity type, and assign it to a variable. Now that variable’s type — inferred, in this case, from the cast — is the internal identity type, and we can send multiple NoisyDog messages to the variable.

Type Testing and Casting Down Safely

A moment ago, I said that the `as!` operator’s exclamation mark reminds you that you are forcing the compiler’s hand. It also serves as a warning: your code can now crash! The reason is that you might be lying to the compiler. Casting down is a way of telling the compiler to relax its strict type checking and to let you call the shots. If you use casting to make a false claim, the compiler may permit it, but you will crash when the app runs:

```
func tellToHush(_ d:Dog) {
    let d = d as! NoisyDog // crash
    d.beQuiet()
}
let d = Dog()
tellToHush(d)
```

In that code, we told the compiler that this object would turn out to be a NoisyDog, and the compiler obediently took its hands off and allowed us to send the `beQuiet` message to it. But in fact, this object was a Dog when our code ran, and so we ultimately crashed when the cast failed because this object was *not* a NoisyDog.

To prevent yourself from lying accidentally, you can *test* the type of an instance at runtime. One way to do that is with the keyword `is`. You can use `is` in a condition; if the condition passes, *then* cast, in the knowledge that your cast is safe:

```
func tellToHush(_ d:Dog) {
    if d is NoisyDog {
        let d = d as! NoisyDog
        d.beQuiet()
    }
}
```

The result is that we won't cast `d` to a NoisyDog unless it really *is* a NoisyDog.

An alternative way to solve the same problem is to use Swift's `as?` operator. This casts down, but with the option of failure; therefore what it casts to is (you guessed it) an Optional — and now we are on familiar ground, because we know how to deal safely with an Optional:

```
func tellToHush(_ d:Dog) {
    let d = d as? NoisyDog // an Optional wrapping a NoisyDog
    if d != nil {
        d!.beQuiet()
    }
}
```

That doesn't look much cleaner or shorter than our previous approach. But remember that we can safely send a message to an Optional by optionally unwrapping the Optional:

```
func tellToHush(_ d:Dog) {
    let d = d as? NoisyDog // an Optional wrapping a NoisyDog
    d?.beQuiet()
}
```

Or, as a one-liner:

```
func tellToHush(_ d:Dog) {
    (d as? NoisyDog)?.beQuiet()
}
```

First we use the `as?` operator to obtain an Optional wrapping a NoisyDog. Then we optionally unwrap that Optional and send a message to it. If the original `d` wasn't a NoisyDog, the Optional will be `nil` and it won't be unwrapped and no message will be sent.

Type Testing and Casting Optionals

The `is`, `as!`, and `as?` operators work with Optionals in the same way that the equality comparison operators do (Chapter 3): they are automatically applied to the object wrapped by the Optional.

Let's start with `is`. Consider an Optional `d` ostensibly wrapping a Dog (that is, `d` is a `Dog?` object). It might, in actual fact, be wrapping either a Dog or a NoisyDog. To find out which it is, you might be tempted to use `is`. But can you? After all, an Optional is neither a Dog nor a NoisyDog — it's an Optional! Nevertheless, Swift knows what you mean; when the thing on the left side of `is` is an Optional, Swift pretends that it's the value wrapped in the Optional. This works just as you would hope:

```
let d : Dog? = NoisyDog()
if d is NoisyDog { // it is!
```

When using `is` with an Optional, the test fails in good order if the Optional is `nil`. Our test really does *two* things: it checks whether the Optional is `nil`, and if it is not, it then checks whether the wrapped value is the type we specify.

What about casting? You can't really cast an Optional to anything. Nevertheless, Swift knows what you mean; you can use the `as!` operator with an Optional. When the thing on the left side of `as!` is an Optional, Swift treats it as the wrapped type. Moreover, the consequence of applying the `as!` operator is that two things happen: Swift unwraps first, and then casts. This code works, because `d` is unwrapped to give us `d2`, which is a NoisyDog:

```
let d : Dog? = NoisyDog()
let d2 = d as! NoisyDog
d2.beQuiet()
```

That code, however, is not safe. You shouldn't cast like that, without testing first, unless you are very sure of your ground. If `d` were `nil`, you'd crash in the second line because you're trying to unwrap a `nil` Optional. And if `d` were a Dog, not a NoisyDog, you'd *still* crash in the second line when the cast fails. That's why there's also an `as?` operator, which *is* safe — but yields an Optional:

```
let d : Dog? = NoisyDog()
let d2 = d as? NoisyDog
d2?.beQuiet()
```

In that code, we use `as?` to cast down from an Optional wrapping a Dog (`d`) to an Optional wrapping a NoisyDog (`d2`). The operation is safe, twice. If `d` is `nil`, `d2` will be `nil`, safely. If `d` is not `nil` but wraps a Dog, not a NoisyDog, the cast will fail, and `d2` will be `nil`, safely. If `d` wraps a NoisyDog, `d2` wraps a NoisyDog. In the last line, we unwrap `d2` and send it a NoisyDog message — safely.

Bridging to Objective-C

Another way you'll use casting is during a value interchange between Swift and Objective-C when two types are *equivalent*. For example, you can cast a Swift String to a Cocoa NSString, and vice versa. That's not because one is a subclass of the other, but because they are *bridged* to one another; in a very real sense, they are the same type. When you cast from String to NSString, you're not casting down, and what you're doing is not dangerous, so you use the `as` operator, with no exclamation mark or question mark.

In general, to cross the bridge from a Swift type to a bridged Objective-C type, you will need to cast explicitly (except in the case of a string literal):

```
let s : NSString = "howdy"         // string literal to NSString
let s2 = "howdy"
let s3 : NSString = s2 as NSString // String to NSString
let i : NSNumber = 1 as NSNumber   // Int to NSNumber
```

That sort of code, however, is rather artificial. In real life, you won't be casting all that often, because the Cocoa API will present itself to you in terms of Swift types. This is legal with no cast:

```
let name = "MyNib" // Swift String
let vc = ViewController(nibName:name, bundle:nil)
```

The UIViewController class comes from Cocoa, and its `nibName` property is an Objective-C NSString — not a Swift String. But you don't have to help the Swift String `name` across the bridge by casting, because, in the Swift world, `nibName:` is typed as a Swift String (actually, an Optional wrapping a String). The bridge, in effect, is crossed *later*.

Similarly, no cast is required here:

```
let ud = UserDefaults.standard
let s = "howdy"
ud.set(s, forKey:"greeting")
```

You don't have to help the Swift String `s` across the bridge by casting, because the first argument of `set(_:forKey:)` is typed as a Swift type, namely Any (actually, an Optional wrapping Any) — and any Swift type can be used, without casting, where an Any is expected. I'll talk more about Any later in this chapter.

Coming back the other way, it is possible that you'll receive from Objective-C a value about whose real underlying type Swift has no information. In that case, you'll probably want to cast explicitly to the underlying type — and now you *are* casting down, with all that that implies. Here's what happens when we go to retrieve the "howdy" that we put into UserDefaults in the previous example:

```
let ud = UserDefaults.standard
let test = ud.object(forKey:"greeting") as! String
```

When we call `ud.object(forKey:)`, Swift has no type information; the result is an Any (actually, an Optional wrapping Any). But we know that this particular call should yield a string — because that's what we put in to begin with. So we can force-cast this value down to a String — and it works. However, if `ud.object(forKey:"greeting")` were *not* a string (or if it were `nil`), we'd crash. If you're not sure of your ground, use `is` or `as?` to be safe.

Type References

This section talks about the ways in which Swift can refer to the type of an object, other than saying the bare type literally.

From Instance to Type

Sometimes, what you've got is an instance, and you want to know its type. This might be for no other reason than to log its type to the console, for the sake of information or debugging; or you might need to use the type as a value, as I'll explain later.

For this purpose, you can use the global `type(of:)` function:

```
let d : Dog = NoisyDog()
print(type(of:d)) // NoisyDog
```

As you would expect, the identity principle applies. We are not asking how `d`, the variable, is typed; we're asking what sort of object the instance referred to by `d` *really* is. It's typed as Dog, but it's a NoisyDog instance.

From self to Type

It is particularly important for an instance to be able to refer to its *own* type. Quite commonly, this is in order to send a message to that type. For instance, suppose an instance wants to send a class message to its class. In an earlier example, a Dog instance method fetched a Dog class property by sending a message to the Dog type, literally using the word `Dog`:

```
class Dog {
    class var whatDogsSay : String {
        return "woof"
    }
    func bark() {
        print(Dog.whatDogsSay) // woof
    }
}
```

The expression `Dog.whatDogsSay` seems clumsy and inflexible. Why should we hard-code into Dog a knowledge of what type it is? It *has* a type; it should just *know* what it is. We can refer to the current type — the type of `self` — using the keyword `Self` (with a capital letter):

```
class Dog {
    class var whatDogsSay : String {
        return "woof"
    }
    func bark() {
        print(Self.whatDogsSay) // woof
    }
}
```

Similarly, we wrote a Filter enum earlier in this chapter that accessed its static `allCases` by saying `Filter.allCases`. We can say `Self.allCases` instead, and I prefer to do so; it's prettier.

Saying `Self` instead of a type name isn't *just* prettier; it's more powerful, because `Self`, like `self`, obeys polymorphism. Here are Dog and its subclass, NoisyDog:

```
class Dog {
    class var whatDogsSay : String {
        return "woof"
    }
    func bark() {
        print(Self.whatDogsSay)
    }
}
class NoisyDog : Dog {
    override class var whatDogsSay : String {
        return "woof woof woof"
    }
}
```

Now watch what happens:

```
let nd = NoisyDog()
nd.bark() // woof woof woof
```

If we tell a NoisyDog instance to `bark`, it says `"woof woof woof"`. The reason is that `Self` means, "The type that this object actually is, right now." We send the `bark` message to a NoisyDog instance. The `bark` implementation refers to `Self`; even though the bark implementation is inherited from Dog, Self means the type of this instance, which is a NoisyDog, and so Self is the NoisyDog class, and it is NoisyDog's version of whatDogsSay that is fetched.

Another important use of polymorphic Self is as a return type. To show why this is valuable, I'll introduce the notion of a *factory method*.

Suppose our Dog class has a name instance property, and its only initializer is init(name:). Let's give our Dog class a class method makeAndName. We want this class method to create and return a named Dog of whatever class we send the make-AndName message to. If we say Dog.makeAndName(), we should get a Dog. If we say NoisyDog.makeAndName(), we should get a NoisyDog. Well, we know how to do that; just initialize polymorphic Self. It works in a class method just as it works in an instance method:

```
class Dog {
    var name : String
    init(name:String) {
        self.name = name
    }
    class func makeAndName() -> Dog {
        let d = Self(name:"Fido") // compile error
        return d
    }
}
class NoisyDog : Dog {
}
```

However, there's a problem: that code doesn't compile. The reason is that the compiler is in doubt as to whether the init(name:) initializer is implemented by every possible subtype of Dog. To reassure the compiler, we must declare that initializer with the required keyword:

```
class Dog {
    var name : String
    required init(name:String) { // *
        self.name = name
    }
    class func makeAndName() -> Dog {
        let d = Self(name:"Fido")
        return d
    }
}
class NoisyDog : Dog {
}
```

I promised earlier that I'd tell you why you might need to declare an initializer as required; now I'm fulfilling that promise! The required designation means that every subclass of Dog *must inherit or reimplement* init(name:), so the compiler permits us to call `init(name:)` on a type reference that might refer to Dog or some subclass of Dog.

So now our code compiles, and we can call our function:

```
let d = Dog.makeAndName() // d is a Dog named Fido
let d2 = NoisyDog.makeAndName() // d2 is a NoisyDog named Fido
```

That code works as expected. But now there's *another* problem. Although `d2` is in fact a NoisyDog, it is *typed* as a Dog. That's because our `makeAndName` class method is declared as returning a Dog. That isn't what we want to declare. We want to declare that this method returns an instance *of the same type* as the class to which the `make-AndName` message was originally sent. In other words, we need a polymorphic type declaration! That type is `Self` once again:

```
class Dog {
    var name : String
    required init(name:String) {
        self.name = name
    }
    class func makeAndName() -> Self { // *
        let d = Self(name:"Fido")
        return d
    }
}
class NoisyDog : Dog {
}
```

The `Self` type is used as a return type in a method declaration to mean "an instance of whatever type this is at runtime." So when we call `NoisyDog.makeAndName()` we get a NoisyDog typed as NoisyDog.

`Self` also works for instance method declarations. Therefore, we can write an instance method version of our factory method. Here, we start with a Dog or a Noisy-Dog and tell it to have a puppy of the same type as itself:

```
class Dog {
    var name : String
    required init(name:String) {
        self.name = name
    }
    func havePuppy(name:String) -> Self {
        return Self(name:name)
    }
}
class NoisyDog : Dog {
}
```

And here's some code to test it:

```
let d = Dog(name:"Fido")
let d2 = d.havePuppy(name:"Fido Junior")
let nd = NoisyDog(name:"Rover")
let nd2 = nd.havePuppy(name:"Rover Junior")
```

As expected, `d2` is a Dog, but `nd2` is a NoisyDog typed as NoisyDog.

Type as Value

In some situations, you may want to treat an object type *as a value.* That is legal. An object type is itself an object, of a sort; it's what Swift calls a *metatype.* So an object type can be assigned to a variable or passed as a parameter:

- To *declare* that an object type is acceptable — as when declaring the type of a variable or parameter — use dot-notation with the name of the type and the keyword `Type`.
- To *use* an object type as a value — as when assigning a type to a variable or passing it as a parameter — hand the object to `type(of:)`, or use dot-notation with the name of the type and the keyword `self`. (In the latter case, the name of the type might be `Self`, in which case you'll be saying `Self.self`!)

Here's a function `dogTypeExpecter` that accepts a Dog type as its parameter:

```
func dogTypeExpecter(_ whattype:Dog.Type) {
}
```

And here's an example of calling that function:

```
dogTypeExpecter(Dog.self)
```

Or you could call it like this:

```
let d = Dog()
dogTypeExpecter(type(of:d))
```

The substitution principle applies, so you could call `dogTypeExpecter` like this:

```
dogTypeExpecter(NoisyDog.self)
```

Or like this:

```
let nd = NoisyDog()
dogTypeExpecter(type(of:nd))
```

To illustrate more practically, I'll rewrite our Dog factory method as a global factory function that will accept a Dog *type* as a parameter and will create an instance from that type. You can use a variable reference to a type (a metatype) to instantiate that type, but you can't just append parentheses to a variable reference:

```
func dogMakerAndNamer(_ whattype:Dog.Type) -> Dog {
    let d = whattype(name:"Fido") // compile error
    return d
}
```

Instead, you must explicitly send the reference an `init(...)` message:

```
func dogMakerAndNamer(_ whattype:Dog.Type) -> Dog {
    let d = whattype.init(name:"Fido")
    return d
}
```

And here's how to call our function:

```
let d = dogMakerAndNamer(Dog.self) // d is a Dog named Fido
let d2 = dogMakerAndNamer(NoisyDog.self) // d2 is a NoisyDog named Fido
```

Unfortunately, the global factory function `dogMakerAndNamer`, displays the same problem we had before — it returns an object typed as Dog, even if the underlying instance is in fact a NoisyDog. We can't return `Self` to solve the problem here, because there's no type for it to refer to. Swift does have a solution, however — generics. I'll discuss generic functions later in this chapter.

Summary of Type Terminology

All this terminology can get a bit confusing, so here's a quick summary:

`type(of:)`
: Applied to an object: the polymorphic (internal) type of the object, regardless of how a reference is typed.

`Self`
: In a method body, or in a method declaration when specifying the return type, this type or this instance's type, polymorphically.

`.Type`
: Appended to a type in a type declaration to specify that the type itself (or a subtype) is expected.

`.self`
: Sent to a type to generate a metatype, suitable for passing where a type (`.Type`) is expected.

Comparing Types

Type references can be compared to one another. On the right side of an `==` comparison, you can use the name of a type with `.self`; on the right side of an `is` comparison, you can use the name of a type with `.Type`. The difference, as you might expect, is that `==` tests for absolutely identical types, whereas `is` permits subtypes.

In this artificial example, if the parameter `whattype` is `Dog.self`, both `equality` and `typology` are `true`; if `whattype` is `NoisyDog.self`, `equality` is `false` but `typology` is still `true`:

```
func dogTypeExpecter(_ whattype:Dog.Type) {
    let equality = whattype == Dog.self
    let typology = whattype is Dog.Type
}
```

In that example, `whattype` might be replaced on the left side of the comparisons by the result of a call to `type(of:)` (or by a type name qualified by `.self`, though that would be pointless); and `Dog.self` might be replaced on the right side of the `==` comparison by `whattype` or the result of a call to `type(of:)`. But neither `whattype` nor `type(of:)` can appear on the right side of an `is` comparison; `is` requires a literal type as its second operand.

In real life, however, comparing type references is a *very* rare thing to do. Passing metatypes around is not Swifty, and comparing metatypes is *really* not Swifty. In general, if you find yourself talking like that, you should probably think of another way of doing whatever it is you're trying to do.

Protocols

A *protocol* is a way of expressing commonalities between otherwise unrelated types. For example, a Bee object and a Bird object might have features in common by virtue of the fact that both a bee and a bird can fly. Thus, it might be useful to define a Flier type. The question is: In what sense can both Bee and Bird be Fliers?

One possibility might be class inheritance. If Bee and Bird are both classes, Flier could be the superclass of both Bee and Bird. However, there may be other reasons why Flier *can't* be the superclass of both Bee and Bird. A Bee is an Insect; a Bird isn't. Yet they both have the power of flight — independently. We need a type that cuts across the class hierarchy somehow, tying remote classes together.

Moreover, what if Bee and Bird are *not* both classes? In Swift, that's a real possibility. Important and powerful objects can be structs instead of classes. But there is no hierarchy of superstructs and substructs! How can a Bee struct and a Bird struct both be Fliers?

Swift solves this problem through the use of protocols. Protocols are tremendously important in Swift; the Swift header defines over 60 protocols! Moreover, Objective-C has protocols as well, and Cocoa makes heavy use of protocols; Swift protocols correspond roughly to Objective-C protocols, and can interchange with them.

A protocol is an object *type*, but there are no protocol *objects* — you can't instantiate a protocol. A protocol declaration is just a lightweight list of properties and methods. The properties have no values, and the methods have no code! The idea is that a "real" object type can formally declare that it belongs to a protocol type; this is called *adopting* the protocol. An object type that adopts a protocol is promising to implement the properties and methods listed by the protocol. And it must keep that promise! This is called *conforming to* the protocol.

Let's say that being a Flier consists of no more than implementing a `fly` method. Then a Flier protocol could specify that there must be a `fly` method; to do so, it lists the `fly` method *with no function body*, like this:

```
protocol Flier {
    func fly()
}
```

Any type — an enum, a struct, a class, or even another protocol — can then adopt this protocol. To do so, it lists the protocol after a colon after its name in its declaration. (If the adopter is a class with a superclass, the protocol comes after a comma after the superclass specification.)

Let's say Bird is a struct. Then it can adopt Flier like this:

```
struct Bird : Flier {
} // compile error
```

So far, so good. But that code won't compile. The Bird struct has promised to implement the features listed in the Flier protocol. Now it must keep that promise! The `fly` method is the only requirement of the Flier protocol. To satisfy that requirement, I'll just give Bird an empty `fly` method:

```
protocol Flier {
    func fly()
}
struct Bird : Flier {
    func fly() {
    }
}
```

That's all there is to it. We've defined a protocol, and we've made a struct adopt and conform to that protocol. Of course, in real life you'll probably want to make the adopter's implementation of the protocol's methods *do* something; but the protocol says nothing about that.

