CHAPTER 3

Variables and Simple Types

A variable is a named "shoebox" whose contained value must be of a single well-defined type. Every variable must be explicitly and formally declared. To put a value into the shoebox, causing the variable name to *refer* to that value, you *assign* the value to the variable. The variable name becomes a *reference* to that value.

This chapter goes into detail about declaration and initialization of variables. It then discusses all the primary built-in Swift simple types. (I mean "simple" as opposed to collections; the primary built-in collection types are discussed at the end of Chapter 4.)

Variable Scope and Lifetime

A variable not only gives its referent a name; it also, by virtue of *where* it is declared, endows its referent with a particular scope (visibility) and lifetime. (See "Scope and Lifetime" on page 12.) Assigning a value to a variable is a way of ensuring that this value can be *seen* by code that needs to see it and that it *persists* long enough to serve its purpose. There are three distinct levels of variable scope and lifetime:

Global variables

A global variable, or simply a *global*, is a variable declared at the top level of a Swift file. A global variable lives as long as the file lives, which is as long as the program runs. A global variable is visible everywhere (that's what "global" means). It is visible to all code within the *same* file, because it is at top level; any other code in the same file is therefore at the same level or at a lower contained level of scope. Moreover, it is visible (by default) to all code within any *other* file in the same module, because Swift files in the same module can automatically see one another, and hence can see one another's top levels:

```
// File1:
let globalVariable = "global"
class Dog {
    func printGlobal() {
        print(globalVariable) // *
    }
}
// File2:
class Cat {
    func printGlobal() {
        print(globalVariable) // *
    }
}
```

Properties

A *property* is a variable declared at the top level of an object type declaration (an enum, struct, or class). There are two kinds of properties: instance properties and static/class properties.

Instance properties

By default, a property is an *instance* property. Its value can differ for each instance of this object type, as I explained in Chapter 1. Its lifetime is the same as the lifetime of the instance. An instance comes into existence through deliberate instantiation of an object type; the subsequent lifetime of the instance, and hence of its instance properties, depends primarily on the lifetime of the variable to which the instance *itself* is assigned.

Static/class properties

A property is a static/class property if its declaration is preceded by the keyword `static` or `class`. (I'll go into detail about those terms in Chapter 4.) Its lifetime is the same as the lifetime of the object type. If the object type is declared at the top level of a file, the property lives as long as the program runs.

A property is visible only by way of the object. An object's methods can see that object's properties directly; such code can refer to the property using dot-notation with `self`, and I always do this as a matter of style, but `self` can usually be omitted except for purposes of disambiguation. An instance property is also visible (by default) to other code, provided the other code has a reference to this instance; in that case, the property can be referred to through dot-notation with the instance reference. A static/class property is visible (by default) to other code that can see the name of this object type; in that case, it can be referred to through dot-notation with the object type:

```
// File1:
class Dog {
    static let staticProperty = "staticProperty"
    let instanceProperty = "instanceProperty"
    func printInstanceProperty() {
        print(self.instanceProperty) // *
    }
}
// File2:
class Cat {
    func printDogStaticProperty() {
        print(Dog.staticProperty) // *
    }
    func printDogInstanceProperty() {
        let d = Dog()
        print(d.instanceProperty) // *
    }
}
```

Local variables

A local variable is a variable declared inside a function body. A local variable lives only as long as its surrounding curly-braces scope lives: it comes into existence when the path of execution passes into the scope and reaches the variable declaration, and it goes out of existence when the path of execution exits the scope. Local variables are sometimes called *automatic*, to signify that they come into and go out of existence automatically. A local variable can be seen only by subsequent code within the same scope (including a subsequent deeper scope within the same scope):

```
class Dog {
    func printLocalVariable() {
        let localVariable = "local"
        print(localVariable) // *
    }
}
```

Variable Declaration

A variable is declared with `let` or `var`:

- With `let`, the variable becomes a *constant* — its value can never be changed after the first assignment of a value.
- With `var`, the variable is a true variable, and its value can be changed by subsequent assignment.

A variable declaration is usually accompanied by *initialization* — you use an equal sign to assign the variable a value, as part of the declaration. That, however, is not a requirement; it is legal to declare a variable without immediately initializing it.

What is *not* legal is to declare a variable without giving it a *type*. A variable *must* have a type from the outset, and that type can *never be changed*. A variable declared with `var` can have its *value* changed by subsequent assignment, but the new value must conform to the variable's fixed type.

You can give a variable a type explicitly or implicitly:

Explicit variable type declaration
: After the variable's name in the declaration, add a colon and the name of the type:

```
var x : Int
```

Implicit variable type by initialization
: If you initialize the variable as part of the declaration, and if you provide no explicit type, Swift will *infer* its type, based on the value with which it is initialized:

```
var x = 1 // and now x is an Int
```

It is perfectly possible to declare a variable's type explicitly *and* assign it an initial value, all in one move:

```
var x : Int = 1
```

In that example, the explicit type declaration is superfluous, because the type (Int) would have been inferred from the initial value. Sometimes, however, providing an explicit type, even while also assigning an initial value, is *not* superfluous. Here are the main situations where that's the case:

Swift's inference would be wrong
: A very common case in my own code is when I want to provide the initial value as a numeric literal. Swift will infer either Int or Double, depending on whether the literal contains a decimal point. But there are a lot of other numeric types! When I mean one of those, I will provide the type explicitly, like this:

```
let value : Float = 2.0
```

Swift can't infer the type
: Sometimes, the type of the initial value is completely unknown to the compiler unless you tell it. A very common case involves option sets (discussed in Chapter 4). This won't compile:

```
var opts = [.autoreverse, .repeat] // compile error
```

The problem is that Swift doesn't know the type of `.autoreverse` and `.repeat` unless we tell it:

```
let opts : UIView.AnimationOptions = [.autoreverse, .repeat]
```

The programmer can't infer the type

I frequently include a superfluous explicit type declaration as a kind of note to myself. Here's an example from my own code:

```
let duration : CMTime = track.timeRange.duration
```

In that code, `track` is an AVAssetTrack. Swift knows perfectly well that the `duration` property of an AVAssetTrack's `timeRange` property is a CMTime. But I don't! In order to remind myself of that fact, I've shown the type explicitly.

Even if the compiler can infer a variable's type correctly from its initial value, such inference takes time. You can reduce compilation times by providing your variable declarations with explicit types.

As I've already said, a variable doesn't have to be initialized when it is declared — even if the variable is a constant. It is legal to write this:

```
let x : Int
```

Now `x` is an empty shoebox — an Int variable without an initial value. You can assign this variable an initial value later. Since this particular variable is a constant, that initial value will be its only value from then on.

In the case of an instance property of an object (at the top level of an enum, struct, or class declaration), that sort of thing is quite normal, because the property can be initialized in the object's initializer function. (I'll have more to say about that in Chapter 4.) For a local variable, however, such behavior is unusual, and I strongly urge you to avoid it. It isn't a disaster — the Swift compiler will stop you from trying to use a variable that has never been assigned a value — but it's not a good habit. A local variable should generally be initialized as part of its declaration.

The exception that proves the rule is what we might call *conditional initialization*. Sometimes, we don't *know* a variable's initial value until we've performed some sort of conditional test. The variable itself, however, can be declared only once; so it must be declared in advance and conditionally initialized afterward. This sort of thing is not unreasonable:

```
let timed : Bool
if val == 1 {
    timed = true
} else {
    timed = false
}
```

That particular example can arguably be better expressed in other ways (which I'll come to in Chapter 5), but there are situations where conditional initialization is the cleanest approach.

When a variable's *address* is to be passed as argument to a function, the variable must be declared *and initialized* beforehand, even if the initial value is fake. Recall this example from Chapter 2:

```
var r : CGFloat = 0
var g : CGFloat = 0
var b : CGFloat = 0
var a : CGFloat = 0
c.getRed(&r, green: &g, blue: &b, alpha: &a)
```

After that code runs, our four CGFloat `0` values will have been replaced; they were just momentary placeholders, to satisfy the compiler.