A protocol can also declare a method *and provide its implementation*, thanks to protocol extensions, which I'll discuss later in this chapter.

Why Protocols?

Perhaps at this point you're wondering: So what? If we wanted a Bird to know how to fly, why didn't we just give Bird a `fly` method *without* adopting any protocol? What difference does the protocol make?

The answer has to do with types. A protocol, such as our Flier, is a kind of type. Therefore, I can *use* Flier as a type when declaring the type of a variable or a function parameter:

```
func tellToFly(_ f:Flier) {
    f.fly()
}
```

Protocols thus give us another way of expressing the notion of type and subtype — and *polymorphism applies*. By the substitution principle, a Flier here could be an instance of *any* object type, *as long as it adopts the Flier protocol.* It might be a Bird, it might be something else; we don't care. If it adopts the Flier protocol, it can be passed where a Flier is expected; moreover, it must have a `fly` method, because that's exactly what it *means* to adopt the Flier protocol! Therefore we can confidently send the `fly` message to this object, and the compiler lets us do that.

The converse, however, is not true: an object with a `fly` method is *not* automatically a Flier. It isn't enough to *obey* the requirements of a protocol; the object type must formally *adopt* the protocol. This code won't compile:

```
func tellToFly(_ f:Flier) {
    f.fly()
}
struct Bee {
    func fly() {
    }
}
let b = Bee()
tellToFly(b) // compile error
```

A Bee *can* be sent the `fly` message, *qua* Bee. But `tellToFly` doesn't take a Bee parameter; it takes a Flier parameter. Formally, a Bee is *not* a Flier. To make a Bee a Flier, just declare formally that Bee adopts the Flier protocol! This code does compile:

```
func tellToFly(_ f:Flier) {
    f.fly()
}
struct Bee : Flier {
    func fly() {
    }
}
let b = Bee()
tellToFly(b)
```

Adopting a Library Protocol

Enough of birds and bees; we're ready for a real-life example! The Swift standard library is chock full of protocols already; let's make one of our own object types adopt one and watch the powers of that protocol spring to life for us.

The CustomStringConvertible protocol requires that we implement a `description` String property. If we do that, a wonderful thing happens: when an instance of this type is used in string interpolation or with `print` (or the `po` command in the console, or in the String initializer `init(describing:)`), its `description` property value is used automatically to represent it.

Recall the Filter enum, from earlier in this chapter. I'll add a `description` property to it:

```
enum Filter : String {
    case albums = "Albums"
    case playlists = "Playlists"
    case podcasts = "Podcasts"
    case books = "Audiobooks"
    var description : String { return self.rawValue }
}
```

But that isn't enough, in and of itself, to give Filter the power of the CustomString-Convertible protocol; to do that, we also need to *adopt* the CustomStringConvertible protocol formally. There is already a colon and a type in the Filter declaration, so an adopted protocol comes after a comma:

```
enum Filter : String, CustomStringConvertible {
    case albums = "Albums"
    case playlists = "Playlists"
    case podcasts = "Podcasts"
    case books = "Audiobooks"
    var description : String { return self.rawValue }
}
```

We have now made Filter formally adopt the CustomStringConvertible protocol. The CustomStringConvertible protocol requires that we implement a `description` String property; we *do* implement a `description` String property, so our code compiles. Now we can interpolate a Filter into a string, or hand it over to `print`, or coerce it to a String, and its `description` will be used automatically:

```
let type = Filter.albums
print("It is \(type)") // It is Albums
print(type) // Albums
let s = String(describing:type) // Albums
```

Behold the power of protocols. You can give *any* object type the power of string conversion in exactly the same way.

Note that a type can adopt more than one protocol! The built-in Double type adopts CustomStringConvertible, Hashable, Strideable, and several other built-in protocols. To declare adoption of multiple protocols, list them separated by a comma after the protocol name and colon in the declaration (after the raw value type or superclass type if there is one):

```
struct MyType : CustomStringConvertible, TextOutputStreamable, Strideable {
    // ...
}
```

(Of course, that code won't compile unless I also declare, in MyType, any required properties and methods, so that MyType actually conforms to those protocols.)

Protocol Type Testing and Casting

The operators for mediating between an object's declared type and its real type work when the declared type is a protocol type. Given a protocol Flier that is adopted by both Bird and Bee, we can use the `is` operator to test whether a particular Flier is in fact a Bird:

```
func isBird(_ f:Flier) -> Bool {
    return f is Bird
}
```

Similarly, `as!` and `as?` can be used to cast an object declared as a protocol type down to its actual type. This is important to be able to do, because the adopting object will typically be able to receive messages that the protocol can't receive. Let's say that a Bird can get a worm:

```
struct Bird : Flier {
    func fly() {
    }
    func getWorm() {
    }
}
```

A Bird can `fly` *qua* Flier, but it can `getWorm` only *qua* Bird. You can't tell just any old Flier to get a worm:

```
func tellGetWorm(_ f:Flier) {
    f.getWorm() // compile error
}
```

But if this Flier is a Bird, clearly it *can* get a worm. That is exactly what casting is all about:

```
func tellGetWorm(f:Flier) {
    (f as? Bird)?.getWorm()
}
```

Declaring a Protocol

Protocol declaration can take place only at the top level of a file. To declare a protocol, use the keyword `protocol` followed by the name of the protocol (which should start with a capital letter, as this is a type). Then come curly braces which may contain declarations for any of the following:

Properties

A property declaration in a protocol consists of `var` (not `let`), the property name, a colon, its type, and curly braces containing the word `get` or the words `get set`. In the former case, the adopter's implementation of this property *can* be writable, while in the latter case, it *must* be: the adopter may not implement a `get set` property as a read-only computed property or as a constant (`let`) stored property.

To declare a static/class property, precede it with the keyword `static`. A class adopter is free to implement this as a `class` property.

Methods

A method declaration in a protocol is a function declaration without a function body; it has no curly braces and no code. Any object function type is legal, including `init` and `subscript`. (The syntax for declaring a subscript in a protocol is the same as the syntax for declaring a subscript in an object type, except that the curly braces will contain `get` or `get set`.)

To declare a static/class method, precede it with the keyword `static`. A class adopter is free to implement this as a `class` method.

To permit an enum or struct adopter to declare a method `mutating`, declare it `mutating` in the protocol. An adopter cannot add `mutating` if the protocol lacks it, but the adopter may omit `mutating` if the protocol has it.

A protocol can itself adopt one or more protocols; the syntax is just as you would expect — a colon after the protocol's name in the declaration, followed by a comma-separated list of the protocols it adopts. In effect, this gives you a way to create an entire secondary hierarchy of types! The Swift headers make heavy use of this.

A protocol that adopts another protocol may repeat the contents of the adopted protocol's curly braces, for clarity; but it doesn't have to, as this repetition is implicit. An object type that adopts a protocol must satisfy the requirements of this protocol and all protocols that the protocol adopts.

Protocol Composition

If the only purpose of a protocol is to combine other protocols by adopting all of them, without adding any new requirements, you can avoid formally declaring the protocol in the first place by specifying the protocol combination on the fly. To do so, join the protocol names with `&`. This is called *protocol composition:*

```
func f(_ x: CustomStringConvertible & CustomDebugStringConvertible) {
}
```

That is a function declaration with a parameter whose type is specified as being some object type that adopts both the CustomStringConvertible protocol and the CustomDebugStringConvertible protocol.

A type can also be specified as a composite of a class type and one or more protocols. A case in point might look something like this:

```
protocol MyViewProtocol {
    func doSomethingReallyCool()
}
class ViewController: UIViewController {
    var v: (UIView & MyViewProtocol)?
    func test() {
        self.v?.doSomethingReallyCool() // a MyViewProtocol requirement
        self.v?.backgroundColor = .red // a UIView property
    }
}
```

To be assigned to ViewController's `v` property, an object would need to be an instance of a UIView subclass that is also an adopter of MyViewProtocol. In this way, we guarantee to the compiler that both UIView messages and MyViewProtocol messages can be sent to a ViewController's `v`; otherwise, we'd have to type `v` as a MyViewProtocol and then cast down to UIView in order to send it UIView messages, even if we knew that `v` would in fact always be a UIView.

There's another way to accomplish the same thing; we can declare MyViewProtocol itself in such a way that it can be adopted *only* by UIView, as I shall now explain.

Class Protocols

A protocol declaration may include the name of a class after the colon. This limits the types capable of adopting this protocol to that class or its subclasses:

```
protocol MyViewProtocol : UIView {
    func doSomethingReallyCool()
}
class ViewController: UIViewController {
    var v: MyViewProtocol? // and therefore a UIView
    func test() {
        self.v?.doSomethingReallyCool() // a MyViewProtocol requirement
        self.v?.backgroundColor = .red // a UIView property
    }
}
```

Here, MyViewProtocol can be adopted only by UIView or a UIView subclass. This means that an object typed as MyViewProtocol can be sent both MyViewProtocol messages and UIView messages, because *ex hypothesi* a MyViewProtocol adopter must *be* a UIView.

To specify that a protocol can be adopted only by *some* class (and not a struct or enum) without specifying *what* class it must be, use the protocol type AnyObject, which every class type adopts (as I'll explain later):

```
protocol MyClassProtocol : AnyObject {
    // ...
}
```

An alternative notation is a where clause before the curly braces. I have not yet talked about where clauses or the use of `Self` to signify the protocol's adopter, but I'll show you the notation now anyway:

```
protocol MyViewProtocol where Self:UIView {
    func doSomethingReallyCool()
}
protocol MyClassProtocol where Self:AnyObject {
    // ...
}
```

A valuable byproduct of declaring a class protocol is that the resulting type can take advantage of special memory management features that apply only to classes. I haven't discussed memory management yet, but I'll give an example anyway (and I'll repeat it when I talk about memory management in Chapter 5):

```
protocol SecondViewControllerDelegate : AnyObject {
    func accept(data:Any)
}
class SecondViewController : UIViewController {
    weak var delegate : SecondViewControllerDelegate?
    // ...
}
```

The keyword `weak` marks the `delegate` property as having special memory management that applies only to class instances. The `delegate` property is typed as a protocol, and a protocol might be adopted by a struct or an enum type. So to satisfy the compiler that this object *will* in fact be a class instance, and *not* a struct or enum instance, the protocol is declared as a class protocol.

An `@objc` protocol is a class protocol, as class protocols are the only kind of protocol Objective-C understands.

Optional Protocol Members

In Objective-C, a protocol member can be declared optional, meaning that this member doesn't have to be implemented by the adopter, but it may be. Swift allows optional protocol members, but this feature is solely for compatibility with Objective-C, and in fact is implemented *by* Objective-C; it isn't really a Swift feature at all. Therefore, everything about an optional protocol member must be explicitly exposed to Objective-C. The protocol declaration must be marked with the `@objc` attribute, and an optional member's declaration must be marked with the keywords `@objc optional`:

```
@objc protocol Flier {
    @objc optional var song : String {get}
    @objc optional func sing()
}
```

(I'll explain in Chapter 11 *how* Objective-C implements optional protocol members.)

Many Cocoa protocols have optional members. For example, your iOS app will have an app delegate class that adopts the UIApplicationDelegate protocol; this protocol has many methods, all of them optional. That fact, however, will have no effect on how you implement those methods; either you implement a method or you don't. (I'll talk more about Cocoa protocols in Chapter 11, and about delegate protocols in Chapter 12.)

An optional member is not guaranteed to be implemented by the adopter, so Swift doesn't know whether it's safe to send a Flier either the `song` message or the `sing` message. How Swift solves that problem depends on whether this is an optional property or an optional method.

Optional properties

In the case of an optional property (like `song`), Swift solves the problem by wrapping its fetched value in an Optional. If the Flier adopter doesn't implement the property, the result is `nil` and no harm done:

```
@objc protocol Flier {
    @objc optional var song : String {get}
}
let f : Flier = Bird()
let s = f.song // s is an Optional wrapping a String
```

This is one of those rare situations where you can wind up with a double-wrapped Optional. If the value of the optional property `song` were itself a `String?`, then fetching its value from a Flier would yield a `String??`.

A curious limitation is that if a protocol declares an optional property `{get set}`, you can't set that property. If `f` is a Flier and `song` is declared `{get set}`, you can't set `f.song`:

```
@objc protocol Flier {
    @objc optional var song : String? {get set}
}
let f : Flier = Bird()
f.song = "tweet tweet" // compile error
```

The error message claims that `f` is immutable, which is blatantly false. This is evidently a bug in the language. A workaround (pointed out to me by Jordan Rose) is to use key path notation (which I'll explain in Chapter 5):

```
let f : Flier = Bird()
f[keyPath: \.song] = "tweet tweet"
```

Optional methods

In the case of an optional method (like `sing`), things are more elaborate. If the method is not implemented, we must not be permitted to call it in the first place. To handle this situation, the method is typed as an Optional version of its declared type. To send the `sing` message to a Flier, therefore, you must unwrap it. What you are unwrapping is not the result of the method call; it's the method *itself.* In the method call, the unwrap operator must appear *before* the parentheses!

The safe approach is to unwrap an optional method optionally, with a question mark:

```
@objc protocol Flier {
    @objc optional func sing()
}
let f : Flier = Bird()
f.sing?()
```

The effect is to send the `sing` message to `f` only if this Flier adopter implements `sing`. If this Flier adopter *doesn't* implement `sing`, nothing happens. You could have force-unwrapped the call — `f.sing!()` — but then your app would crash if the adopter doesn't implement `sing`.

If an optional method returns a value, that value is wrapped in an Optional as well:

```
@objc protocol Flier {
    @objc optional func sing() -> String
}
```

If we now call `sing?()` on a Flier, the result is an Optional wrapping a String:

```
let f : Flier = Bird()
let s = f.sing?() // s is an Optional wrapping a String
```

If we force-unwrap the call — `f.sing!()` — the result is either a String (if the adopter implements `sing`) or a crash (if it doesn't).

Implicitly Required Initializers

Suppose that a protocol declares an initializer. And suppose that a class adopts this protocol. By the terms of this protocol, this class and any subclass it may ever have must implement this initializer. Therefore, the class not only must implement the initializer, but also must mark it as `required`. An initializer declared in a protocol is *implicitly required*, and a class that adopts this protocol is forced to make that requirement explicit.

Consider this simple example, which won't compile:

```
protocol Flier {
    init()
}
class Bird : Flier {
    init() {} // compile error
}
```

That code generates an elaborate but perfectly informative compile error message: "Initializer requirement `init()` can only be satisfied by a `required` initializer in non-final class Bird." To compile our code, we must designate our initializer as `required`:

```
protocol Flier {
    init()
}
class Bird : Flier {
    required init() {}
}
```

Alternatively, if Bird were marked `final`, there would be no need to mark its `init` as `required`, because this would mean that Bird cannot have any subclasses — guaranteeing that the problem will never arise in the first place.

In the above code, Bird is *not* marked as `final`, and its `init` *is* marked as `required`. This, as I've already explained, means that any subclass of Bird that implements any designated initializers — and thus loses initializer inheritance — must implement the required initializer and mark it `required` as well.

That fact is responsible for a strange and annoying feature of real-life iOS programming with Swift. Let's say you subclass the built-in Cocoa class UIViewController — something that you are extremely likely to do. And let's say you give your subclass an initializer — something that you are also extremely likely to do:

```
class ViewController: UIViewController {
    init() {
        super.init(nibName:"ViewController", bundle:nil) // compile error
    }
}
```

That code won't compile. The compile error says: "`required` initializer `init(coder:)` must be provided by subclass of UIViewController." What's going on here?

It turns out that UIViewController adopts a protocol called NSCoding. And this protocol requires an initializer `init(coder:)`, which UIViewController duly implements. None of that is your doing; UIViewController and NSCoding are declared by Cocoa, not by you. But that doesn't matter! This is the same situation I was just describing. Your UIViewController subclass must either inherit `init(coder:)` or must explicitly implement it and mark it `required`. Well, your subclass has implemented a designated initializer of its own — thus *cutting off initializer inheritance*. Therefore it *must* implement `init(coder:)` and mark it `required`.

But that makes no sense if you are not expecting `init(coder:)` ever to be *called* on your UIViewController subclass. You are being forced to write an initializer for which you can provide no meaningful functionality! Fortunately, Xcode's Fix-it feature (Chapter 10) will offer to write the initializer for you, like this:

```
required init?(coder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

That code satisfies the compiler. The initializer is aborted, so it is legal even though it doesn't fulfill an initializer's contract. It also crashes if it is ever called — which is fine, because *ex hypothesi* you don't expect it ever to be called. (I'll talk more about `fatalError` in Chapter 5).

If, on the other hand, you *do* have functionality for this initializer, you will delete the `fatalError` line and insert that functionality in its place. A minimum meaningful implementation would be to call `super.init(coder:coder)`, but of course if your class has properties that need initialization, you will need to initialize them first.

Not only UIViewController but *lots* of built-in Cocoa classes adopt NSCoding. You will encounter this problem if you subclass *any* of those classes and implement your own initializer. It's just something you'll have to get used to.

Expressible by Literal

One of the delightful things about Swift is that so many of its features, rather than being built-in and accomplished by magic, are exposed in the Swift header. Literals are a case in point. The reason you can say `5` to express an Int whose value is 5, instead of formally initializing Int by saying `Int(5)`, is not because of magic (or at least, not entirely because of magic). It's because Int adopts a protocol, ExpressibleByIntegerLiteral. Not only Int literals, but *all* literals work this way. The following protocols are declared in the Swift header:

- ExpressibleByNilLiteral
- ExpressibleByBooleanLiteral
- ExpressibleByIntegerLiteral
- ExpressibleByFloatLiteral
- ExpressibleByStringLiteral
- ExpressibleByExtendedGraphemeClusterLiteral

- ExpressibleByUnicodeScalarLiteral
- ExpressibleByArrayLiteral
- ExpressibleByDictionaryLiteral

Your own object type can adopt an expressible-by-literal protocol as well. This means that a literal can appear where an instance of your object type is expected! Here we declare a Nest type that contains some number of eggs (its `eggCount`):

```
struct Nest : ExpressibleByIntegerLiteral {
    var eggCount : Int = 0
    init() {}
    init(integerLiteral val: Int) {
        self.eggCount = val
    }
}
```

Because Nest adopts ExpressibleByIntegerLiteral, we can pass an Int where a Nest is expected, and our `init(integerLiteral:)` will be called automatically, causing a new Nest object with the specified `eggCount` to come into existence at that moment:

```
func reportEggs(_ nest:Nest) {
    print("this nest contains \(nest.eggCount) eggs")
}
reportEggs(4) // this nest contains 4 eggs
```

Generics

A *generic* is a sort of placeholder for a type, into which an actual type will be slotted later. In particular, there are situations where you want to say that a certain *same* type is to be used in several places, without specifying precisely *what* type this is to be. Swift generics allow you to say that, without sacrificing or evading Swift's fundamental strict typing.

We're already familiar with one very important case in point: an Optional. Any type of value can be wrapped up in an Optional. Yet there is no doubt as to what type is wrapped up in a *particular* Optional. How can this be? Here's how an Optional works.

I have already said that an Optional is an enum, with two cases: `.none` and `.some`. If an Optional's case is `.some`, it has an associated value — the value that is wrapped by this Optional. But what is the type of that associated value? On the one hand, one wants to say that it can be any type; that, after all, is why anything can be wrapped up in an Optional. On the other hand, any *particular* Optional can wrap a value only of some one *specific* known type.

That is the very essence of a generic! The declaration for the Optional enum in the Swift header starts like this:

```
enum Optional<Wrapped> : ExpressibleByNilLiteral {
    // ...
}
```

That syntax means: "In the course of this enum declaration, I'm going to be using a made-up type — a type *placeholder* — that I call Wrapped. It's a real and individual type, but I'm not going to say more about it right now. All you need to know is that whenever I say Wrapped, I mean this one particular type. When an actual Optional is created, it will be perfectly clear what type Wrapped stands for, and then, wherever I say Wrapped, you should substitute the type that it stands for."

Let's look at more of the Optional declaration:

```
enum Optional<Wrapped> : ExpressibleByNilLiteral { ❶
    case none
    case some(Wrapped) ❷
    init(_ some: Wrapped) ❸
    // ...
}
```

Here's how that code works. The placeholder type name Wrapped appears in three different places:

❶ The angle-bracket syntax `<Wrapped>` *declares* that Wrapped is a placeholder.

❷ Besides the case `.none`, there's also a case `.some`, which has an associated value — of type Wrapped.

❸ There's also an initializer, which takes a parameter — of type Wrapped.

All of that constitutes a sort of template for how Optional works. In real life, Swift has strict typing, so any *particular* Optional instance must have a real type substituted for the Wrapped placeholder. The process of substitution is called *resolving* (or *specializing*) the generic. The type in question must be made unambiguously clear for each actual usage of this enum, so that the compiler is satisfied.

And how will the placeholder be resolved? Here's one way. When an Optional is created, it will be initialized with an actual value of some definite type:

```
let s = Optional("howdy")
```

We're calling `init(_ some: Wrapped)`, so `"howdy"` is being supplied here as a Wrapped instance — resolving the generic as String. So the compiler now knows that Wrapped must be String *throughout* this particular `Optional<Wrapped>`. The type with which we are initialized — whatever type that may be — *is* type Wrapped, and therefore is the type associated with the `.some` case. The declaration for this *particular* Optional looks, in the compiler's mind, like this (pseudocode):

```
enum Optional<String> {
    case none
    case some(String)
    init(_ some: String)
    // ...
}
```

That is the pseudocode declaration of an Optional whose Wrapped placeholder has been replaced everywhere with the String type. We can summarize this by saying that `s` is an `Optional<String>`. In fact, that is legal syntax! We can create the same Optional like this:

```
let s : Optional<String> = "howdy"
```

Not only can types (such as enums) be generic; so can functions. A generic function solves a problem that was posed earlier. We wrote a global factory function for dogs:

```
func dogMakerAndNamer(_ whattype:Dog.Type) -> Dog {
    let d = whattype.init(name:"Fido")
    return d
}
```

That works, but it isn't quite what we'd like to say. This function's declared return type is Dog. So if we are passed a Dog subclass such as NoisyDog as the parameter, we will instantiate that type (which is good) but then return that instance typed as Dog (which is bad). Instead, we'd like the type declared as the return type after the arrow operator to be the *same* type that we were passed as a parameter in the first line and that we instantiated in the second line — whatever that type may be. Generics permit us to say that:

```
func dogMakerAndNamer<WhatType:Dog>(_:WhatType.Type) -> WhatType {
    let d = WhatType.init(name:"Fido")
    return d
}
```

The generic is resolved when we actually call that function. For example:

```
let dog = dogMakerAndNamer(NoisyDog.self)
```

In that call, we pass `NoisyDog.self` as the parameter. That tells the compiler what WhatType is! It is NoisyDog. In effect, the compiler now substitutes NoisyDog for WhatType throughout the generic, like this (pseudocode):

```
func dogMakerAndNamer(_:NoisyDog.Type) -> NoisyDog {
    let d = NoisyDog.init(name:"Fido")
    return d
}
```

And this resolution extends beyond the generic itself. Now that the compiler knows that this call to our function will return a NoisyDog instance, it can type the variable initialized to the result of the call as a NoisyDog by inference:

```
let dog = dogMakerAndNamer(NoisyDog.self) // dog is typed as NoisyDog
```

Generic Declarations

Here's a list of places where generics, in one form or another, can be declared in Swift:

Generic protocol with `Self`
: In a protocol body, use of the keyword `Self` turns the protocol into a generic. `Self` here is a placeholder meaning *the type of the adopter*. Here's a Flier protocol that declares a method that takes a `Self` parameter:

```
protocol Flier {
    func flockTogetherWith(_ f:Self)
}
```

That means that if a Bird object type adopts the Flier protocol, its implementation of `flockTogetherWith` must declare its parameter as a Bird:

```
struct Bird : Flier {
    func flockTogetherWith(_ f:Bird) {}
}
```

Generic protocol with associated type
: A protocol can declare an *associated type* using an `associatedtype` statement. This turns the protocol into a generic; the associated type name is a placeholder:

```
protocol Flier {
    associatedtype T
    func flockTogetherWith(_ f:T)
    func mateWith(_ f:T)
}
```

An adopter will declare a particular type at some point where the generic uses the associated type name, resolving the placeholder:

```
struct Bee {}
struct Bird : Flier {
    func flockTogetherWith(_ f:Bee) {}
    func mateWith(_ f:Bee) {}
}
```

T here does *not* have to be this adopter of Flier, or *any* adopter of Flier. It could be String, or Int, or anything at all. Rather, the protocol dictates that whatever a Flier adopter is declared as flocking together with, it must be declared as mating with as well. The Bird struct adopts the Flier protocol and declares the parameter of `flockTogetherWith` as a Bee. That declaration resolves T to Bee for this particular adopter — and therefore Bird must declare the parameter for `mateWith` as a Bee as well.

Generic functions

A function declaration can use a generic placeholder type for any of its parameters, for its return type, and within its body. The placeholder name is declared in angle brackets after the function name:

```
func takeAndReturnSameThing<T> (_ t:T) -> T {
    print(T.self)
    return t
}
```

The caller will use a particular type at some point where the placeholder appears in the function declaration, resolving the placeholder:

```
let thing = takeAndReturnSameThing("howdy")
```

Here, the type of the argument `"howdy"` used in the call resolves T to String; therefore this call to `takeAndReturnSameThing` must also return a String, and the type of the variable capturing the result, `thing`, is inferred as String.

Generic object types

An object type declaration can use a generic placeholder type anywhere within its curly braces. The placeholder name is declared in angle brackets after the object type name:

```
struct HolderOfTwoSameThings<T> {
    var firstThing : T
    var secondThing : T
    init(thingOne:T, thingTwo:T) {
        self.firstThing = thingOne
        self.secondThing = thingTwo
    }
}
```

A user of this object type will use a particular type at some point where the placeholder appears in the object type declaration, resolving the placeholder:

```
let holder = HolderOfTwoSameThings(thingOne:"howdy", thingTwo:"bye")
```

Here, the type of the `thingOne:` argument, `"howdy"`, used in the initializer call, resolves T to String; therefore `thingTwo:` must also be a String, and the properties `firstThing` and `secondThing` must be Strings as well.

The angle brackets that declare a placeholder may declare multiple placeholders, separated by a comma:

```
func flockTwoTogether<T, U>(_ f1:T, _ f2:U) {}
```

The two parameters of `flockTwoTogether` can now be resolved to two different types (though they do not *have* to be different).

Inside a generic's code, the generic placeholder is a type reference standing for the resolved type, which can be interrogated using type reference comparison, as described earlier in this chapter:

```
func takeAndReturnSameThing<T> (_ t:T) -> T {
    if T.self is String.Type {
        // ...
    }
    return t
}
```

If we call `takeAndReturnSameThing("howdy")`, the condition will be true. That sort of thing, however, is unusual; a generic whose behavior depends on interrogation of the placeholder type may need to be rewritten in some other way.

Type Constraints

A generic declaration can *limit* the types that are eligible to be used for resolving a particular placeholder. This is called a *type constraint*.

The simplest form of type constraint is to put a colon and a type name after the placeholder's name when it is declared. The type name after the colon can be a class name or a protocol name:

Class name
: A class name means that the type must be this class *or a subclass* of this class.

Protocol name
: A protocol name means that the type must be *an adopter* of this protocol.

For a protocol associated type, the type constraint can appear as part of the `associatedtype` declaration:

```
protocol Flier {
    func fly()
}
protocol Flocker {
    associatedtype T : Flier // *
    func flockTogetherWith(f:T)
}
struct Bee : Flier {
    func fly() {}
}
struct Bird : Flocker {
    func flockTogetherWith(f:Bee) {}
}
```

In that example, Flocker's associated type T is constrained to be an adopter of Flier. Bee *is* an adopter of Flier; therefore Bird can adopt Flocker by specifying that the parameter type in its `flockTogetherWith` implementation is Bee.

Observe that we could not have achieved the same effect without the associated type, by declaring Flocker like this:

```
protocol Flocker {
    func flockTogetherWith(f:Flier)
}
```

That's not the same thing! That requires that a Flocker adopter specify the parameter for `flockTogetherWith` *as Flier*; we would have had to write Bird like this:

```
struct Bird : Flocker {
    func flockTogetherWith(f:Flier) {}
}
```

For a generic function or a generic object type, the type constraint can appear in the angle brackets. For example, earlier I described a global function `func dogMakerAndNamer<WhatType:Dog>`; Dog is a class, so the constraint says that WhatType must be Dog or a Dog subclass.

A type constraint on a placeholder is often used to reassure the compiler that some message can be sent to an instance of the placeholder type. Let's say we want to implement a function `myMin` that returns the smallest member of a list. Here's a promising implementation as a generic function, but there's one problem — it doesn't compile:

```
func myMin<T>(_ things:T...) -> T {
    var minimum = things.first!
    for item in things.dropFirst() {
        if item < minimum { // compile error
            minimum = item
        }
    }
    return minimum
}
```

The problem is the comparison `item < minimum`. How does the compiler know that the type T, the type of `item` and `minimum`, will be resolved to a type that can in fact be compared using the less-than operator in this way? It doesn't, and that's exactly why it rejects that code. The solution is to promise the compiler that the resolved type of T *will* in fact work with the less-than operator. The way to do that, it turns out, is to constrain T to Swift's built-in Comparable protocol:

```
func myMin<T:Comparable>(_ things:T...) -> T {
```

Now `myMin` compiles, because it cannot be called except by resolving T to an object type that adopts Comparable and hence *can* be compared with the less-than operator. Naturally, built-in object types that you think should be comparable, such as Int, Double, String, and Character, do in fact adopt the Comparable protocol! If you look in the Swift headers, you'll find that the built-in `min` global function is declared in just this way, and for just this reason.

A generic protocol type can be used *only* as a type constraint. If you try to use it in any other way, you'll get a compile error. This restriction can be quite frustrating. One way of circumventing it is *type erasure*; for an excellent discussion, see *http://robnapier.net/erasure*.

Explicit Specialization

In the generic examples so far, the placeholder's type has been resolved mostly through *inference*. But there's another way to perform resolution: we can resolve the type *manually*. This is called *explicit specialization*.

There are two forms of explicit specialization:

Generic protocol with associated type

The adopter of a protocol can resolve an associated type manually through a type alias equating the associated type with some explicit type:

```
protocol Flier {
    associatedtype T
}
struct Bird : Flier {
    typealias T = String
}
```

Generic object type

The user of a generic object type can resolve a placeholder type manually using the same angle bracket syntax used to declare the generic in the first place, with the actual type name in the angle brackets:

```
class Dog<T> {
    var name : T?
}
let d = Dog<String>()
```

You cannot explicitly specialize a generic function. One solution is to make your generic function take a type parameter resolving the generic. That's what I did in my earlier `dogMakerAndNamer` example:

```
func dogMakerAndNamer<WhatType:Dog>(_:WhatType.Type) -> WhatType {
    let d = WhatType.init(name:"Fido")
    return d
}
```

The parameter to `dogMakerAndNamer` is never used within the function body, which is why it has no name (just an underscore). It does, however, serve to resolve the generic!

Another approach is not to use a generic function in the first place. Instead, declare a generic object type wrapping a nongeneric function that uses the generic type's placeholder. The generic type *can* be explicitly specialized, resolving the placeholder in the function:

```
protocol Flier {
    init()
}
struct Bird : Flier {
    init() {}
}
struct FlierMaker<T:Flier> {
    static func makeFlier() -> T {
        return T()
    }
}
let f = FlierMaker<Bird>.makeFlier() // returns a Bird
```

When a class is generic, you can subclass it, provided you resolve the generic. You can do this either through a matching generic subclass or by resolving the superclass generic explicitly. Here's a generic Dog:

```
class Dog<T> {
    func speak(_ what:T) {}
}
```

You can subclass it as a generic whose placeholder matches that of the superclass:

```
class NoisyDog<T> : Dog<T> {}
```

That's legal because the resolution of the NoisyDog placeholder T will resolve the Dog placeholder T. The alternative is a nongeneric subclass of an explicitly specialized Dog:

```
class NoisyDog : Dog<String> {}
```

In that case, a method override in the subclass can use the specialized type where the superclass uses the generic:

```
class NoisyDog : Dog<String> {
   override func speak(_ what:String) {}
}
```

Generic Types and Covariance

A generic object type can't be used as the declared type of anything. Given a class `Dog<T>`, you can't declare a variable as a Dog plain and simple:

```
class Dog<T> {
    func speak(_ what:T) {}
}
var d: Dog? // compile error
```

Only a *resolved* generic object type can be a declared type — and there can be as many of those as there are possible resolutions, each different resolution constituting a distinct type:

```
class Dog<T> {
    func speak(_ what:T) {}
}
var d: Dog<Int>? // that's a type
var d2: Dog<String>? // that's a different type
// ... and so on ...
```

The exception, which is only apparent, is when the resolution can be inferred from the context. Here's a classic example:

```
class Node<T> {
    let value:T
    let parent:Node?
    init(_ value:T, parent:Node?) {
        self.value = value
        self.parent = parent
    }
}
```

It looks as if the `parent` property is typed as an unresolved Node (wrapped in an Optional); but it isn't. In the absence of an explicit specialization, the compiler assumes we mean `Node<T>`, resolving `parent` to the same type as the surrounding instance.

Not only are different specializations of a generic *distinct* types; they are *unrelated* types. In particular, a generic type specialized to a subtype is *not polymorphic* with respect to the same generic type specialized to a supertype.

Suppose we have a simple generic Wrapper struct along with a Cat class and its CalicoCat subclass:

```
struct Wrapper<T> {
}
class Cat {
}
class CalicoCat : Cat {
}
```

Then you can't assign a Wrapper specialized to CalicoCat where a Wrapper specialized to Cat is expected:

```
let w : Wrapper<Cat> = Wrapper<CalicoCat>() // compile error
```

It appears that polymorphism is failing here — but it isn't. The two generic types, `Wrapper<Cat>` and `Wrapper<CalicoCat>`, are not superclass and subclass. Rather, if this assignment were possible, we would say that the types are *covariant*, meaning that the polymorphic relationship between the specializations of the placeholders is applied to the generic types themselves. Certain Swift built-in generic types *are* covariant; Optional is a clear example! But, frustratingly, covariance is not a *general* language feature; there's no way for you to specify that *your* generic types should be covariant.

One workaround is to have your generic placeholder constrained to a protocol, and have your types adopt that protocol:

```
protocol Meower {
    func meow()
}
struct Wrapper<T:Meower> {
    let meower : T
}
class Cat : Meower {
    func meow() { print("meow") }
}
class CalicoCat : Cat {
}
```

Now it is legal to say:

```
let w : Wrapper<Cat> = Wrapper(meower:CalicoCat())
```

Associated Type Chains

When a generic placeholder is constrained to a generic protocol with an associated type, you can refer to the associated type using dot-notation: the placeholder name, a dot, and the associated type name.

Here's an example. Imagine that in a game program, soldiers and archers are enemies of one another. I'll express this by subsuming a Soldier struct and an Archer struct under a Fighter protocol that has an Enemy associated type, which is itself constrained to be a Fighter:

```
protocol Fighter {
    associatedtype Enemy : Fighter
}
```

I'll resolve that associated type manually for both the Soldier and the Archer structs:

```
struct Soldier : Fighter {
    typealias Enemy = Archer
}
struct Archer : Fighter {
    typealias Enemy = Soldier
}
```

Now I'll create a generic struct to express the opposing camps of these fighters:

```
struct Camp<T:Fighter> {
}
```

Now suppose that a camp may contain a spy from the opposing camp. What is the type of that spy? Well, if this is a Soldier camp, it's an Archer; and if it's an Archer camp, it's a Soldier. More generally, since T is a Fighter, it's the type of the Enemy of this adopter of Fighter. I can express that as `T.Enemy`:

```
struct Camp<T:Fighter> {
    var spy : T.Enemy?
}
```

The result is that if, for a particular Camp, T is resolved to Soldier, `T.Enemy` means Archer (and vice versa). The compiler will enforce that rule:

```
var c = Camp<Soldier>()
c.spy = Archer() // compiles
c.spy = Soldier() // compile error
```

Longer chains of associated type names are possible — because a generic protocol might have an associated type which is constrained to a generic protocol that *also* has an associated type. Let's give each type of Fighter a characteristic weapon: a soldier has a sword, while an archer has a bow. I'll make a Sword struct and a Bow struct, and I'll unite them under an Armament protocol:

```
protocol Armament {}
struct Sword : Armament {}
struct Bow : Armament {}
```

I'll add a Weapon associated type to Fighter, which is constrained to be an Armament, and I'll resolve it manually for each type of Fighter:

```
protocol Fighter {
    associatedtype Enemy : Fighter
    associatedtype Weapon : Armament
}
struct Soldier : Fighter {
    typealias Weapon = Sword
    typealias Enemy = Archer
}
struct Archer : Fighter {
    typealias Weapon = Bow
    typealias Enemy = Soldier
}
```

Now let's say that every Fighter has the ability to steal his enemy's weapon. I'll give the Fighter generic protocol a `steal(weapon:from:)` method. How can the Fighter generic protocol express the parameter types in a way that forces its adopter to declare this method with the proper types?

The `from:` parameter type is this Fighter's Enemy. We already know how to express that: it's the placeholder plus dot-notation with the associated type name. Here, the placeholder is the adopter of this protocol — namely, `Self`. So the `from:` parameter type is `Self.Enemy`. And what about the `weapon:` parameter type? That's the Weapon of that Enemy! So the `weapon:` parameter type is `Self.Enemy.Weapon`:

```
protocol Fighter {
    associatedtype Enemy : Fighter
    associatedtype Weapon : Armament
    func steal(weapon:Self.Enemy.Weapon, from:Self.Enemy)
}
```

(We could omit `Self` from that code, and it would compile and would mean the same thing. But `Self` would still be the implicit start of the chain, and I think explicit `Self` makes the meaning of the code clearer.)

The following declarations for Soldier and Archer correctly adopt the Fighter protocol, and the compiler approves:

```
struct Soldier : Fighter {
    typealias Weapon = Sword
    typealias Enemy = Archer
    func steal(weapon:Bow, from:Archer) { }
}
struct Archer : Fighter {
    typealias Weapon = Bow
    typealias Enemy = Soldier
    func steal(weapon:Sword, from:Soldier) { }
}
```

(There's a trick for shortening potentially long associated type chains, which I'll demonstrate in the next section.)

Where Clauses

The most flexible way to express a type constraint is to add a where clause. Before I tell you what a where clause looks like, I'll tell you where it goes:

- For a generic function, a where clause may appear after the signature declaration (after the parameter list, following the arrow operator and return type if included).
- For a generic type, a where clause may appear after the type declaration, before the curly braces.
- For a generic protocol, a where clause may appear after the protocol declaration, before the curly braces.
- For an associated type in a generic protocol, a where clause may appear at the end of the associated type declaration.

Now let's talk about the syntax of a where clause. It starts with the keyword `where`. Then what? One possibility is a comma-separated list of additional constraints on an already declared placeholder.

I'll start with some trivial examples. You already know that we can constrain a placeholder at the point of declaration, using a colon and a type (which might be a protocol composition):

```
func flyAndWalk<T: Flier> (_ f:T) {}
func flyAndWalk2<T: Flier & Walker> (_ f:T) {}
```

Using a where clause, we can move those constraints out of the angle brackets. No new functionality is gained, but the resulting notation is arguably neater (and we can even use multiple type constraints to replace the protocol composition):

```
func flyAndWalk<T> (_ f:T) where T: Flier {}
func flyAndWalk2<T> (_ f:T) where T: Flier & Walker {}
func flyAndWalk2a<T> (_ f:T) where T: Flier, T: Walker {}
```

Similarly, an associated type declaration can express a type constraint as a where clause. Instead of this:

```
protocol Flocker {
    associatedtype T : Flier
}
```

We can write this, which is arguably clearer:

```
protocol Flocker {
    associatedtype T where T : Flier
}
```

Now let's do something a bit more powerful. When a constraint on a placeholder is a generic protocol with an associated type, you can use an associated type chain to impose additional constraints *on the associated type*. This pseudocode shows what I mean (I've omitted the content of the where clause, to focus on what the where clause will be constraining):

```
protocol Flier {
    associatedtype T
}
func flockTogether<U> (_ f:U) where U:Flier, U.T /* ... */ {}
```

In that pseudocode, the placeholder U is constrained to be a Flier adopter — and Flier is itself a generic protocol, with an associated type T. We can constrain the types eligible to resolve this Flier adopter's T — and this, in turn, will further constrain the types eligible to resolve U.

Let's fill in the blank in our pseudocode. What sort of restriction are we allowed to impose here? One possibility is a colon expression, as for any type constraint:

```
protocol Flier {
    associatedtype T
}
struct Bird : Flier {
    typealias T = String
}
```

```
struct Insect : Flier {
    typealias T = Bird
}
func flockTogether<U> (_ f:U) where U:Flier, U.T:Equatable {}
```

Both Bird and Insect adopt Flier. The `flockTogether` function can be called with a Bird argument, because a Bird's T associated type is resolved to String, which adopts the built-in Equatable protocol. But `flockTogether` can't be called with an Insect argument, because an Insect's T associated type is resolved to Bird, which *doesn't* adopt the Equatable protocol:

```
flockTogether(Bird()) // okay
flockTogether(Insect()) // compile error
```

The other possibility is the equality operator == followed by a type or an associated type chain, and the constrained type must then match it *exactly:*

```
protocol Flier {
    associatedtype T
}
struct Bird : Flier {
    typealias T = String
}
struct Airplane : Flier {
    typealias T = String
}
struct Insect : Flier {
    typealias T = Int
}
func flockTwoTogether<U1,U2> (_ f1:U1, _ f2:U2)
    where U1:Flier, U2:Flier, U1.T == U2.T {}
```

Bird, Airplane, and Insect are all Flier adopters. The `flockTwoTogether` function can be called with a Bird and a Bird, or an Insect and an Insect, or an Airplane and an Airplane, or even a Bird and an Airplane. But it can't be called with an Insect and a Bird, because they don't resolve their associated type T to the same type as one another.

The Swift header makes extensive use of where clauses with an == operator, especially as a way of restricting a sequence type. Take the String `append(contentsOf:)` method, declared like this:

```
mutating func append<S>(contentsOf newElements: S)
    where S:Sequence, S.Element == Character
```

The Sequence protocol has an Element associated type, representing the type of the sequence's elements. This where clause means that a sequence of Character — but *not* a sequence of something else, such as Int — can be concatenated to a String:

```
var s = "hello"
s.append(contentsOf: " there")
s.append(contentsOf: Array(" world"))
s.append(contentsOf: ["!" as Character])
```

The Array `append(contentsOf:)` method is declared a little differently:

```
mutating func append<S>(contentsOf newElements: S)
    where S:Sequence, S.Element == Self.Element
```

An array is a sequence; its element type is its Element associated type. The where clause means that you can append to an Array the elements of any sort of Sequence, but only if they are the same kind of element as the elements of this array. If the array consists of String elements, you can add more String elements to it, but you can't add Int elements.

Actually, a sequence's Element associated type is just a kind of shorthand. In reality, a sequence has an Iterator associated type, which is constrained to be an adopter of the generic IteratorProtocol, which in turn has an associated type Element. So a sequence's element type is its Iterator.Element. But a generic protocol or its associated type can have a where clause, and this can be used to reduce the length of associated type chains:

```
protocol Sequence {
    associatedtype Iterator : IteratorProtocol
    associatedtype Element where Self.Element == Self.Iterator.Element
    // ...
}
```

As a result, wherever the Swift header would have to say `Iterator.Element`, it can say simply `Element` instead.

Extensions

An *extension* is a way of injecting your own code into an object type that has already been declared elsewhere; you are *extending* an existing object type. You can extend your own object types; you can also extend one of Swift's object types or one of Cocoa's object types, in which case you are *adding functionality* to a type that doesn't belong to you!

Extension declaration can take place only at the top level of a file. To declare an extension, put the keyword `extension` followed by the name of an existing object type, then optionally a colon plus the names of any protocols you want to add to the list of those adopted by this type, and finally curly braces containing the usual things that go inside an object type declaration — with some restrictions:

- An extension can't declare a stored property (but it can declare a computed property).

- An extension of a class can't declare a designated initializer or a deinitializer (but it can declare a convenience initializer).
- An extension can't override an existing member (but it can overload an existing method), and a method declared in an extension can't be overridden.

In my real programming life, I sometimes extend a built-in Swift or Cocoa type just to inject some missing functionality by expressing it as a property or method.

For example, Cocoa's Core Graphics framework has many useful functions associated with the CGRect struct, and Swift already extends CGRect to add some helpful properties and methods; but there's no shortcut for getting the center point (a CGPoint) of a CGRect, something that in practice is often needed. I extend CGRect to give it a `center` property:

```
extension CGRect {
    var center : CGPoint {
        return CGPoint(x:self.midX, y:self.midY)
    }
}
```

An extension can declare a static or class member; this can be a good way to slot a global function into an appropriate namespace. In one of my apps, I find myself frequently using a certain color (a UIColor). Instead of creating that color repeatedly, it makes sense to encapsulate the instructions for generating it in a global function. But instead of making that function *completely* global, I make it — appropriately enough — a read-only static property of UIColor:

```
extension UIColor {
    static var myGolden : UIColor {
        return self.init(
            red:1.000, green:0.894, blue:0.541, alpha:0.900
        )
    }
}
```

Now I can use that color throughout my code as `UIColor.myGolden`, parallel to built-in class properties such as `UIColor.red`.

Extensions on one's own object types can help to organize code. A common convention is to add an extension for each protocol the object type needs to adopt:

```
class ViewController: UIViewController {
    // ... UIViewController method overrides go here ...
}
extension ViewController : UIPopoverPresentationControllerDelegate {
    // ... UIPopoverPresentationControllerDelegate methods go here ...
}
extension ViewController : UIToolbarDelegate {
    // ... UIToolbarDelegate methods go here ...
}
```

Extensions and Overrides

The rules about class extensions and overrides are more complicated than I've stated. A native Swift method in an extension can neither override nor be overridden, but then Objective-C comes along with its own rules and messes everything up. Let's say we have a class Dog and its subclass NoisyDog:

- If we have an extension on Dog that declares a method, NoisyDog *can* override the Dog extension method *if* the method is exposed to Objective-C.
- If we have a method in Dog, an extension on NoisyDog *can* override it *if* Dog's method is exposed to Objective-C and is marked `dynamic`.

Things are made even messier by the existence of modules; if there's a type with a method in a module, an extension on that type in another module can declare *the same method*, not overriding but effectively replacing it. My advice: Don't do that.

An extension on your own object type can also be a way to spread your definition of that object type over multiple files, if you feel that several shorter files are better than one long file.

When you extend a Swift struct, a curious thing happens with initializers: it becomes possible to declare an initializer and keep the implicit initializers:

```
struct Digit {
    var number : Int
}
extension Digit {
    init() {
        self.init(number:42)
    }
}
```

In that code, the explicit declaration of an initializer through an extension did not cause us to lose the implicit memberwise initializer, as would have happened if we had declared the same initializer inside the original struct declaration. Now we can instantiate a Digit by calling the explicitly declared initializer — `Digit()` — or by calling the implicit memberwise initializer — `Digit(number:7)`.

Extending Protocols

When you extend a protocol, you can add methods and properties to the protocol, just as for any object type. Unlike a protocol declaration, these methods and properties are not mere requirements, to be fulfilled by the adopter of the protocol; they are actual methods and properties, to be *inherited* by the adopter of the protocol (or perhaps I should say they are *injected* into the adopter):

```
protocol Flier {
}
extension Flier {
    func fly() {
        print("flap flap flap")
    }
}
struct Bird : Flier {
}
```

As that code demonstrates, Bird can now adopt Flier *without* implementing the `fly` method! That's because the Flier protocol extension supplies the `fly` method implementation and injects it into Bird:

```
let b = Bird()
b.fly() // flap flap flap
```

An adopter can still provide its own implementation of a method inherited from a protocol extension:

```
protocol Flier {
}
extension Flier {
    func fly() {
        print("flap flap flap")
    }
}
struct Insect : Flier {
    func fly() {
        print("whirr")
    }
}
let i = Insect()
i.fly() // whirr
```

But this kind of inheritance is *not polymorphic!* The adopter's implementation is not an override; it is merely another implementation. The internal identity rule does *not* apply; it matters how a reference is typed:

```
let f : Flier = Insect()
f.fly() // flap flap flap (!!)
```

Even though `f` is internally an Insect (as we can discover with the `is` operator), the `fly` message is being sent to an object reference typed as Flier, so it is Flier's implementation of the `fly` method that is called, not Insect's implementation.

To get something that looks like polymorphic inheritance, we must also declare `fly` as a requirement *in the original protocol*:

```
protocol Flier {
    func fly() // *
}
extension Flier {
    func fly() {
        print("flap flap flap")
    }
}
struct Insect : Flier {
    func fly() {
        print("whirr")
    }
}
```

Now an Insect maintains its internal integrity:

```
let f : Flier = Insect()
f.fly() // whirr
```

Protocol extensions sometimes allow you to say things that a type adopting the protocol would not be able to say directly. Here's an example. We have a Dog class and its NoisyDog subclass, and we have a DogWrapper struct that wraps a Dog:

```
class Dog {
    func wrapped() -> DogWrapper { DogWrapper(self) }
}
class NoisyDog : Dog {}
struct DogWrapper {
    private let dog : Dog
    init(_ dog: Dog) { self.dog = dog }
    func unwrap() -> Dog { self.dog }
}
```

However, we're not getting quite the result I was hoping for. The return type of `unwrap` is Dog, whereas I want it to be the *real* type that is wrapped inside the DogWrapper (Dog or NoisyDog). That sounds like a generic:

```
struct DogWrapper<T:Dog> {
    private let dog : T
    init(_ dog: T) { self.dog = dog }
    func unwrap() -> T { self.dog }
}
```

So far, so good; but now Dog doesn't compile:

```
class Dog {
    func wrapped() -> DogWrapper { DogWrapper(self) } // compile error
}
```

The trouble is that the return type of `wrapped`, DogWrapper, is no longer a legal type; it is now generic, and a generic type can't be used as the declared type of anything. We have to resolve the generic with angle brackets. But we can't do that either, because there is no way to say "the type that I really am" (Dog or NoisyDog); the phrase `DogWrapper<Self>` is illegal.

The solution is to remove the `wrapped` method from Dog itself, and inject it back into Dog by way of a protocol extension:

```
protocol WrappableDog : Dog { }
extension WrappableDog {
    func wrapped() -> DogWrapper<Self> { DogWrapper(self) }
}
class Dog : WrappableDog {
}
```

That's legal because the phrase `DogWrapper<Self>` *is* legal in a protocol. (Thanks go to Tyler Prevost; see *https://stackoverflow.com/a/66414305/341994*.)

Extending Generics

When you extend a generic type, the generic's placeholder type names are visible to your extension. That's good, because you might need to use them; but it can make your code a little mystifying, because you seem to be using an undefined type name out of the blue. It might be a good idea to add a comment, to remind yourself what you're up to:

```
class Dog<T> {
    var name : T?
}
extension Dog {
    func sayYourName() -> T? { // T? is the type of self.name
        return self.name
    }
}
```

A generic type extension declaration can include a where clause. This limits *which* resolutions of the generic placeholder can call the code injected by this extension.

That, in turn, can be used to assure the compiler that your code is legal. For instance, recall this example from earlier in this chapter:

```
func myMin<T:Comparable>(_ things:T...) -> T {
    var minimum = things.first!
    for item in things.dropFirst() {
        if item < minimum {
            minimum = item
        }
    }
    return minimum
}
```

That's a global function. I'd prefer to inject it into Array as a method. I can do that with an extension. Array is a generic struct whose placeholder type is called Element. To make this work, I need somehow to bring along the Comparable type constraint that makes this code legal; without it, as you remember, my use of the < operator won't compile. I can do that with a where clause on the extension:

```
extension Array where Element:Comparable {
    func myMin() -> Element? {
        var minimum = self.first
        for item in self.dropFirst() {
            if item < minimum! {
                minimum = item
            }
        }
        return minimum
    }
}
```

The where clause is a constraint guaranteeing that this array's elements adopt Comparable, so the compiler permits the use of the `<` operator — and it doesn't permit the `myMin` method to be called on an array whose elements *don't* adopt Comparable. The Swift standard library makes heavy use of that sort of thing, and in fact Sequence has a `min` method declared like `myMin`.

Starting in Swift 5.3, a where clause constraining a generic's placeholder can be applied to a *method* of the generic:

```
extension Array {
    func myMin() -> Element? where Element:Comparable {
        // ...
    }
}
```

The advantages are primarily organizational. In Swift 5.2 and before, the only way for a generic type to restrict a method to one or more constraints on the placeholder type was to put the method into an extension with all of the constraints in the extension declaration:

```
struct MyStruct<T> {
    // ...
}
extension MyStruct where T:Protocol1, T:Protocol2 {
    func f() {
        // ...
    }
}
```

Now, however, we can distribute the statement of the constraints in whatever way seems clearest. Perhaps we prefer this:

```
extension MyStruct where T:Protocol1 {
    func f() where T:Protocol2 {
        // ...
    }
}
```

If the type belongs to us, we could forego the extension altogether:

```
struct MyStruct<T> {
    // ...
    func f() where T:Protocol1, T:Protocol2 {
        // ...
    }
}
```

An extension with a where clause can also be used to express *conditional conformance* to a protocol. The syntax is:

```
extension GenericType : SomeProtocol where constraint {
```

The idea is that a generic type should adopt a certain protocol only if something is true of its placeholder type — and the extension then contains whatever is needed to satisfy the protocol requirements.

In the standard library, conditional conformance fills what used to be a serious hole in the Swift language. For example, an Array can consist of Equatable elements, and in that case it is possible to compare two arrays for equality:

```
let arr1 = [1,2,3]
let arr2 = [1,2,3]
if arr1 == arr2 { // ...
```

It's clear what array equality should consist of: the two arrays should consist of the same elements in the same order. The elements must be Equatable so as to guarantee the meaningfulness of the notion "same elements."

Now consider what it takes to compare two arrays of arrays:

```
let arr1 = [[1], [2], [3]]
let arr2 = [[1], [2], [3]]
if arr1 == arr2 { // ...
```

That's possible only if we can make Array *itself* Equatable. In particular, we want to assert that an Array should be Equatable only just in case its elements are Equatable. That's conditional conformance! Here's what the standard library says:

```
extension Array : Equatable where Element : Equatable {
    // ...
}
```

Umbrella Types

Swift provides a few built-in types as general umbrella types, capable of embracing multiple real types under a single heading.

Any

The Any type is the universal Swift umbrella type. Where an Any object is expected, absolutely any object or function can be passed, without casting:

```
func anyExpecter(_ a:Any) {}
anyExpecter("howdy")      // a struct instance
anyExpecter(String.self) // a struct type
anyExpecter(Dog())        // a class instance
anyExpecter(Dog.self)    // a class type
anyExpecter(anyExpecter) // a function
```

Going the other way, if you want to type an Any object as a more specific type, you will generally have to cast down. Such a cast is legal for any specific object type or function type. A forced cast isn't safe, but you can easily make it safe, because you can also test an Any object against any specific object type or function type. Here, `anything` is typed as Any:

```
if anything is String {
    let s = anything as! String
    // ...
}
```

(In Chapter 5 I'll introduce a more elegant syntax for casting down safely.)

The Any umbrella type is the general medium of interchange between Swift and the Cocoa Objective-C APIs. When an Objective-C object type is nonspecific (`id`), it will appear to Swift as Any. Commonly encountered examples are UserDefaults and key–value coding (Chapter 11); these allow you to *pass* an object of indeterminate class along with a string key name, and they allow you to *retrieve* an object of indeterminate class by a string key name.

For example, the first parameter of UserDefaults `set(_:forKey:)` is typed as Any (actually, an Optional wrapping Any, so that it can be `nil`). So I can pass anything as the argument. Here, I'll pass a Date:

```
let ud = UserDefaults.standard
ud.set(Date(), forKey:"now")
```

When a Swift object is assigned or passed to an Any that acts as a conduit to Objective-C, it crosses the bridge to Objective-C. If the object's type is not an Objective-C type (a class derived from NSObject), it will be transformed in order to cross the bridge. If this type is automatically bridged to an Objective-C class type, it becomes that type; other types are boxed up in a way that allows them to survive the journey into Objective-C's world, even though Objective-C can't deal with them directly. (For full details, see the Appendix.)

To illustrate, suppose we have an Objective-C class Thing with a method `take1id:`, declared like this:

```
- (void) take1id: (id) anid;
```

That appears to Swift as:

```
func take1id(_ anid: Any)
```

When we pass an object to `take1Id(_:)` as its parameter, it crosses the bridge:

```
let t = Thing()
t.take1id("howdy")  // String to NSString
t.take1id(1)        // Int to NSNumber
t.take1id(CGRect()) // CGRect to NSValue
t.take1id(Date())   // Date to NSDate
t.take1id(Bird())   // Bird (struct) to boxed type
```

Coming back the other way, if Objective-C hands you an Any object, you will need to cast it down to its underlying type in order to do anything useful with it:

```
let ud = UserDefaults.standard
let d = ud.object(forKey:"now")
if d is Date {
    let d = d as! Date
    // ...
}
```

The result returned from UserDefaults `object(forKey:)` is typed as Any — actually, as an Optional wrapping an Any, because UserDefaults might need to return `nil` to indicate that no object exists for that key. But you know that it's supposed to be a date, so you cast it down to Date.

AnyObject

AnyObject is an empty protocol with the special feature that *all class types* conform to it automatically. Although Objective-C APIs present Objective-C `id` as Any in Swift, Swift AnyObject *is* Objective-C `id`. AnyObject is useful primarily when you want to take advantage of the *behavior* of Objective-C `id`, as I'll demonstrate in a moment.

A class instance can be assigned directly where an AnyObject is expected; to retrieve it as its original type, you'll need to cast down:

```
class Dog {
}
let d = Dog()
let anyo : AnyObject = d
let d2 = anyo as! Dog
```

Assigning a nonclass type to an AnyObject requires casting (with `as`). The bridge to Objective-C is then crossed immediately, as I described for Any in the preceding section:

```
let s = "howdy" as AnyObject  // String to NSString to AnyObject
let i = 1 as AnyObject        // Int to NSNumber to AnyObject
let r = CGRect() as AnyObject // CGRect to NSValue to AnyObject
let d = Date() as AnyObject   // Date to NSDate to AnyObject
let b = Bird() as AnyObject   // Bird (struct) to boxed type to AnyObject
```

Suppressing type checking

Because AnyObject is Objective-C `id`, it can be used, like Objective-C `id`, to suspend the compiler's judgment as to whether a certain message can be sent to an object. Thus, you can send a message to an AnyObject without bothering to cast down to its real type.

You can't send just any old message to an AnyObject; this is an Objective-C feature, so the message must correspond to a class member that meets one of the following criteria:

- It is a member of an Objective-C class.
- It is a member of your own Swift subclass of an Objective-C class.
- It is a member of your own Swift extension of an Objective-C class.
- It is a member of a Swift class or protocol marked `@objc`.

This feature is fundamentally parallel to optional protocol members, which I discussed earlier in this chapter. Let's start with two classes:

```
class Dog {
    @objc var noise : String = "woof"
    @objc func bark() -> String {
        return "woof"
    }
}
class Cat {}
```

The Dog property `noise` and the Dog method `bark` are marked `@objc`, so they are visible as potential messages to be sent to an AnyObject. To prove it, I'll type a Cat as an AnyObject and send it one of those messages. Let's start with the `noise` property:

```
let c : AnyObject = Cat()
let s = c.noise
```

That code, amazingly, compiles. Moreover, it doesn't crash when the code runs! The `noise` property has been typed as an Optional wrapping its original type. Here, that's an Optional wrapping a String. If the object typed as AnyObject doesn't implement `noise`, the result is `nil` and no harm done.

Now let's try it with a method call:

```
let c : AnyObject = Cat()
let s = c.bark?()
```

Again, that code compiles and is safe. If the Object typed as AnyObject doesn't implement `bark`, no `bark()` call is performed; the method result type has been wrapped in an Optional, so `s` is typed as `String?` and has been set to `nil`. If the AnyObject turns out to have a `bark` method (because it's a Dog), the result is an Optional wrapping the returned String. If you call `bark!()` on the AnyObject instead, the result will be a String, but you'll crash if the AnyObject doesn't implement `bark`. Unlike an optional protocol member, you can even send the message *with no unwrapping*. This is legal:

```
let c : AnyObject = Cat()
let s = c.bark()
```

That's just like force-unwrapping the call: the result is a String, but it's possible to crash.

Don't make a habit of sending messages to an AnyObject; because it involves dynamic lookup, it's expensive at build time and expensive at runtime.

Object identity

Sometimes, what you want to know is not what *type* an object is, but whether an object itself is the *particular object* you think it is. This problem can't arise with a value type, but it can arise with a reference type — in particular, with class instances.

Swift's solution is the identity operator (===). Its operands are typed as `AnyObject?`, meaning an object whose type is a class or an Optional whose wrapped type is a class; it compares one object reference with another. This is not a comparison of *values*, like the equality operator (==); you're asking whether two object *references* refer to *one and the same* object. There is also a negative version (`!==`) of the identity operator.

A typical use case is that a class instance arrives from Cocoa, and you need to know whether it is in fact a particular object to which you already have a reference. For example, a Notification has an `object` property that helps identify the notification (usually, it is the original sender of the notification). We can use === to test whether this `object` is a certain object to which we already have a reference. However, `object` is typed as Any (actually, as an Optional wrapping Any), so we must cast to Any-Object in order to take advantage of the identity operator:

```
@objc func changed(_ n:Notification) {
    let player = MPMusicPlayerController.applicationMusicPlayer
    if n.object as AnyObject === player {
        // ...
    }
}
```

AnyClass

AnyClass is the type of AnyObject. It corresponds to the Objective-C Class type. It arises typically in declarations where a Cocoa API wants to say that a class is expected. The UIView `layerClass` class property is declared, in its Swift translation, like this:

```
class var layerClass : AnyClass {get}
```

That means that if you override this class property, you'll implement your getter to return a class (which will presumably be a CALayer subclass):

```
override class var layerClass : AnyClass { CATiledLayer.self }
```

A reference to an AnyClass object behaves much like a reference to an AnyObject object. You can send it any Objective-C message that Swift knows about — any Objective-C *class* message. To demonstrate, once again I'll start with two classes:

```
class Dog {
    @objc static var whatADogSays : String = "woof"
}
class Cat {}
```

Objective-C can see `whatADogSays`, and it sees it as a class property. Therefore you can send `whatADogSays` to an AnyClass reference:

```
let c : AnyClass = Cat.self
let s = c.whatADogSays
```

Collection Types

Swift, in common with most modern computer languages, has built-in collection types Array and Dictionary, along with a third type, Set. Array and Dictionary are sufficiently important that the language accommodates them with some special syntax.

Array

An array (Array, a struct) is an ordered collection of object instances (the *elements* of the array) accessible by index number, where an index number is an Int numbered from `0`. If an array contains four elements, the first has index `0` and the last has index `3`. A Swift array cannot be sparse: if there is an element with index `3`, there is also an element with index `2` and so on.

The salient feature of Swift arrays is their strict typing. Unlike some other computer languages, a Swift array's elements must be *uniform* — that is, the array must consist solely of elements of the same definite type. Even an empty array must have a definite element type, despite lacking elements at this moment. An array is itself typed in accordance with its element type. Two arrays whose elements are of different types are considered, themselves, to be of two different types: an array of Int elements has a different type from an array of String elements.

If all this reminds you of Optionals, it should. Like Optional, Array is a generic. It is declared as `Array<Element>`, where the placeholder Element is the type of a particular array's elements. And, like Optional types, Array types are covariant, meaning that they behave polymorphically in accordance with their element types: if NoisyDog is a subclass of Dog, then an array of NoisyDog can be used where an array of Dog is expected.

To state an Array type, then, you need to state its element type. You could explicitly resolve the generic placeholder; an array of Int elements would be an `Array<Int>`. However, Swift offers syntactic sugar using square brackets around the name of the element type, like this: `[Int]`. That's the syntax you'll use most of the time.

A literal array is represented as square brackets containing a list of its elements separated by a comma (and optional spaces): `[1,2,3]`. The literal for an empty array is empty square brackets: `[]`.

Array's default initializer `init()`, called by appending empty parentheses to the array's type, yields an empty array of that type. You can create an empty array of Int like this:

```
var arr = [Int]()
```

Alternatively, if a reference's type is known in advance, the empty array `[]` can be inferred as that type. So you can also create an empty array of Int like this:

```
var arr : [Int] = []
```

If you're starting with a literal array containing elements, you won't usually need to declare the array's type, because Swift can infer it by looking at the elements. Swift will infer that `[1,2,3]` is an array of Int. If the array element types consist of a class and its subclasses, like Dog and NoisyDog, Swift will infer the common superclass as the array's type. However, in some cases you will need to declare an array reference's type explicitly even while assigning a literal to that array:

```
let arr : [Any] = [1, "howdy"]         // mixed bag
let arr2 : [Flier] = [Insect(), Bird()] // protocol adopters
```

Moreover, if an array variable is declared and initialized to a literal with many elements, even though Swift might be able to infer the type, it's often a good idea to declare the variable's type explicitly anyway. This saves the compiler from having to examine the entire array to decide its type, and makes compilation faster.

Array also has an initializer whose parameter is a sequence. This means that if a type is a sequence, you can split an instance of it into the elements of an array. For example:

- `Array(1...3)` generates the array of Int `[1,2,3]`.
- `Array("hey")` generates the array of Character `["h","e","y"]`.

- `Array(d)`, where `d` is a Dictionary, generates an array of tuples of the key–value pairs of `d`.

Another Array initializer, `init(repeating:count:)`, lets you populate an array with the same value. In this example, I create an array of 100 Optional strings initialized to `nil`:

```
let strings : [String?] = Array(repeating:nil, count:100)
```

That's the closest you can get in Swift to a sparse array; we have 100 slots, each of which might or might not contain a string (and to start with, none of them do).

Beware of using `init(repeating:count:)` with a reference type! If Dog is a class, and you say `let dogs = Array(repeating:Dog(), count:3)`, you don't have an array of three Dogs; you have an array consisting of three *references* to *one* Dog. I'll give a workaround later.

Array casting and type testing

When you assign, pass, or cast an array of a certain type to another array type, you are really operating on the individual elements of the array:

```
let arr : [Int?] = [1,2,3]
```

That code is actually syntactic sugar: assigning an array of Int where an array of Optionals wrapping Int is expected constitutes a request that each individual Int in the original array should be wrapped in an Optional. And that is exactly what happens:

```
let arr : [Int?] = [1,2,3]
print(arr) // [Optional(1), Optional(2), Optional(3)]
```

Similarly, suppose we have a Dog class and its NoisyDog subclass; then this code is legal:

```
let dog1 : Dog = NoisyDog()
let dog2 : Dog = NoisyDog()
let arr = [dog1, dog2]
let arr2 = arr as! [NoisyDog]
```

In the third line, we have an array of Dog. In the fourth line, we apparently cast this array down to an array of NoisyDog — which really means that we cast each individual Dog in the first array to a NoisyDog. We can crash when we do that, but we won't if each element of the first array really *is* a NoisyDog.

The `as?` operator will cast an array to an Optional wrapping an array, which will be `nil` if the requested cast cannot be performed for each element individually:

```
let dog1 : Dog = NoisyDog()
let dog2 : Dog = NoisyDog()
let dog3 : Dog = Dog()
let arr = [dog1, dog2]
let arr2 = arr as? [NoisyDog] // Optional wrapping an array of NoisyDog
let arr3 = [dog2, dog3]
let arr4 = arr3 as? [NoisyDog] // nil
```

You can test each element of an array with the `is` operator by testing the array itself. Given the array of Dog from the previous code, you can say:

```
if arr is [NoisyDog] { // ...
```

That will be `true` if each element of the array is in fact a NoisyDog.

Array comparison

Array equality works just as you would expect: two arrays are equal if they contain the same number of elements and all the elements are pairwise equal in order. Of course, this presupposes that the notion "equal" is meaningful for these elements:

```
let i1 = 1
let i2 = 2
let i3 = 3
let arr : [Int] = [1,2,3]
if arr == [i1,i2,i3] { // they are equal!
```

Two arrays don't have to be of the same type to be compared against one another for equality, but the test won't succeed unless they do in fact contain objects that are equal to one another. Here, I compare a Dog array against a NoisyDog array; this is legal if equatability is defined for two Dogs. (Dog might be an NSObject subclass; or you might make Dog adopt Equatable, as I'll explain in Chapter 5.) The two arrays are in fact equal, because the dogs they contain are the same dogs in the same order:

```
let nd1 = NoisyDog()
let d1 = nd1 as Dog
let nd2 = NoisyDog()
let d2 = nd2 as Dog
let arr1 = [d1,d2] // [Dog]
let arr2 = [nd1,nd2] // [NoisyDog]
if arr1 == arr2 { // they are equal!
```

Arrays are value types

Because an array is a struct, it is a value type, not a reference type. This means that every time an array is assigned to a variable or passed as argument to a function, it is effectively copied.

I do not mean to imply, however, that merely assigning or passing an array is expensive, or that a lot of actual copying takes place every time. If the reference to an array is a constant, clearly no copying is necessary; and even operations that yield a new array derived from another array, or that mutate an array, may be quite efficient. You just have to trust that the designers of Swift have thought about these problems and have implemented arrays efficiently behind the scenes.

Although an array is *itself* a value type, its *elements* might not be. If an array of class instances is assigned to multiple variables, the result is multiple references to the same instances.

Array subscripting

The Array struct implements subscript methods to allow access to elements using square brackets after a reference to an array.

You can use an Int inside the square brackets. If an array is referred to by a variable `arr`, then `arr[1]` accesses the second element.

You can also use a Range of Int inside the square brackets. If `arr` is an array, then `arr[1...2]` signifies the second and third elements. Technically, an expression like `arr[1...2]` yields something called an ArraySlice, which stands in relation to Array much as Substring stands in relation to String (Chapter 3). It's very similar to an array, and in general you will probably pretend that an ArraySlice *is* an array. You can subscript an ArraySlice in just the same ways you would subscript an array. Nevertheless, they are not the same thing. An ArraySlice is not a new array; it's just a way of pointing into a section of the original array. For this reason, its index numbers are those of the original array:

```
let arr = ["manny", "moe", "jack"]
let slice = arr[1...2] // ["moe", "jack"]
print(slice[1]) // moe
```

The ArraySlice `slice` consists of two elements, `"moe"` and `"jack"`, of which `"moe"` is the first element. But these are not merely `"moe"` and `"jack"` taken *from* the original array, but the `"moe"` and `"jack"` *in* the original array. For this reason, their index numbers are not 0 and 1, but rather 1 and 2, just as in the original array. If you need to extract a new array based on this slice, coerce the slice to an Array:

```
let arr2 = Array(slice) // ["moe", "jack"]
print(arr2[1]) // jack
```

If the reference to an array is mutable (`var`, not `let`), then it's possible to assign to a subscript expression. This alters what's in that slot. Of course, what is assigned must accord with the type of the array's elements:

```
var arr = [1,2,3]
arr[1] = 4 // arr is now [1,4,3]
```

If the subscript is a range, what is assigned must be a slice. You can assign a literal array, because it will be coerced for you to an ArraySlice; but if what you're starting with is an array reference, you'll have to coerce it to a slice yourself. Such assignment can change the length of the array being assigned to:

```
var arr = [1,2,3]
arr[1..<2] = [7,8] // arr is now [1,7,8,3]
arr[1..<2] = []    // arr is now [1,8,3]
arr[1..<1] = [10]  // arr is now [1,10,8,3] (no element was removed!)
let arr2 = [20,21]
// arr[1..<1] = arr2 // compile error! You have to say this:
arr[1..<1] = ArraySlice(arr2) // arr is now [1,20,21,10,8,3]
```

Subscripting an array with a Range is an opportunity to use partial range notation. The missing value is taken to be the array's first or last index. If `arr` is `[1,2,3]`, then `arr[1...]` is `[2,3]`, and `arr[...1]` is `[1,2]`. Similarly, you can assign into a range specified as a partial range:

```
var arr = [1,2,3]
arr[1...] = [4,5] // arr is now [1,4,5]
```

It is a runtime error to access an element by an index number larger than the largest element number or smaller than the smallest element number. If `arr` has three elements, speaking of `arr[-1]` or `arr[3]` is not illegal linguistically, but your program will crash.

Nested arrays

It is legal for the elements of an array to be arrays:

```
let arr = [[1,2,3], [4,5,6], [7,8,9]]
```

That's an array of arrays of Int. Its type declaration, therefore, is `[[Int]]`. (No law says that the contained arrays have to be the same length; that's just something I did for clarity.)

To access an individual Int inside those nested arrays, you can chain subscripts:

```
let arr = [[1,2,3], [4,5,6], [7,8,9]]
let i = arr[1][1] // 5
```

If the outer array reference is mutable, you can also write into a nested array:

```
var arr = [[1,2,3], [4,5,6], [7,8,9]]
arr[1][1] = 100
```

You can modify the inner arrays in other ways as well; for example, you can insert additional elements into them.

Thanks to conditional conformance (discussed earlier in this chapter), nested arrays can be compared with == as long as the inner array's elements are Equatable. If `arr` and `arr2` are both `[[Int]]`, you can compare them by saying `arr == arr2`.

Basic array properties and methods

An array is a Collection, which is itself a Sequence. If those terms have a familiar ring, they should: the same is true of a String's underlying character sequence, which I discussed in Chapter 3. For this reason, an array and a character sequence have some properties and methods that bear striking similarities to one another.

Many additional methods on sequences and collections are provided in the Swift Algorithms package (`import Algorithms`; see Chapter 7). These, too, are applicable to arrays, so I'll mention some of them here as well.

As a collection, an array's `count` read-only property reports the number of elements it contains. If an array's `count` is `0`, its `isEmpty` property is `true`.

An array's `first` and `last` read-only properties return its first and last elements, but they are wrapped in an Optional because the array might be empty and so these properties would need to be `nil`. This is one of those rare situations in Swift where you can wind up with an Optional wrapping an Optional; consider an array of Optionals wrapping Ints, and what happens when you get its `last` property.

An array's largest accessible index is one less than its `count`. You may find yourself calculating index values with reference to the `count`; to refer to the last two elements of `arr`, you might say:

```
let arr = [1,2,3]
let slice = arr[arr.count-2...arr.count-1] // [2,3]
```

Can we do better? Unfortunately, Swift doesn't adopt the modern convention of letting you use negative indexes as a shorthand. On the other hand, if you want the last `n` elements of an array, you can use the `suffix(_:)` method:

```
let arr = [1,2,3]
let slice = arr.suffix(2) // [2,3]
```

Therefore, a neat way to obtain, say, the next-to-last element of an array is to combine `suffix` with `first`:

```
let arr = [1,2,3]
let nextToLast = arr.suffix(2).first // Optional(2)
```

Both `suffix(_:)` and its companion `prefix(_:)` yield ArraySlices, and have the remarkable feature that there is no penalty for going out of range:

```
let arr = [1,2,3]
let slice = arr.suffix(10) // [1,2,3] (and no crash)
```

Instead of describing the size of the suffix or prefix by its count, you can express the limit of the suffix or prefix by its index. And partial range notation may provide yet another useful alternative:

```
let arr = [1,2,3]
let slice = arr.suffix(from:1)     // [2,3]
let slice2 = arr[1...]             // [2,3]
let slice3 = arr.prefix(upTo:1)    // [1]
let slice4 = arr.prefix(through:1) // [1,2]
```

An array's `startIndex` property is `0`, and its `endIndex` property is its `count`. An array's `indices` property is a half-open range whose endpoints are the array's `start-Index` and `endIndex` — that is, a range accessing the entire array. Moreover, these values are Ints, so you can use ordinary arithmetic operations on them:

```
let arr = [1,2,3]
let slice = arr[arr.endIndex-2..<arr.endIndex] // [2,3]
```

But the `startIndex`, `endIndex`, and `indices` of an ArraySlice are measured against the original array; after the previous code, `slice.indices` is `1..<3`, and `slice.start-Index` is 1.

The `firstIndex(of:)` method reports the index of the first occurrence of an element in an array, but it is wrapped in an Optional so that `nil` can be returned if the element doesn't appear in the array. In general, the comparison uses == behind the scenes to identify the element being sought, and therefore the array elements must adopt Equatable (otherwise the compiler will stop you):