On rare occasions, you'll want to call a Cocoa method that returns a value immediately and later uses that value in a function passed to that same method. For example, Cocoa has a UIApplication instance method declared like this:

```
func beginBackgroundTask(expirationHandler handler: (() -> Void)? = nil)
    -> UIBackgroundTaskIdentifier
```

`beginBackgroundTask(expirationHandler:)` returns an identifier object, and will later call the `expirationHandler:` function passed to it — a function in which you will want to *use* the identifier object that was returned at the outset. Swift's safety rules won't let you declare the variable that holds this identifier and use it in an anonymous function all in the same statement:

```
let bti = UIApplication.shared.beginBackgroundTask {
    UIApplication.shared.endBackgroundTask(bti)
} // error: variable used within its own initial value
```

Therefore, you need to declare the variable beforehand; but then Swift has another complaint:

```
var bti : UIBackgroundTaskIdentifier
bti = UIApplication.shared.beginBackgroundTask {
    UIApplication.shared.endBackgroundTask(bti)
} // error: variable captured by a closure before being initialized
```

One solution is to declare the variable beforehand with a placeholder value:

```
var bti : UIBackgroundTaskIdentifier = .invalid
bti = UIApplication.shared.beginBackgroundTask {
    UIApplication.shared.endBackgroundTask(bti)
}
```

The real trouble here is that `beginBackgroundTask` is rather archaic. A more modern method would probably pass the returned value directly into the function parameter. Cocoa's UIAction initializer `init(title:handler:)` is a case in point:

```
let myAction = UIAction(title: "Hello") { action in
    print(action)
}
```

In that code, `action` *is* `myAction`, so the body of the anonymous `handler:` function can refer to it without saying `myAction`, which would be illegal. (This approach also prevents a possible retain cycle, discussed in Chapter 5.)

Computed Variable Initialization

Sometimes, you'd like to run several lines of code in order to compute a variable's initial value. A simple and compact way to express this is with a define-and-call anonymous function (see "Define-and-Call" on page 50). I'll illustrate by rewriting an earlier example:

```
let timed : Bool = {
    if val == 1 {
        return true
    } else {
        return false
    }
}()
```

You can do the same thing when you're initializing an instance property. Here's a class with an image (a UIImage) that I'm going to need many times later on. It makes sense to create this image in advance as a constant instance property of the class. To create it means to draw it. That takes several lines of code. So I declare and initialize the property by defining and calling an anonymous function, like this (for my `imageOfSize` utility, see Chapter 2):

```
class RootViewController : UITableViewController {
    let cellBackgroundImage : UIImage = {
        return imageOfSize(CGSize(width:320, height:44)) {
            // ... drawing goes here ...
        }
    }()
    // ... rest of class goes here ...
}
```

You might ask: Instead of a define-and-call initializer, why don't I declare an instance method and initialize the instance property by calling that method? The reason is that that's illegal:

```
class RootViewController : UITableViewController {
    let cellBackgroundImage : UIImage = self.makeTheImage() // compile error
    func makeTheImage() -> UIImage {
        return imageOfSize(CGSize(width:320, height:44)) {
            // ... drawing goes here ...
        }
    }
}
```

The problem is that, at the time of initializing the instance property, there is no instance yet — the instance is what we are in the process of creating. Therefore you

can't refer to `self` (implicitly or explicitly) in a property declaration's initializer. A define-and-call anonymous function, however, is legal. But the define-and-call anonymous function *still* can't refer to `self`! I'll provide a workaround a little later in this chapter.

Computed Variables

The variables I've been describing so far in this chapter have all been *stored* variables. The named shoebox analogy applies: a value can be put into the shoebox by assigning it to the variable, and it then sits there and can be retrieved later by referring to the variable, for as long the variable lives.

But a variable in Swift can work in a completely different way: it can be *computed*. This means that the variable, instead of having a value, has *functions*. One function, the *setter*, is called when the variable is assigned to. The other function, the *getter*, is called when the variable is referred to. Here's some code illustrating schematically the syntax for declaring a computed variable:

```
var now : String { ❶
    get { ❷
        return Date().description ❸
    }
    set { ❹
        print(newValue) ❺
    }
}
```

❶ The variable must be declared with `var` (not `let`). Its type must be declared explicitly. The type is followed immediately by curly braces.

❷ The getter function is called `get`. There is no formal function declaration; the word `get` is simply followed immediately by a function body in curly braces.

❸ The getter function must return a value of the same type as the variable. When the getter is a single statement, it is legal to omit the keyword `return`.

❹ The setter function is called `set`. There is no formal function declaration; the word `set` is simply followed immediately by a function body in curly braces.

❺ The setter behaves like a function taking one parameter. By default, this parameter arrives into the setter function body with the local name `newValue`.

Here's some code that illustrates the use of our computed variable. You don't treat it any differently than any other variable! To assign to the variable, assign to it; to use the variable, use it. Behind the scenes, though, the setter and getter functions are called:

```
now = "Howdy" // Howdy ❶
print(now) // 2021-06-26 17:03:30 +0000 ❷
```

❶ Assigning to `now` calls its setter. The argument passed into this call is the assigned value; here, that's `"Howdy"`. That value arrives in the `set` function as `newValue`. Our `set` function prints `newValue` to the console.

❷ Fetching `now` calls its getter. Our `get` function obtains the current date-time and translates it into a string, and returns the string. Our code then prints that string to the console.

There are a couple of variants on the basic syntax I've just illustrated:

- The name of the `set` function parameter doesn't have to be `newValue`. To specify a different name, put it in parentheses after the word `set`, like this:

```
set (val) { // now you can use "val" inside the setter function body
```

- There doesn't have to be a setter. If the setter is omitted, this becomes a *read-only* variable. This is the computed variable equivalent of a `let` variable: attempting to assign to it is a compile error.
- There must always be a getter! However, if there is no setter, the word `get` and the curly braces that follow it can be omitted. This is a legal declaration of a read-only variable (omitting the `return` keyword):

```
var now : String {
    Date().description
}
```

Computed Properties

In real life, your main use of computed variables will nearly always be as instance properties. Here are some common ways in which computed properties are useful:

Façade for a longer expression
: When a value can be readily calculated or obtained each time it is needed, it often makes for simpler syntax to express it as a read-only computed variable, which effectively acts as a shorthand for a longer expression. Here's an example from my own code:

```
var mp : MPMusicPlayerController {
    MPMusicPlayerController.systemMusicPlayer
}
var nowPlayingItem : MPMediaItem? {
    self.mp.nowPlayingItem
}
```

No work is saved by these computed variables; each time we ask for `self.nowPlayingItem`, we are fetching `MPMusicPlayerController.systemMusicPlayer.nowPlayingItem`. Still, the clarity and convenience of the resulting code justifies the use of computed variables here.

Façade for an elaborate calculation

A computed variable getter can encapsulate multiple lines of code, in effect turning a method into a property. Here's an example from my own code:

```
var authorOfItem : String? {
    guard let authorNodes =
        self.extensionElements(
            withXMLNamespace: "http://www.tidbits.com/dummy",
            elementName: "app_author_name")
        else {return nil}
    guard let authorNode = authorNodes.last as? FPExtensionNode
        else {return nil}
    return authorNode.stringValue
}
```

In that example, I'm diving into some parsed XML and extracting a value. I could have declared this as a method `func authorOfItem() -> String?`, but a method expresses a *process*, whereas a computed property characterizes it more intuitively as a *thing*.

Façade for storage

A computed variable can sit in front of one or more stored variables, acting as a gatekeeper on how those stored variables are set and fetched. This is comparable to an accessor method in Objective-C. Commonly, a public computed variable is backed by a private stored variable. The simplest possible storage façade would do no more than get and set the private stored variable directly:

```
private var _p : String = ""
var p : String {
    get {
        self._p
    }
    set {
        self._p = newValue
    }
}
```

That's legal but pointless. A storage façade becomes useful when it does *other* things while getting or setting the stored variable. Here's a more realistic example: a "clamped" setter. This is an Int property to which only values between 0 and 5 can be assigned; larger values are replaced by 5, and smaller values are replaced by 0:

```
private var _pp : Int = 0
var pp : Int {
    get {
        self._pp
    }
    set {
        self._pp = max(min(newValue,5),0)
    }
}
```

As the preceding examples demonstrate, a computed instance property getter or setter can refer to other instance members. That's important, because in general the initializer for a stored property can't do that. The reason it's legal for a computed property is that the getter and setter functions won't be called until the instance actually exists.