```
let arr = [1,2,3]
let ix = arr.firstIndex(of:2) // Optional wrapping 1
```

Alternatively, you can call `firstIndex(where:)`, supplying your own function that takes an element type and returns a Bool, and you'll get back the index of the first element for which that Bool is `true`. In this example, my Bird struct has a `name` String property:

```
let aviary = [Bird(name:"Tweety"), Bird(name:"Flappy"), Bird(name:"Lady")]
let ix = aviary.firstIndex {$0.name.count < 5} // Optional(2)
```

If what you want is not the index but the object itself, the `first(where:)` method returns it — wrapped, naturally, in an Optional. These methods are matched by `last-Index(of:)`, `lastIndex(where:)`, and `last(where:)`.

As a sequence, an array's `contains(_:)` method reports whether it contains an element. Again, you can rely on the == operator if the elements are Equatable, or you can supply your own function that takes an element type and returns a Bool:

```
let arr = [1,2,3]
let ok = arr.contains(2) // true
let ok2 = arr.contains {$0 > 3} // false
```

The `starts(with:)` method reports whether an array's starting elements match the elements of a given sequence of the same type. Once more, you can rely on the ==

operator for Equatable elements, or you can supply a function that takes two values of the element type and returns a Bool stating whether they match:

```
let arr = [1,2,3]
let ok = arr.starts(with:[1,2]) // true
let ok2 = arr.starts(with:[1,-2]) {abs($0) == abs($1)} // true
```

The `min` and `max` methods return the smallest or largest element in an array, wrapped in an Optional in case the array is empty. If the array consists of Comparable elements, you can let the `<` operator do its work; alternatively, you can call `min(by:)` or `max(by:)`, supplying a function that returns a Bool stating whether the smaller of two given elements is the first:

```
let arr = [3,1,-2]
let min = arr.min() // Optional(-2)
let min2 = arr.min {abs($0) < abs($1)} // Optional(1)
```

The Swift Algorithms package adds variants with a `count:` parameter, letting you specify how many smallest or largest elements you want returned, as an array:

```
let arr = [3,1,-2]
let min = arr.min(count: 2) // [-2, 1]
let min2 = arr.min(count: 2) {abs($0) < abs($1)} // [1, -2]
```

If the reference to an array is mutable, the `append(_:)` and `append(contentsOf:)` instance methods add elements to the end of it. The difference between them is that `append(_:)` takes a single value of the element type, while `append(contentsOf:)` takes a sequence of the element type:

```
var arr = [1,2,3]
arr.append(4)
arr.append(contentsOf:[5,6])
arr.append(contentsOf:7...8) // arr is now [1,2,3,4,5,6,7,8]
```

The `+` operator is overloaded to behave like `append(contentsOf:)` when the left-hand operand is an array, except that it generates a new array, so it works even if the reference to the array is a constant (`let`). If the reference to the array is mutable (`var`), you can append to it in place with the overloaded `+=` operator:

```
let arr = [1,2,3]
let arr2 = arr + [4] // arr2 is now [1,2,3,4]
var arr3 = [1,2,3]
arr3 += [4] // arr3 is now [1,2,3,4]
```

If the reference to an array is mutable, the instance method `insert(at:)` inserts a single element at the given index. To insert multiple elements at once, call the `insert(contentsOf:at:)` method. Assignment into a range-subscripted array, which I described earlier, is even more flexible.

If the reference to an array is mutable, the instance method `remove(at:)` removes the element at that index; the instance method `removeLast` removes the last element.

These methods also *return* the value that was removed from the array; you can ignore the returned value if you don't need it. These methods do not wrap the returned value in an Optional, and accessing an out-of-range index will crash your program. On the other hand, `popLast` does wrap the returned value in an Optional, and is safe even if the array is empty.

Similar to `removeLast` and `popLast` are `removeFirst` and `popFirst`. Alternate forms `removeFirst(_:)` and `removeLast(_:)` allow you to specify how many elements to remove, but return no value; they, too, can crash if there aren't as many elements as you specify. `popFirst`, remarkably, operates on a slice, not an array, presumably for the sake of efficiency: all it has to do is increase the slice's `startIndex`, whereas with an array, the whole array must be renumbered.

Even if the reference is not mutable, you can use the `dropFirst` and `dropLast` methods to return a slice with the end element removed. Again, you can supply a parameter stating how many elements to drop. And again, there is no penalty for dropping too many elements; you simply end up with an empty slice.

If your intention is primarily to insert or remove elements at the start and end of an array — as in a FIFO queue (first in, first out) — consider instead using a Deque, which is optimized for that sort of usage. It's available from the Swift Collections package (`import Collections`; see Chapter 7).

The `joined(separator:)` instance method starts with an array of arrays. It extracts their individual elements, and interposes between each sequence of extracted elements the elements of the `separator:` array. The result is an intermediate sequence called a JoinSequence, which might have to be coerced further to an Array if that's what you were after:

```
let arr = [[1,2], [3,4], [5,6]]
let joined = Array(arr.joined(separator:[10,11]))
// [1, 2, 10, 11, 3, 4, 10, 11, 5, 6]
```

The `joined()` method is extended in the Swift Algorithms package so that the separator can be generated through a function that you supply. The function receives as its parameters two adjacent inner arrays, and returns the separator array whose elements will go between theirs:

```
let arr = [[1,2], [3,4], [5,6]]
let joined = arr.joined { [$0.last!*10, $1.first!*10] }
let result = Array(joined)
// [1, 2, 20, 30, 3, 4, 40, 50, 5, 6]
```

Calling `joined()` with no separator is a way to flatten an array of arrays. Again, it returns an intermediate sequence (or collection), so you might want to coerce to an Array:

```
let arr = [[1,2], [3,4], [5,6]]
let arr2 = Array(arr.joined())
// [1, 2, 3, 4, 5, 6]
```

The `split` instance method breaks an array into an array of slices. If you call `split(separator:)`, the parameter is an element value, and the array is broken at elements equatable to that parameter. If you call `split(isSeparator:)`, the parameter is a function that takes a value of the element type and returns a Bool, and the array is broken at elements for which the function returns `true`. The separator elements themselves are eliminated:

```
let arr = [1,2,3,4,5,6]
let arr2 = arr.split {$0.isMultiple(of:2)}
// split at evens, removing the evens: [[1], [3], [5]]
```

The Swift Algorithms package has a `chunked(by:)` method that lets you break up an array *without* eliminating any elements. You supply a function that inspects each successive pair of elements and returns a Bool; returning `true` means that the given next element should be chunked together with the given preceding element:

```
let arr = [1,2,3,4,5,6]
let chunks = arr.chunked {before, after in !after.isMultiple(of: 2)}
// split before evens, keeping the evens: [[1], [2, 3], [4, 5], [6]]
```

The `reverse` instance method reverses an array (if the reference to it is mutable). The `reversed` instance method yields a new array reversed from the original.

The `sort` and `sorted` instance methods respectively sort the original array (if the reference to it is mutable) and yield a new sorted array based on the original. Once again, you get two choices. If this is an array of Comparable elements, you can let the `<` operator dictate the new order. Alternatively, you can call `sort(by:)` or `sorted(by:)`; you supply a function that takes two parameters of the element type and returns a Bool stating whether the first parameter should be ordered before the second (just like `min` and `max`):

```
var arr = [4,3,5,2,6,1]
arr.sort() // [1, 2, 3, 4, 5, 6]
arr.sort {$0 > $1} // [6, 5, 4, 3, 2, 1]
```

In that last line, I provided an anonymous function. Alternatively, of course, you can pass as argument the name of a declared function. In Swift, comparison operators *are* the names of functions! Therefore, I can do the same thing like this:

```
var arr = [4,3,5,2,6,1]
arr.sort(by: >) // [6, 5, 4, 3, 2, 1]
```

An interesting problem is *subsorting* an array. Suppose we have a Person struct with a `firstName` and a `lastName`, and we have an array of Persons:

```
var arr = [
    Person(firstName: "Manny", lastName: "Pep"),
    Person(firstName: "Harpo", lastName: "Marx"),
    Person(firstName: "Jack", lastName: "Pep"),
    Person(firstName: "Groucho", lastName: "Marx")
]
```

We wish to sort this array by last name, but if two Persons have the *same* last name, they should be sorted by first name. In other words, all Marx brothers should precede all Pep boys, and within those groups, Groucho should precede Harpo, and Jack should precede Manny.

Cocoa provides an elegant solution, NSSortDescriptor. New in Swift 5.5, there's a Swift Foundation overlay type SortDescriptor (Chapter 11), but it's still just an overlay on Cocoa, effectively requiring that the array elements be Objective-C class instances. Instead, we could write our own comparison function; but that's an invitation to make a mistake, and won't scale if we're subsorting on many properties.

In a simple case like this one, where the comparison operator is the same for all properties, there's a trick: use *tuples*. It turns out that tuples, by default, are comparable if their element types are comparable, by a rule that a given element is compared only if all preceding elements are equal. That's exactly how we want to sort this array! So we can sort the array in accordance with the corresponding tuple:

```
arr.sort { ($0.lastName, $0.firstName) < ($1.lastName, $1.firstName) }
```

The `swapAt` method accepts two Int index numbers and interchanges those elements of a mutable array:

```
var arr = [1,2,3]
arr.swapAt(0,2) // [3,2,1]
```

The `shuffle` and `shuffled` methods sort an array in random order. The `randomElement` method generates a valid index at random and hands you the element at that index (wrapped in an Optional, in case the array is empty).To generate an array of elements chosen randomly from an array, the Swift Algorithms package provides `randomSample` and `randomStableSample` methods.

There are even more utility methods in the Swift Algorithms package. There are methods for chaining arrays or repeating an array without allocating new storage. You can obtain an array's combinations and permutations. You can rotate an array. You can partition an array into two subarrays of which a given predicate is false and true, respectively, maintaining the element order:

```
var arr = [1,2,3,4,5,6,7]
let ix = arr.stablePartition {$0.isMultiple(of: 2)} // odds followed by evens
let odds = arr[..<ix] // [1, 3, 5, 7]
let evens = arr[ix...] // [2, 4, 6]
```

You can also generate tuples of adjacent pairs (`adjacentPairs`) or successive groups of a given length (`windows(ofCount:)`); alternatively, you can chunk an array into smaller arrays by length. An example will demonstrate the difference between windows and chunks:

```
let arr = [1,2,3,4,5]
let windows = arr.windows(ofCount: 3)
// [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
let chunks = arr.chunks(ofCount: 3)
// [[1, 2, 3], [4, 5]]
```

Array enumeration and transformation

An array is a sequence, and so you can enumerate it, inspecting or operating with each element in turn. The simplest way is by means of a `for...in` loop; I'll have more to say about this construct in Chapter 5:

```
let pepboys = ["Manny", "Moe", "Jack"]
for pepboy in pepboys {
    print(pepboy) // prints Manny, then Moe, then Jack
}
```

Alternatively, you can use the `forEach(_:)` instance method. Its parameter is a function that takes an element and returns no value. Think of it as the functional equivalent of the imperative `for...in` loop:

```
let pepboys = ["Manny", "Moe", "Jack"]
pepboys.forEach {print($0)} // prints Manny, then Moe, then Jack
```

If you need index numbers as well as elements, call the `enumerated` instance method and loop on the result; what you get on each iteration is a tuple with labels `offset` and `element`:

```
let pepboys = ["Manny", "Moe", "Jack"]
for (ix,pepboy) in pepboys.enumerated() {
    print("Pep boy \(ix) is \(pepboy)") // Pep boy 0 is Manny, etc.
}
// or:
pepboys.enumerated().forEach {
    print("Pep boy \($0.offset) is \($0.element)")
}
```

The `offset` member of the tuple is actually just a counter that starts at zero. In some situations, the start index might not be zero. If you need the *real* index number, use the `indexed` method, available from the Swift Algorithms package.

The `allSatisfy(_:)` method tells you whether all elements pass some test; you supply a function that takes an element and returns a Bool:

```
let pepboys = ["Manny", "Moe", "Jack"]
let ok = pepboys.allSatisfy {$0.hasPrefix("M")} // false
let ok2 = pepboys.allSatisfy {$0.hasPrefix("M") || $0.hasPrefix("J")} // true
```

Swift also provides some powerful array transformation instance methods. Like `forEach(_:)` and `allSatisfy(_:)`, these methods enumerate the array for you, so that the loop is buried implicitly inside the method call, making your code tighter and cleaner.

The `filter(_:)` instance method yields a new array, each element of which is an element of the old array, in the same order; but some of the elements of the old array may be omitted — they were filtered out. What filters them out is a function that you supply; it accepts a parameter of the element type and returns a Bool stating whether this element should go into the new array:

```
let pepboys = ["Manny", "Moe", "Jack"]
let pepboys2 = pepboys.filter {$0.hasPrefix("M")} // ["Manny", "Moe"]
```

If the function is effectively negative, and if the reference to the collection is mutable, you should call `removeAll(where:)` rather whan `filter(_:)`:

```
var pepboys = ["Manny", "Jack", "Moe"]
pepboys.removeAll {$0.hasPrefix("M")} // pepboys is now ["Jack"]
```

That's better in general than saying `pepboys.filter {!$0.hasPrefix("M")}` because of efficiencies achieved under the hood.

Similar to `filter(_:)` is `prefix(while:)`. The difference is that `prefix(while:)` stops looping as soon as it encounters an element for which the supplied function returns `false`; it returns the start of the original array as a slice. The complement of `prefix(while:)` is `drop(while:)`; it stops where `prefix(while:)` stops, but it returns the *rest* of the original array as a slice:

```
let pepboys = ["Manny", "Jack", "Moe"]
let arr1 = pepboys.filter {$0.hasPrefix("M")} // ["Manny", "Moe"]
let arr2 = pepboys.prefix {$0.hasPrefix("M")} // ["Manny"]
let arr3 = pepboys.drop {$0.hasPrefix("M")} // ["Jack", "Moe"]
```

The Swift Algorithms package adds `suffix(while:)`, `trimmingPrefix(while:)`, `trimmingSuffix(while:)`, and `trimming(while:)`. These methods are also applicable to strings.

The `map(_:)` instance method yields a new array, each element of which is the result of passing the corresponding element of the old array through a function that you supply. This function accepts a parameter of the element type and returns a result which may be of some other type; Swift can usually infer the type of the resulting array elements by looking at the type returned by the function.

Here's how to multiply every element of an array by 2:

```
let arr = [1,2,3]
let arr2 = arr.map {$0 * 2} // [2,4,6]
```

Here's another example, to illustrate the fact that `map(_:)` can yield an array with a different element type:

```
let arr = [1,2,3]
let arr2 = arr.map {Double($0)} // [1.0, 2.0, 3.0]
```

Here's a real-life example showing how neat and compact your code can be when you use `map(_:)`. In order to remove all the table cells in a section of a UITableView, I have to specify the cells as an array of IndexPath objects. If `sec` is the section number, I can form those IndexPath objects individually like this:

```
let path0 = IndexPath(row:0, section:sec)
let path1 = IndexPath(row:1, section:sec)
// ...
```

Hmmm, I think I see a pattern here! I could generate my array of IndexPath objects by looping through the row values using `for...in`. But with `map(_:)`, there's a much tighter way to express the same loop — namely, to loop through the range `0..<ct` (where `ct` is the number of rows in the section). Since `map(_:)` is a Collection instance method, and a Range is itself a Collection, I can call `map(_:)` directly on the range:

```
let paths = (0..<ct).map {IndexPath(row:$0, section:sec)}
```

The `map(_:)` method provides a neat alternative to `init(repeating:count:)` with a reference type:

```
let dogs = Array(repeating:Dog(), count:3) // probably a mistake
```

You probably wanted an array of three Dogs. But if Dog is a class, the array consists of three references to *one and the same* Dog instance! Instead, generate the array using `map(_:)`, like this:

```
let dogs = (0..<3).map {_ in Dog()}
```

The `map(_:)` method has a specialized companion, `flatMap(_:)`. Applied to an array, `flatMap(_:)` first calls `map(_:)`, and then, if the map function produces an array of arrays, flattens it. In the simplest case, the map function does nothing and `flatMap` just flattens the array:

```
let arr = [[1],[2]]
let flattened = arr.flatMap {$0} // [1,2]
```

Here's a more interesting example:

```
let arr = [[1, 2], [3, 4]]
let arr2 = arr.flatMap {$0.map {String($0)}} // ["1", "2", "3", "4"]
```

First our map function calls `map(_:)` to coerce the individual elements of each inner array to a string, yielding an array of arrays of String: `[["1", "2"], ["3", "4"]]`. Then `flatMap(_:)` flattens the array of arrays, and we end up with a single array of String.

Another specialized `map(_:)` companion is `compactMap(_:)`. Given a map function that produces an array of Optionals, `compactMap(_:)` safely unwraps them by first eliminating any `nil` elements. In the simple case where you already have an array of Optionals and you just want to unwrap them all while eliminating `nil` elements, you can write that as `compactMap{$0}`. The Swift Algorithms package provides a nice synonym for it, `compacted`.

`compactMap(_:)` neatly solves a type of problem that arises quite often, where we want to coerce or cast an array safely while eliminating those elements that *can't* be coerced or cast. Suppose I have a mixed bag of strings, some of which represent integers. I'd like to coerce to Int those that *can* be coerced to Int, and eliminate the others. Int coercion of a String yields an Optional, so the `compactMap(_:)` lightbulb should go on in our heads:

```
let arr = ["1", "hey", "2", "ho"]
let arr2 = arr.compactMap {Int($0)} // [1, 2]
```

First we map the original array to an array of Optionals wrapping Int, by coercing; that yields `[Optional(1), nil, Optional(2), nil]`. Then `compactMap(_:)` removes the `nil` elements and unwraps the remaining elements, resulting in an array of Int.

The Swift Algorithms package has a `firstNonNil` method that applies a map function yielding an Optional and returns the first mapped value that isn't `nil`:

```
if let result = ["hey", "1", "ho", "2"].firstNonNil {Int($0)} {
    // result is 1
}
```

The `reduce` instance method is a way of *combining* all the elements of an array (actually, a sequence) into a single value. This value's type — the result type — doesn't have to be the same as the array's element type. `reduce` takes two parameters:

- You supply, as the *second* parameter, a function that takes two parameters; the first is of the result type, the second is of the element type, and the function's result is your combination of those two parameters, as the result type. That result, on each iteration, becomes the function's *first* parameter in the *next* iteration, along with the next element of the array as the *second* parameter. In this way, the output of combining pairs accumulates, and the final accumulated value is the final output of the function.

- However, that doesn't explain where the first parameter for the *first* iteration comes from. The answer is that you have to supply it as the *first* parameter of the `reduce` call.

That will all be easier to understand with a simple example. Suppose we've got an array of Int. Then we can use `reduce` to sum the elements of the array. Here's some pseudocode where I've left out the first argument of the call, so that you can think about what it needs to be:

```
let sum = arr.reduce(/* ... */) {$0 + $1}
```

Each pair of parameters will be added together to get the first parameter (`$0`) on the next iteration. The second parameter on every iteration (`$1`) is a successive element of the array. Clearly we are just summing the elements, adding each element one by one to an accumulated total. So the remaining question is: What should the *first* element of the array be added to? We want the actual sum of all the elements, no more and no less; so the first element of the array should be added to `0`:

```
let arr = [1, 4, 9, 13, 112]
let sum = arr.reduce(0) {$0 + $1} // 139
```

The `+` operator is the name of a function of the required type, so here's another way to write the same thing:

```
let sum = arr.reduce(0, +)
```

There is also `reduce(into:)`, which greatly improves efficiency when the goal is to build a collection such as an array or a dictionary. The `into:` argument is passed into your function as an `inout` parameter, and persists through each iteration; instead of returning a value, your function modifies it, and the final result is its final value.

Suppose we have an array of integers, and our goal is to "deal" them into two piles consisting of the even elements and the odd elements respectively. You can't do that with a single call to `map`; you'd have to cycle through the original array *twice*. With `reduce(into:)`, both target arrays are constructed while cycling through the original array once:

```
let nums = [1,3,2,4,5]
let result = nums.reduce(into: [[],[]]) { temp, i in
    temp[i%2].append(i)
}
// result is now [[2, 4], [1, 3, 5]]
```

The Swift Algorithms package supplies `reductions` methods corresponding to those `reduce` methods. The result is an array of the initial, intermediate, and final values that are accumulated as the `reduce` method loops through the array:

```
let reduct = [1,2,3,4].reductions(0,+) // [0, 1, 3, 6, 10]
```

Swift's array transformation methods are very powerful and very useful. In real life, your code is likely to depend heavily on methods like `filter`, `map`, and `reduce`, alone or in combination, nested or chained together.

Swift Array and Objective-C NSArray

When you're programming iOS, you import the Foundation framework (or UIKit, which imports Foundation) and the Objective-C NSArray type. Swift Array is bridged to Objective-C NSArray.

The most general medium of array interchange between Objective-C and Swift is `[Any]`; if an Objective-C API specifies an NSArray, with no further type information, Swift will see this as an array of Any. This reflects the fact that Objective-C's rules for what can be an element of an NSArray are looser than Swift's: the elements of an NSArray do not all have to be of the same type. On the other hand, the elements of an Objective-C NSArray must be Objective-C *objects* — that is, they must be class types.

Passing a Swift array to Objective-C is usually easy. Typically, you'll just pass the array, either by assignment or as an argument in a function call:

```
let arr = [UIBarButtonItem(), UIBarButtonItem()]
self.navigationItem.leftBarButtonItems = arr
```

The objects that you pass as elements of the array will cross the bridge to Objective-C in the usual way:

```
let lay = CAGradientLayer()
lay.locations = [0.25, 0.5, 0.75] // bridged to NSArray of NSNumber
```

CAGradientLayer's `locations` property needs to be an array of NSNumber; but we can pass an array of Double, because Double is bridged to NSNumber, and so Objective-C receives an NSArray of NSNumber.

To call an NSArray method on a Swift array, you may have to cast to NSArray:

```
let arr = ["Manny", "Moe", "Jack"]
let s = (arr as NSArray).componentsJoined(by:", ")
// s is "Manny, Moe, Jack"
```

A Swift Array seen through a `var` reference is mutable, but an NSArray isn't mutable ever. For mutability in Objective-C, you need an NSMutableArray, a subclass of NSArray. You can't cast, assign, or pass a Swift array as an NSMutableArray; you have to coerce. The best way is to call the NSMutableArray initializer `init(array:)`, to which you can pass a Swift array directly. To convert back from an NSMutable-Array to a Swift array, you can cast:

```
var arr = ["Manny", "Moe", "Jack"]
let arr2 = NSMutableArray(array:arr)
arr2.remove("Moe")
arr = arr2 as! [String]
```

Now let's talk about what happens when an NSArray arrives from Objective-C into Swift. There won't be any problem crossing the bridge: the NSArray will arrive safely as a Swift Array. But a Swift Array *of what?*

Of itself, an NSArray carries no information about what type of element it contains. Starting in Xcode 7, however, the Objective-C language was modified so that the declaration of an NSArray, NSDictionary, or NSSet — the three collection types that are bridged to Swift — can include element type information. (Objective-C calls this a *lightweight generic*.) Thus, for the most part, the arrays you receive from Cocoa will be correctly typed.

For example, this elegant code was impossible in the bad old days before Xcode 7:

```
let arr = UIFont.familyNames.map {
    UIFont.fontNamesForFamilyName($0)
}
```

The result is an array of arrays of String, listing all available fonts grouped by family. That code is possible because both of those UIFont class methods are seen by Swift as returning an array of String. Before Xcode 7, however, those arrays were untyped, and casting down to an array of String was up to you.

Nevertheless, lightweight generics are not omnipresent. You might read an array from a *.plist* file stored on disk with NSArray's initializer `init(contentsOf:)`; you might retrieve an array from UserDefaults; you might even be dealing with an Objective-C API that hasn't been updated to use lightweight generics. In such a situation, you're going to end up with a plain vanilla NSArray or a Swift array of Any. If that happens, you will usually want to cast down or otherwise transform this array into an array of some specific Swift type. Here's an Objective-C class containing a method whose return type of NSArray hasn't been marked up with an element type:

```
@implementation Pep
- (NSArray*) boys {
    return @[@"Manny", @"Moe", @"Jack"];
}
@end
```

To call that method and do anything useful with the result, it will be necessary to cast that result down to an array of String. If I'm sure of my ground, I can force the cast:

```
let p = Pep()
let boys = p.boys() as! [String]
```

As with any cast, though, be sure you don't lie! An Objective-C array can contain more than one type of object. Don't force such an array to be cast down to a type to which not all the elements can be cast, or you'll crash when the cast fails; you'll need a more deliberate strategy (possibly involving `compactMap`) for eliminating or otherwise transforming the problematic elements.

Dictionary

A dictionary (Dictionary, a struct) is an unordered collection of object pairs. In each pair, the first object is the *key*; the second object is the *value*. The idea is that you use a key to access a value.

Keys are usually strings, but they don't have to be; the formal requirement is that they be types that conform to the Hashable protocol. For a type to be Hashable, three things are required:

- The type must be Equatable.
- The type must implement an Int `hashValue` property.
- The type's implementation of equality and `hashValue` must be such that *equal keys have equal hash values*. The protocol itself cannot formally insist upon this rule, but that is what is needed for hashability to be useful and well-behaved.

The hash values can then be used behind the scenes for rapid key access. Most Swift standard types are Hashable; I'll talk in Chapter 5 about how to make your own object types Hashable.

Do not use mutable objects as keys. Mutating a key while it is in use will break the dictionary (lookup will no longer work correctly).

As with arrays, the types of a given Swift Dictionary must be uniform. The key type and the value type don't have to be the same as one another, and they often will not be. But within any dictionary, all keys must be of the same type, and all values must be of the same type.

Formally, a dictionary is a generic, and its placeholder types are its key type and its value type: `Dictionary<Key,Value>`. As with arrays, Swift provides syntactic sugar for expressing a dictionary's type, and that is what you'll usually use: `[Key: Value]`. That's square brackets containing a colon (and optional spaces) separating the key type from the value type. This code creates an empty dictionary whose keys (when they exist) will be Strings and whose values (when they exist) will be Strings:

```
var d = [String:String]()
```

The colon is used also between each key and value in the literal syntax for expressing a dictionary. The key–value pairs appear between square brackets, separated by a comma, just like an array. This code creates a dictionary by describing it literally (and the dictionary's type of `[String:String]` is inferred):

```
var d = ["CA": "California", "NY": "New York"]
```

But if a dictionary variable is declared and initialized to a literal with many elements, it's a good idea to declare the variable's type explicitly. This saves the compiler from having to examine the entire dictionary to decide its type, and makes compilation faster.

The literal for an empty dictionary is square brackets containing just a colon: `[:]`. That notation can be used provided the dictionary's type is known in some other way. This is another way to create an empty `[String:String]` dictionary:

```
var d : [String:String] = [:]
```

You can also initialize a dictionary from a sequence of key–value tuples. This is useful particularly if you're starting with two sequences. Suppose we happen to have state abbreviations in one array and state names in another:

```
let abbrevs = ["CA", "NY"]
let names = ["California", "New York"]
```

We can combine those two arrays into a single array of tuples and call `init(uniqueKeysWithValues:)` to generate a dictionary:

```
let tuples = (abbrevs.indices).map {(abbrevs[$0],names[$0])}
let d = Dictionary(uniqueKeysWithValues: tuples)
// ["NY": "New York", "CA": "California"]
```

There is actually a simpler way to form those tuples — the global `zip` function, which takes two sequences and yields a sequence of tuples:

```
let tuples = zip(abbrevs, names)
let d = Dictionary(uniqueKeysWithValues: tuples)
```

A nice feature of `zip` is that if one sequence is longer than the other, the extra elements of the longer sequence are ignored — tuple formation simply stops when the end of the shorter sequence is reached. For example, one of the zipped sequences can be a partial range; in theory the range is infinite, but in fact the end of the other sequence ends the range as well:

```
let r = 1...
let names = ["California", "New York"]
let d = Dictionary(uniqueKeysWithValues: zip(r,names))
// [2: "New York", 1: "California"]
```

If the keys in the tuple sequence are not unique, you'll crash at runtime when `init(uniqueKeysWithValues:)` is called. To work around that, you can use `init(_:uniquingKeysWith:)` instead. The second parameter is a function taking two values — the existing value for this key, and the new incoming value for the same key — and returning the value that should actually be used for this key.

Another way to form a dictionary is `init(grouping:by:)`. This is useful for forming a dictionary whose values are *arrays*. You start with a sequence of the *elements* of the arrays, and the initializer clumps them into arrays for you, in accordance with a function that generates the corresponding key from each value.

Suppose we have a list (`states`) of the 50 U.S. states in alphabetical order as an array of strings, and we want to group them by the letter they start with. Here's a verbose strategy. We loop through the list to construct two arrays (an array of String and an array of arrays of String); we then zip those arrays together to form the dictionary:

```
var sectionNames = [String]()
var cellData = [[String]]()
var previous = ""
for aState in states {
    // get the first letter
    let c = String(aState.prefix(1))
    // only add a letter to sectionNames when it's a different letter
    if c != previous {
        previous = c
        sectionNames.append(c.uppercased())
        // and in that case also add new subarray to our array of subarrays
        cellData.append([String]())
    }
    cellData[cellData.count-1].append(aState)
}
let d = Dictionary(uniqueKeysWithValues: zip(sectionNames,cellData))
// ["H": ["Hawaii"], "V": ["Vermont", "Virginia"], ...
```

With `init(grouping:by:)`, however, that becomes effectively a one-liner:

```
let d = Dictionary(grouping: states) {$0.prefix(1).uppercased()}
```

Dictionary subscripting

Access to a dictionary's contents is usually by subscripting. To fetch a value by key, use the key as a subscript:

```
let d = ["CA": "California", "NY": "New York"]
let state = d["CA"]
```

If you try to fetch a value through a nonexistent key, there is no error, but Swift needs a way to report failure; to do so, by default, it returns `nil`. This, in turn, implies that the value returned when you successfully access a value through a key must be an Optional wrapping the real value. After that code, therefore, `state` is not a String — it's an Optional wrapping a String! Forgetting this is a common beginner mistake.

You can change that behavior by supplying your own `default` value as part of the subscript. If the key isn't found in the dictionary, the `default` value is returned, and so there is no need for the returned value to be wrapped in an Optional:

```
let d = ["CA": "California", "NY": "New York"]
let state = d["MD", default:"N/A"] // state is a String (not an Optional)
```

If the reference to a dictionary is mutable, you can also assign into a key subscript expression. If the key already exists, its value is replaced. If the key doesn't already exist, it is created and the value is attached to it:

```
var d = ["CA": "California", "NY": "New York"]
d["CA"] = "Casablanca"
d["MD"] = "Maryland"
// d is now ["MD": "Maryland", "NY": "New York", "CA": "Casablanca"]
```

Instead of assigning into a subscript expression, you can call `updateValue(_:forKey:)`; it has the advantage that it returns the old value.

As with fetching a value by key, you can supply a `default` value when assigning into a key subscript expression. This can be a source of great economy of expression. Consider the common task of collecting a histogram: we want to know how many times each element appears in a sequence:

```
let sentence = "how much wood would a wood chuck chuck"
let words = sentence.split(separator: " ").map {String($0)}
```

Our goal is now to make a dictionary pairing each word with the number of times it appears. A manual approach would be rather laborious, along these lines:

```
var d = [String:Int]()
for word in words {
    let ct = d[word]
    if ct != nil {
        d[word]! += 1
    } else {
        d[word] = 1
    }
}
// d is now ["how": 1, "wood": 2, "a": 1, "chuck": 2, "would": 1, "much": 1]
```

With a default value, it's effectively a one-liner:

```
var d = [String:Int]()
words.forEach {d[$0, default:0] += 1}
```

By a kind of shorthand, assigning `nil` into a key subscript expression removes that key–value pair if it exists:

```
var d = ["CA": "California", "NY": "New York"]
d["NY"] = nil // d is now ["CA": "California"]
```

Alternatively, call `removeValue(forKey:)`; it has the advantage that it returns the removed value before it removes the key–value pair.

Dictionaries have no order

Dictionaries are *unordered*. Whenever you probe a dictionary's entire contents — when you `print` them, when you cycle through them with `for...in`, and so forth — each entry arrives in a completely unpredictable order. If you run the same code as I do (indeed, even if you run the same code as yourself on two different occasions), your results may be ordered differently. This makes no difference to the actual contents of the dictionary, which consists of particular keys, each associated with a particular value. `[2: "New York", 1: "California"]` is actually the same dictionary as `[1: "California", 2: "New York"]`.

If you needed order to be meaningful, you may have been thinking of an array, not a dictionary. For an ordered array of key-value pairs — with no subscripting by key, no hashability requirement, and no guaranteed uniqueness of keys — you can use a Key-ValuePairs object, which is essentially an array of tuples with labels `key` and `value`; it can be initialized from a dictionary literal:

```
let pairs : KeyValuePairs = ["CA": "California", "NY": "New York"]
print(pairs.count) // 2
print(pairs[0]) // (key: "CA", value: "California")
// to access by key, cycle through the array
if let pair = pairs.first(where: {$0.key == "NY"}) {
    let val = pair.value // New York
}
```

On the other hand, there is an OrderedDictionary struct in the Swift Collections package; I'll talk about it later.

Dictionary casting and comparison

As with arrays, a dictionary type is legal for casting down, meaning that the individual elements will be cast down. Typically, only the value types will differ:

```
let dog1 : Dog = NoisyDog()
let dog2 : Dog = NoisyDog()
let d = ["fido": dog1, "rover": dog2]
let d2 = d as! [String : NoisyDog]
```

As with arrays, `is` can be used to test the actual types in the dictionary, and `as?` can be used to test and cast safely.

Dictionary equality is like array equality. Key types are necessarily Equatable, because they are Hashable. Value types are not necessarily Equatable, but if they are, the `==` and `!=` operators work as you would expect.

Basic dictionary properties and enumeration

A dictionary has a `count` property reporting the number of key–value pairs it contains, and an `isEmpty` property reporting whether that number is `0`.

A dictionary has a `keys` property reporting all its keys, and a `values` property reporting all its values. These are effectively opaque structs providing a specialized view of the dictionary itself. You can't assign one to a variable, or print it out, but you can work with them as collections. For example, you can enumerate them with `for...in` (though you should not expect them to arrive in any particular order, as a dictionary is unordered):

```
var d = ["CA": "California", "NY": "New York"]
for s in d.keys {
    print(s) // NY, then CA (or vice versa)
}
```

You can coerce them to an array:

```
var d = ["CA": "California", "NY": "New York"]
var keys = Array(d.keys) // ["NY", "CA"] or ["CA", "NY"]
```

You can sort them, filter them, or map them (yielding an array); you can take their `min` or `max`; you can `reduce` them; you can compare `keys` of different dictionaries for equality:

```
let d : [String:Int] = ["one":1, "two":2, "three":3]
let keysSorted = d.keys.sorted() // ["one", "three", "two"]
let arr = d.values.filter {$0 < 2} // [1]
let min = d.values.min() // Optional(1)
let sum = d.values.reduce(0, +) // 6
let ok = d.keys == ["one":1, "three":3, "two":2].keys // true
```

You can also enumerate a dictionary itself. Each iteration provides a key–value tuple (arriving in no particular order, because a dictionary is unordered):

```
var d = ["CA": "California", "NY": "New York"]
for (abbrev, state) in d {
    print("\(abbrev) stands for \(state)")
}
```

The tuple members have labels `key` and `value`, and `forEach` works on a dictionary, so the preceding example can be rewritten like this:

```
var d = ["CA": "California", "NY": "New York"]
d.forEach { print("\($0.key) stands for \($0.value)") }
```

You can extract a dictionary's entire contents at once as an array of key–value tuples (in an unpredictable order) by coercing the dictionary to an array:

```
var d = ["CA": "California", "NY": "New York"]
let arr = Array(d)
// [(key: "NY", value: "New York"), (key: "CA", value: "California")]
```

When you apply `filter` to a dictionary, what you get is a dictionary. In addition, there's a `mapValues` method that yields a dictionary with its values changed according to your map function:

```
let d = ["CA": "California", "NY": "New York"]
let d2 = d.filter {$0.value > "New Jersey"}.mapValues {$0.uppercased()}
// ["NY": "NEW YORK"]
```

There's also a `compactMapValues` method that applies a map function yielding an Optional and filters out any keys for which the resulting value is `nil`.

You can combine two dictionaries with the merging(_:uniquingKeysWith:) method — or, if your reference to the first dictionary is mutable, you can call merge to modify it directly. The second parameter is like the second parameter of init(_:uniquingKeysWith:), saying what the value should be in case the second dictionary has a key matching an existing key in the first dictionary:

```
let d1 = ["CA": "California", "NY": "New York"]
let d2 = ["MD": "Maryland", "NY": "New York"]
let d3 = d1.merging(d2) {orig, _ in orig}
// ["MD": "Maryland", "NY": "New York", "CA": "California"]
```

Swift Dictionary and Objective-C NSDictionary

The Foundation framework dictionary type is NSDictionary, and Swift Dictionary is bridged to it. The untyped API characterization of an NSDictionary in Swift will be [AnyHashable:Any]. (AnyHashable is a *type eraser* struct, to cope with the possibility, legal in Objective-C, that the keys may be of different Hashable types.)

Like NSArray element types, NSDictionary key and value types can be marked in Objective-C using a lightweight generic. The most common key type in a real-life Cocoa NSDictionary is NSString, so you might well receive an NSDictionary typed as [String:Any]. Specific typing of an NSDictionary's values is rare, because dictionaries that you pass to and receive from Cocoa will often have values of multiple types; it is not surprising to have a dictionary whose keys are strings but whose values include a string, a number, a color, and an array. For this reason, you will usually *not* cast down the entire dictionary's type; instead, you'll work with the dictionary as having Any values, and cast when fetching an *individual value* from the dictionary. Since the value returned from subscripting a key is itself an Optional, you will typically unwrap and cast the value as a standard single move.

Here's an example. A Cocoa Notification object comes with a userInfo property. It is an NSDictionary that might itself be nil, so the Swift API characterizes it as [AnyHashable:Any]?. Let's say I'm expecting this dictionary to be present and to contain a "progress" key whose value is an NSNumber containing a Double. My goal is to extract that NSNumber and assign the Double that it contains to a property, self.progress. Here's one way to do that safely, using optional unwrapping and optional casting (n is the Notification object):