Property Wrappers

If we have several storage façade computed properties that effectively do the same thing, we're going to end up with a lot of repeated code. Imagine implementing more than one Int property with a clamped setter, as in the preceding section. It would be nice to move the common functionality off into a single location. We can do that using a *property wrapper*.

A property wrapper is declared as a type marked with the `@propertyWrapper` attribute, and must have a `wrappedValue` computed property. Here's a property wrapper implementing the "clamped" pattern:

```
@propertyWrapper struct Clamped {
    private var _i : Int = 0
    var wrappedValue : Int {
        get {
            self._i
        }
        set {
            self._i = Swift.max(Swift.min(newValue,5),0)
        }
    }
}
```

The result is that we can declare a computed property marked with a custom attribute whose name is the same as that struct (`@Clamped`), with *no* getter or setter:

```
@Clamped var p
```

Our property `p` doesn't need to be initialized, because it's a computed property. And it doesn't need a getter or a setter — indeed, in this case it doesn't even need a type declaration — because the `wrappedValue` computed property of the Clamped struct supplies them!

Behind the scenes, an actual Clamped instance has been created for us. When we set `self.p` to some value through assignment, the assignment passes through the Clamped instance's `wrappedValue` setter, and the resulting clamped value is stored in the Clamped instance's `_i` property. When we fetch the value of `self.p`, what we get is the value returned from the Clamped instance's `wrappedValue` getter, which is the value stored in the Clamped instance's `_i` property.

Thanks to our property wrapper, we have *encapsulated* this computed property pattern, which means we can now declare *another* `@Clamped` property which will behave in just the same way. It's also nice that this pattern now has a *name:* the declaration `@Clamped var` tells us what the behavior of this computed property will be.

(Believe it or not, I created the Clamped example *before* discovering that the Swift language proposal for property wrappers uses the same example!)

There's considerably more to know about property wrappers, but I'll postpone further discussion to Chapter 5, after I've explained other language features that will help you appreciate their power and flexibility.

Setter Observers

Computed variables are not needed as a stored variable façade as often as you might suppose. That's because Swift has another feature, which lets you inject functionality into the setter of a *stored* variable — *setter observers*. These are functions that are called just before and just after other code sets a stored variable.

The syntax for declaring a variable with a setter observer is very similar to the syntax for declaring a computed variable; you can write a `willSet` function, a `didSet` function, or both:

```
var s = "whatever" { ❶
    willSet { ❷
        print(newValue) ❸
    }
    didSet { ❹
        print(oldValue) ❺
        // self.s = "something else"
    }
}
```

❶ The variable must be declared with `var` (not `let`). It can be assigned an initial value. It is then followed immediately by curly braces.

❷ The `willSet` function, if there is one, is the word `willSet` followed immediately by a function body in curly braces. It is called when other code sets this variable, just *before* the variable actually receives its new value.

❸ By default, the `willSet` function receives the incoming new value as `newValue`. You can change this name by writing a different name in parentheses after the word `willSet`. The old value is still sitting in the stored variable, and the `willSet` function can access it there.

❹ The `didSet` function, if there is one, is the word `didSet` followed immediately by a function body in curly braces. It is called when other code sets this variable, just *after* the variable actually receives its new value.

❺ By default, the `didSet` function receives the old value, which has already been replaced as the value of the variable, as `oldValue`. You can change this name by writing a different name in parentheses after the word `didSet`. The new value is already sitting in the stored variable, and the `didSet` function can access it there. Moreover, it is legal for the `didSet` function to set the stored variable to a different value.

Setter observer functions are *not* called when the stored variable is initialized or when the `didSet` function changes the stored variable's value. That would be circular!