```
let prog = n.userInfo?["progress"] as? Double
if prog != nil {
    self.progress = prog!
}
```

The variable prog is implicitly typed as an Optional wrapping a Double. The code is safe, because if there is no userInfo dictionary, or if it doesn't contain a "progress" key, or if that key's value isn't a Double, nothing happens, and prog will be nil.

I then test `prog` to see whether it *is* `nil`; if it isn't, I know that it's safe to force-unwrap it, and that the unwrapped value is the Double I'm after.

(In Chapter 5 I'll describe another syntax for accomplishing the same goal, using conditional binding.)

Conversely, here's a typical example of creating a dictionary and handing it off to Cocoa. This dictionary is a mixed bag: its values are a UIFont, a UIColor, and an NSShadow. Its keys are all strings, which I obtain as constants from Cocoa. I form the dictionary as a literal and pass it, all in one move, with no need to cast anything:

```
UINavigationBar.appearance().titleTextAttributes = [
    .font: UIFont(name: "ChalkboardSE-Bold", size: 20)!,
    .foregroundColor: UIColor.darkText,
    .shadow.: {
        let shad = NSShadow()
        shad.shadowOffset = CGSize(width:1.5,height:1.5)
        return shad
    }()
]
```

As with NSArray and NSMutableArray, if you want Cocoa to mutate a dictionary, you must coerce to NSDictionary's subclass NSMutableDictionary:

```
var d1 = ["NY":"New York", "CA":"California"]
let d2 = ["MD":"Maryland"]
let mutd1 = NSMutableDictionary(dictionary:d1)
mutd1.addEntries(from:d2)
d1 = mutd1 as! [String:String]
// d1 is now ["MD": "Maryland", "NY": "New York", "CA": "California"]
```

Set

A set (Set, a struct) is an *unordered* collection of *unique* objects. Its elements must be all of one type; it has a `count` and an `isEmpty` property; it can be initialized from any sequence; you can cycle through its elements with `for...in` (where the order of elements is unpredictable).

The uniqueness of set elements is implemented by constraining their type to be Hashable (and hence Equatable), just like the keys of a dictionary, so that the hash values can be used behind the scenes for rapid access. Checking whether a set contains a given element, which you can do with the `contains(_:)` instance method, is *very* efficient — far more efficient than doing the same thing with an array. Therefore, if element uniqueness is acceptable (or desirable) and you don't need indexing or a guaranteed order, a set can be a much better choice of collection than an array. I'll talk in Chapter 5 about how to make your own types Hashable.

The warnings in the preceding section apply: don't store a mutable value in a Set. Also, don't expect Set values to be reported to you in any particular order. (An OrderedSet struct is available from the Swift Collections package; I'll talk about it later.)

There are no set literals in Swift, but you won't need them because you can pass an array literal where a set is expected. There is no syntactic sugar for expressing a set type, but the Set struct is a generic, so you can express the type by explicitly specializing the generic:

```
let set : Set<Int> = [1, 2, 3, 4, 5]
```

In that particular example there was no real need to specialize the generic, as the Int type can be inferred from the array. However, when setting a Set variable from a literal, it is more efficient to specialize the generic. This saves the compiler the trouble of reading the whole literal.

It sometimes happens (more often than you might suppose) that you want to examine one element of a set as a kind of sample. Order is meaningless, so it's sufficient to obtain *any* element. For this purpose, use the `first` instance property; it returns an Optional, just in case the set is empty.

The distinctive feature of a set is the uniqueness of its objects. If an object is added to a set and that object is already present, it isn't added a second time. Conversion from an array to a set and back to an array is a quick and reliable way of *uniquing* the array — though of course order is not preserved:

```
let arr = [1,2,1,3,2,4,3,5]
let set = Set(arr)
let arr2 = Array(set) // [5, 2, 3, 1, 4], perhaps
```

A set is a Collection and a Sequence. Like Array, Set has a `map(_:)` instance method; it returns an array, but of course you can turn that right back into a set if you need to:

```
let set : Set = [1,2,3,4,5]
let set2 = Set(set.map {$0+1}) // Set containing 2, 3, 4, 5, 6
```

On the other hand, applying `filter` to a Set yields a Set directly:

```
let set : Set = [1,2,3,4,5]
let set2 = set.filter {$0 > 3} // Set containing 4, 5
```

If the reference to a set is mutable, you can add an object to it with `insert(_:)`; there is no penalty for trying to add an object that's already in the set, but the object won't be added. `insert(_:)` also returns a result, a tuple whose `inserted` element will be `false` if an equivalent object was already present in the set. This result is usually disregarded, but it can sometimes be useful; here, we use it to unique an array while preserving its order:

```
var arr = ["Manny", "Manny", "Moe", "Jack", "Jack", "Moe", "Manny"]
var temp = Set<String>()
arr = arr.filter { temp.insert($0).inserted }
```

It might be preferable, though, to use the `uniqued` or `uniqued(on:)` method from the Swift Algorithms package.

The other element of the tuple returned by `insert(_:)` is `memberAfterInsert`. If `inserted` is `true`, this is simply the parameter of `insert(_:)`. If `inserted` is `false`, however, it is the existing member of the set that caused the insertion to fail because it is regarded as equal to the parameter of `insert(_:)`; this may be of interest because it tells you *why* the insertion was rejected.

Instead of `insert(_:)`, you can call `update(with:)`; the difference is that if you're trying to add an object that already has an equivalent in the set, the former doesn't insert the new object, but the latter *always* inserts, replacing the old object (if there is one) with the new one.

You can remove an object and return it by specifying an equivalent object with the `remove(_:)` method; it returns the object from the set, wrapped in an Optional, or `nil` if the object was not present. You can remove and return an arbitrary object from the set with `removeFirst`; it crashes if the set is empty, so take precautions — or use `popFirst`, which is safe.

Equality comparison (==) is defined for sets as you would expect; two sets are equal if every element of each is equal to an element of the other.

If the notion of a set evokes visions of Venn diagrams from elementary school, that's good, because sets have instance methods giving you all those set operations you remember so fondly. The parameter can be a set, or it can be any sequence, which will be converted to a set; it might be an array, a range, or even a character sequence:

`intersection(_:)`, `formIntersection(_:)`
: Yields the elements of this set that also appear in the parameter. The first forms a new Set; the second is mutating.

`union(_:)`, `formUnion(_:)`
: Yields the elements of this set along with the (unique) elements of the parameter. The first forms a new Set; the second is mutating.

`symmetricDifference(_:)`, `formSymmetricDifference(_:)`
: Yields the elements of this set that don't appear in the parameter, plus the (unique) elements of the parameter that don't appear in this set. The first forms a new Set; the second is mutating.

`subtracting(_:)`, `subtract(_:)`
: Yields the elements of this set except for those that appear in the parameter. The first forms a new Set; the second is mutating.

`isSubset(of:)`, `isStrictSubset(of:)`
`isSuperset(of:)`, `isStrictSuperset(of:)`
: Returns a Bool reporting whether the elements of this set are respectively embraced by or embrace the elements of the parameter. The "strict" variant yields `false` if the two sets consist of the same elements.

`isDisjoint(with:)`
: Returns a Bool reporting whether this set and the parameter have no elements in common.

Here's a real-life example of Set usage from one of my apps. I have a lot of numbered pictures, of which we are to choose one randomly. But I don't want to choose a picture that has recently been chosen. Therefore, I keep a list of the numbers of all recently chosen pictures. When it's time to choose a new picture, I convert the list of all possible numbers to a Set, convert the list of recently chosen picture numbers to a Set, and call `subtracting(_:)` to get a list of unused picture numbers! Now I choose a picture number at random and add it to the list of recently chosen picture numbers:

```
let ud = UserDefaults.standard
let recents = ud.object(forKey: Defaults.recents) as? [Int] ?? []
var forbiddenNumbers = Set(recents)
let legalNumbers = Set(1...PIXCOUNT).subtracting(forbiddenNumbers)
let newNumber = legalNumbers.randomElement()!
forbiddenNumbers.insert(newNumber)
ud.set(Array(forbiddenNumbers), forKey:Defaults.recents)
```

Option sets

An *option set* (OptionSet struct) is Swift's way of treating a certain type of Cocoa enumeration as a Swift struct. It is not, strictly speaking, a Set; but it is deliberately set-like, sharing common features with Set through the SetAlgebra protocol. An option set has `contains(_:)`, `insert(_:)`, and `remove(_:)` methods, along with all the various set operation methods.

The purpose of option sets is to help you grapple with Objective-C *bitmasks*. A bitmask is an integer whose bits are used as switches when multiple options are to be specified simultaneously. Bitmasks are very common in Cocoa. In Objective-C, bitmasks are manipulated through the arithmetic bitwise-or and bitwise-and operators. Such manipulation can be mysterious and error-prone. But in Swift, thanks to option sets, bitmasks can be manipulated easily through set operations instead.

For example, when specifying how a UIView is to be animated, you are allowed to pass an `options:` argument whose value comes from the UIView.AnimationOptions enumeration, whose definition (in Objective-C) begins:

```
typedef NS_OPTIONS(NSUInteger, UIViewAnimationOptions) {
    UIViewAnimationOptionLayoutSubviews           = 1 << 0,
    UIViewAnimationOptionAllowUserInteraction     = 1 << 1,
    UIViewAnimationOptionBeginFromCurrentState    = 1 << 2,
    UIViewAnimationOptionRepeat                   = 1 << 3,
    UIViewAnimationOptionAutoreverse              = 1 << 4,
    // ...
};
```

Pretend that an NSUInteger is 8 bits (it isn't, but let's keep things simple and short). Then this enumeration means that (in Swift) the following name–value pairs are defined:

```
UIView.AnimationOptions.layoutSubviews          0b00000001
UIView.AnimationOptions.allowUserInteraction    0b00000010
UIView.AnimationOptions.beginFromCurrentState   0b00000100
UIView.AnimationOptions.repeat                  0b00001000
UIView.AnimationOptions.autoreverse             0b00010000
```

These values can be combined into a single value — a *bitmask* — that you pass as the `options:` argument for your animation. All Cocoa has to do to understand your intentions is to look to see which bits in the value that you pass are set to 1. So, for example, `0b00011000` would mean that `UIView.AnimationOptions.repeat` and `UIView.AnimationOptions.autoreverse` are both true (and that the others are all false).

The question is how to *form* the value `0b00011000` in order to pass it. You could form it directly as a literal and set the `options:` argument to `UIView.Animation-Options(rawValue:0b00011000)`; but that's not a very good idea, because it's error-prone and makes your code incomprehensible. In Objective-C, you'd use the arithmetic bitwise-or operator, analogous to this Swift code:

```
let val =
    UIView.AnimationOptions.autoreverse.rawValue |
    UIView.AnimationOptions.repeat.rawValue
let opts = UIView.AnimationOptions(rawValue: val)
```

That's rather ugly! However, help is on the way: the UIView.AnimationOptions type is an option set struct in Swift (because it is marked as `NS_OPTIONS` in Objective-C), and therefore can be treated much like a Set. Given a UIView.AnimationOptions value, you can add an option to it using `insert(_:)`:

```
var opts = UIView.AnimationOptions.autoreverse
opts.insert(.repeat)
```

Alternatively, you can start with an array literal, just as if you were initializing a Set:

```
let opts : UIView.AnimationOptions = [.autoreverse, .repeat]
```

To indicate that no options are to be set, pass an empty option set (`[]`) or, where permitted, omit the `options:` parameter altogether.

Sometimes Cocoa hands *you* a bitmask, and you want to know whether a certain bit is set. In this example from a UITableViewCell subclass, the cell's `state` comes to us as a bitmask; we want to know about the bit indicating that the cell is showing its edit control. The Objective-C way is to extract the raw values and use the bitwise-and operator:

```
override func didTransition(to state: UITableViewCell.StateMask) {
    let editing = UITableViewCell.StateMask.showingEditControl.rawValue
    if state.rawValue & editing != 0 {
        // ... the ShowingEditControl bit is set ...
    }
}
```

That's a tricky formula, all too easy to get wrong. But in Swift this is an option set, so the `contains(_:)` method tells you the answer:

```
override func didTransition(to state: UITableViewCell.StateMask) {
    if state.contains(.showingEditControl) {
        // ... the ShowingEditControl bit is set ...
    }
}
```

Swift Set and Objective-C NSSet

Swift's Set type is bridged to Objective-C NSSet. The untyped medium of interchange is `Set<AnyHashable>`. Coming back from Objective-C, if Objective-C doesn't know what this is a set of, you would probably cast down as needed. As with NSArray, however, NSSet can be marked up using lightweight generics to indicate its element type, in which case no casting will be necessary:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    let t = touches.first // an Optional wrapping a UITouch
    // ...
}
```

OrderedSet and OrderedDictionary

The Swift Collections package provides an OrderedSet struct and an OrderedDictionary struct. These, as their names imply, allow a Set or a Dictionary to have a fixed order.

OrderedSet

OrderedSet is basically implemented as an array plus a hash table. Like an array, an OrderedSet has an order, and you can refer to elements consistently by index number. Unlike an array, elements of an OrderedSet must be unique.

Lookup in an OrderedSet, such as `firstIndex(of:)` and `contains`, is fast because it uses the hash table rather than having to walk the array examining every element. So if you were thinking of using an array and you need fast lookup, and if the uniqueness requirement is not a barrier, you might prefer an OrderedSet:

```
let pep = OrderedSet(["Manny", "Moe", "Jack"])
let first = pep.first! // Manny
let ix = pep.firstIndex(of: "Moe") // Optional(1), fast
if pep.contains("Jack") { // true, fast
```

OrderedSet conforms to the Collection and Sequence protocols, so it can do other array-like things, even if you don't get any special benefits. For example, `firstIndex(where:)` works fine, because OrderedSet is partly an array; but it doesn't work any differently than if it *were* an array.

Unlike a Set, you can't just `insert` into an OrderedSet; it has an order, like an array, so you must also provide an index saying *where* you want to insert this element, or you can `append` to the end. But it's still a Set, so if the proposed new element already exists, it won't be added! The `insert` and `append` methods return a tuple saying whether the element was added and, if so, at what index; if the element was not added, the index reports where the matching element is.

OrderedSet also has two properties that constitute special views of the data:

`elements`
: An OrderedSet's `elements` property is an Array, and is useful primarily when you need to pass the OrderedSet data where an array is expected. Direct operations upon the `elements` affect the OrderedSet (with the proviso that operations that would violate the Set's uniqueness contract will not be performed).

`unordered`
: An OrderedSet's `unordered` property is not a Set, but it acts like one; in particular, it obeys all the set algebra commands applicable to a Set, because it conforms to the SetAlgebra protocol.

When you assign the `elements` or `unordered` property of an OrderedSet to another variable, or pass it as a parameter, you have effectively made a copy. There's nothing wrong with that, but keep in mind that mutations on the resulting object will *not* affect the original ordered set:

```
var arr = myOrderedSet.elements
arr.append(100) // does not affect myOrderedSet
```

On the whole, you probably won't be mutating an OrderedSet's `elements` or `unordered` properties in the first place. Mutate the OrderedSet itself! Mutating operations on an OrderedSet are faster than the same operations on its `elements` property.

An OrderedSet can help solve a problem that frequently arises in connection with arrays: how to *cosort* an array — that is, sorting one array in an order dictated by another array. Suppose we have a `model` array with its elements in the desired order:

```
let model = ["Manny", "Moe", "Jack"]
```

And we have another array whose elements are drawn from the `model` array:

```
let arr = ["Jack", "Manny"]
```

The question is: how can we sort the second array so that its elements are ordered like the first array?

A naïve solution might involve a sort function that looks up the index of each element in the model array:

```
let result = arr.sorted {
    model.firstIndex(of:$0)! < model.firstIndex(of:$1)!
}
```

But `firstIndex` on a normal array is horribly inefficient, as we must walk the array one element at a time — and if these arrays are big, sorting could involve calling `firstIndex` many times, including multiple times on the same element.

Without OrderedSet, the standard solution is to start by making a dictionary of model element values and positions, to facilite rapid lookup. But an OrderedSet makes calling `firstIndex` just as efficient as dictionary lookup:

```
let model = OrderedSet(["Manny", "Moe", "Jack"])
```

When the arrays are large, even that becomes needlessly slow, because we are *still* calling `firstIndex` many times on each element. We can vastly improve efficiency if we start by mapping each element of the target array to a tuple of positions and values, so that we fetch each element's `firstIndex` only once. We then sort the tuples and map back to the original elements:

```
let result = arr.map {(model.firstIndex(of:$0)!, $0)}.sorted(by:<).map {$0.1}
```

OrderedDictionary

OrderedDictionary has most of the capabilities of a normal Dictionary, along with ordering of its key–value pairs. Its `keys` property is an immutable OrderedSet. Its `elements` property looks a lot like what you get when you coerce a Dictionary to an Array — it's a list of tuples with labels `key` and `value`, indexable by number (because it's a RandomAccessCollection).

Think of an OrderedDictionary as something that you sometimes treat as a dictionary and sometimes treat as an array:

As a dictionary
: You can index by key in the normal way. But unlike a normal dictionary, if you assign into a key that doesn't exist, the new pair is appended to the end by default; alternatively, you can call `updateValue(_:forKey:insertingAt:)` to dictate where the pair should be inserted if the key doesn't already exist.

As an array
: To access a key–value pair by index number, pass through the `elements` property so that the index number is not mistaken for a key. But for an obviously array-like operation, such as `sort`, `shuffle`, or `remove(at:)`, you don't need to use the `elements` property; the OrderedDictionary knows what you mean, and exposes its elements as key–value pairs directly.

To illustrate, here are some example commands (we assume the existence of a simple Planet struct):

```
var d : OrderedDictionary<String,Planet> = [:]
d["Mercury"] = Planet(distance: 57_900_000, diameter: 4_878, gravity: 0.38)
d["Venus"] = Planet(distance: 108_160_000, diameter: 12_104, gravity: 0.9)
d["Earth"] = Planet(distance: 149_600_000, diameter: 12_756, gravity: 1)
let names = d.keys // OrderedSet: ["Mercury", "Venus", "Earth"]
let thirdPlanet = d.elements[2] // tuple with `key` and `value`
let name = thirdPlanet.key // "Earth"
let mercuryGravity = d["Mercury"]?.gravity // Optional, 0.38
let planetX = Planet(distance: 100_000_000, diameter: 8_000, gravity: 0.8)
d.updateValue(planetX, forKey: "PlanetX", insertingAt: 2)
```

OrderedDictionary also has some ways of updating a value that don't exist for a normal Dictionary, but ought to:

`modifyValue(forKey:default:_:)`
`modifyValue(forKey:insertingDefault:at:_:)`
: The last parameter is a function that takes the existing value as an `inout` parameter.

I'll rewrite the earlier histogram example to use that syntax:

```
var histogram = OrderedDictionary<String, Int>()
let sentence = "how much wood would a wood chuck chuck"
let words = sentence.split(separator: " ").map {String($0)}
for word in words {
    histogram.modifyValue(forKey: word, default: 0) {$0 += 1}
}
// [how: 1, much: 1, wood: 2, would: 1, a: 1, chuck: 2]
```

And if we prefer our keys in alphabetical order, we would now say:

```
histogram.sort {$0.key < $1.key}
```