In real-life iOS programming, you'll want the visible interface to reflect the state of your objects. A setter observer is a simple but powerful way to synchronize the interface with a property. In this example, we have an instance property of a view class, determining how much the view should be rotated; every time this property changes, we change the interface to reflect it, setting `self.transform` so that the view *is* rotated by that amount:

```
var angle : CGFloat = 0 {
    didSet {
        // modify interface to match
        self.transform = CGAffineTransform(rotationAngle: self.angle)
    }
}
```

A computed variable can't have setter observers. But it doesn't need them! There's a setter function, so anything additional that needs to happen during setting can be programmed directly into that setter function. (But a property-wrapped computed variable *can* have setter observers.)

Lazy Initialization

The term *lazy* is not a pejorative puritanical judgment; it's a formal description of a useful behavior. If a stored variable is assigned an initial value as part of its declaration, and if it uses lazy initialization, then the initial value is not actually evaluated and assigned until running code accesses the variable's value.

There are four types of variable that can be initialized lazily in Swift:

Global variables
: Global variables are *automatically lazy*. This makes sense if you ask yourself when they should be initialized. As the app launches, files and their top-level code are encountered. It would make no sense to initialize globals now, because the app isn't even running yet. Thus global initialization must be postponed to some moment that *does* make sense. Therefore, a global variable's initialization doesn't happen until other code first refers to that global. Under the hood, this behavior is implemented in such a way as to make initialization both singular (it can happen only once) and thread-safe.

Static properties
: Static properties are *automatically lazy*. They behave exactly like global variables, and for basically the same reason. (There are no stored class properties in Swift, so class properties can't be initialized and thus can't have lazy initialization.)

Instance properties
: An instance property is not lazy by default, but it may be made lazy by marking its declaration with the keyword `lazy`. This property must be declared with `var`, not `let`.

Local variables
: New in Swift 5.5, a local variable can be declared with `lazy var`.

If your code *never* refers to a lazily initialized variable, its initializer never runs; clearly, that's useful if the initial value might be expensive to generate, so you'd like to avoid generating it unless it is actually needed.

What if the first reference to a lazily initialized variable *sets* that variable? For a global variable or a static property, the initializer is evaluated and is immediately replaced by the new value. But the initializer for a `lazy` instance property or local variable will *never* be evaluated if the variable is set before it is fetched.

Singleton

Lazy initialization is often used to implement *singleton*. Singleton is a pattern where all code is able to get access to a single shared instance of a certain class:

```
class MyClass {
    static let shared = MyClass()
}
```

Now other code can obtain a reference to MyClass's singleton by saying `MyClass.shared`. The singleton instance is not created until the *first* time other code says `MyClass.shared`; subsequently, no matter how many times other code may say `MyClass.shared`, the instance returned is always *that same instance*. (That is *not* what

would happen if this were a computed read-only property whose getter calls `MyClass()` and returns that instance; do you see why?)

Lazy Initialization of Instance Properties

Aside from the efficiency benefits of a laziness, a lazy initializer can do things that a normal initializer can't. In particular, a lazy initializer can *refer to the instance*. A normal initializer can't do that, because the instance doesn't yet exist at the time that a normal initializer would need to run (we're in the middle of creating the instance, so it isn't ready yet). A lazy initializer, by contrast, is guaranteed not to run until some time after the instance has fully come into existence, so referring to the instance is fine. This code would be illegal if the `arrow` property weren't declared `lazy`:

```
class MyView : UIView {
    lazy var arrow = self.arrowImage() // legal
    func arrowImage () -> UIImage {
        // ... big image-generating code goes here ...
    }
}
```

A very common idiom is to initialize a lazy instance property with a define-and-call anonymous function whose code can refer to `self`:

```
lazy var prog : UIProgressView = {
    let p = UIProgressView(progressViewStyle: .default)
    p.alpha = 0.7
    p.trackTintColor = UIColor.clear
    p.progressTintColor = UIColor.black
    p.frame =
        CGRect(x:0, y:0, width:self.view.bounds.size.width, height:20) // legal
    p.progress = 1.0
    return p
}()
```

Built-In Simple Types

Every variable, and every value, must have a type. But what types are there? Up to this point, I've assumed the existence of some types, such as Int and String, without formally telling you about them. Here's a survey of the primary simple types provided by Swift, along with some instance methods, global functions, and operators that apply to them. (Collection types will be discussed at the end of Chapter 4.)

Bool

The Bool object type (a struct) has only two values, commonly regarded as true and false (or yes and no). You can represent these values using the literal keywords `true` and `false`, and it is natural to think of a Bool value as *being* either `true` or `false`:

No Lazy Let for Instance Properties

There's no `lazy let` for instance properties, so you can't readily make a lazy instance property read-only. That's unfortunate, because there are some common situations that would benefit from such a feature.

Suppose we want to arm ourselves (`self`) with a `helper` property holding an instance of a Helper class that needs a reference back to `self`. We also want this one Helper instance to persist for the entire lifetime of `self`. We can enforce that rule by making `helper` a `let` property and initializing it in its declaration. But we can't pass `self` into the Helper initializer, because we can't refer to `self` in the property declaration. We can solve that problem by declaring the `helper` property `lazy`.

The trouble is that then this has to be a `var` property, meaning that, in theory, other code can come along and replace this Helper instance with another. That's not what we want. Of course, we'll try not to let it happen, but the point is that the expression `lazy var` fails to enforce the desired policy.

```
var selected : Bool = false
```

In that code, `selected` is a Bool variable initialized to `false`; it can subsequently be set to `false` or `true`, and to no other values. Because of its simple yes-or-no state, a Bool variable of this kind is often referred to as a *flag*.

Cocoa methods very often expect a Bool parameter or return a Bool value. For example, when your app launches, Cocoa calls a method in your code declared like this:

```
func application(_ application: UIApplication,
    didFinishLaunchingWithOptions
    launchOptions: [UIApplication.LaunchOptionsKey : Any]?)
    -> Bool {
```

You can do anything you like in that method; often, you will do nothing. But you must return a Bool! And in real life, that Bool will probably be `true`. A minimal implementation looks like this:

```
func application(_ application: UIApplication,
    didFinishLaunchingWithOptions
    launchOptions: [UIApplication.LaunchOptionsKey : Any]?)
    -> Bool {
        return true
}
```

A Bool is useful in conditions; as I'll explain in Chapter 5, when you say `if` *`something`*, the *`something`* is the condition, and is a Bool or an expression that evaluates to a Bool. When you compare two values using the equality comparison operator `==`, the result is a Bool — `true` if they are equal to each other, `false` if they are not:

```
if meaningOfLife == 42 { // ...
```

(I'll talk more about equality comparison in a moment, when we come to discuss types that can be compared, such as Int and String.)

When preparing a condition, you will sometimes find that it enhances clarity to store the Bool value in a variable beforehand:

```
let comp = self.traitCollection.horizontalSizeClass == .compact
if comp { // ...
```

Observe that, when employing that idiom, we use the Bool variable `comp` *directly* as the condition. There is no need to test explicitly whether a Bool equals `true` or `false`; the conditional expression itself is already testing that. It is pointless to say `if comp == true`, because `if comp` already *means* "if `comp` is `true`."

Since a Bool can be used as a condition, a call to a function that returns a Bool can be used as a condition. Here's an example from my own code. I've declared a function that returns a Bool to say whether the cards the user has selected constitute a correct answer to the puzzle:

```
func isCorrect(_ cells:[CardCell]) -> Bool { // ...
```

I can then use a call to `isCorrect` as a condition:

```
if self.isCorrect(cellsToTest) { // ...
```

Unlike many computer languages, nothing else in Swift is implicitly coerced to or treated as a Bool. In C, for example, a boolean is actually a number, and `0` is false. But in Swift, nothing is false but `false`, and nothing is true but `true`.

The type name, Bool, comes from the English mathematician George Boole; Boolean algebra provides operations on logical values. Bool values are subject to these same operations:

`!` *(not)*
: The `!` unary operator reverses the truth value of the Bool to which it is applied as a prefix. If `ok` is `true`, `!ok` is `false` — and vice versa.

`&&` *(logical-and)*
: Returns `true` only if both operands are `true`; otherwise, returns `false`. If the first operand is `false`, the second operand is not even evaluated (avoiding possible side effects).

`||` *(logical-or)*
: Returns `true` if either operand is `true`; otherwise, returns `false`. If the first operand is `true`, the second operand is not even evaluated (avoiding possible side effects).

If a logical operation is complicated or elaborate, parentheses around subexpressions can help clarify both the logic and the order of operations.

A common situation is that we have a Bool stored in a `var` variable somewhere, and we want to reverse its value — that is, make it `true` if it is `false`, and `false` if it is `true`. The `!` operator solves the problem; we fetch the variable's value, reverse it with `!`, and assign the result back into the variable:

```
v.isUserInteractionEnabled = !v.isUserInteractionEnabled
```

That, however, is cumbersome and error-prone, so there's a simpler way — call the `toggle` method on the Bool variable:

```
v.isUserInteractionEnabled.toggle()
```

Numbers

The main numeric types are Int and Double — meaning that, left to your own devices, those are the types you'll generally use. Other numeric types exist mostly for compatibility with the C and Objective-C APIs that Swift needs to be able to talk to when you're programming iOS.

Int

The Int object type (a struct) represents an integer between `Int.min` and `Int.max` inclusive. The actual values of those limits might depend on the platform and architecture under which the app runs, so don't count on them to be absolute; in my testing at this moment, they are -2^{63} and $2^{63}-1$ respectively (64-bit words).

The easiest way to represent an Int value is as a numeric literal. A numeric literal without a decimal point is taken as an Int by default. Internal underscores are legal; this is useful for making long numbers readable. Leading zeroes are legal; this is useful for padding and aligning values in your code.

You can write an Int literal using binary, octal, or hexadecimal digits. To do so, start the literal with `0b`, `0o`, or `0x` respectively. For example, `0x10` is decimal 16.

Negative numbers are stored in the two's complement format (consult Wikipedia if you're curious). You can write a binary literal that looks like the underlying storage, but to use it you must pass it through the `Int(bitPattern:)` initializer.

Double

The Double object type (a struct) represents a floating-point number to a precision of about 15 decimal places (64-bit storage).

The easiest way to represent a Double value is as a numeric literal. A numeric literal containing a decimal point is taken as a Double by default. Internal underscores and leading zeroes are legal.

A Double literal may *not* begin with a decimal point (unlike C and Objective-C). If the value to be represented is between 0 and 1, start the literal with a leading `0`.

You can write a Double literal using scientific notation. Everything after the letter `e` is the exponent of 10. You can omit the decimal point if the fractional digits would be zero. For example, `3e2` is 3 times 10^2 (300).

You can write a Double literal using hexadecimal digits. To do so, start the literal with `0x`. You can use exponentiation here too (and again, you can omit the decimal point); everything after the letter `p` is the exponent of 2. For example, `0x10p2` is decimal 64, because you are multiplying 16 by 2^2.

There are static properties `Double.infinity` and `Double.pi`, and an instance property `isZero`, among others.

Numeric coercion

Coercion is the conversion of a value from one type to another, and numeric coercion is the conversion of a value from one numeric type to another. Swift doesn't really have explicit coercion, but it has something that serves the same purpose — instantiation. Swift numeric types are supplied with initializers that take another numeric type as parameter. To convert an Int explicitly into a Double, instantiate Double with the Int in the parentheses. To convert a Double explicitly into an Int, instantiate Int with the Double in the parentheses; this will truncate the original value (everything after the decimal point will be thrown away):

```
let i = 10
let x = Double(i)
print(x) // 10.0, a Double
let y = 3.8
let j = Int(y)
print(j) // 3, an Int
```

When numeric values are assigned to variables or passed as arguments to a function, Swift can perform implicit coercion *of literals only*. This code is legal:

```
let d : Double = 10
```

But this code is not legal, because what you're assigning is a *variable* (not a literal) of a different type; the compiler will stop you:

```
let i = 10
let d : Double = i // compile error
```

The problem is that `i` is an Int and `d` is a Double, and never the twain shall meet. The solution is to *coerce explicitly* as you assign or pass the variable:

```
let i = 10
let d : Double = Double(i)
```

The same rule holds when numeric values are combined by an arithmetic operation. Swift will perform implicit coercion *of literals only*. The usual situation is an Int combined with a Double; the Int is treated as a Double:

```
let x = 10/3.0
print(x) // 3.33333333333333
```

But *variables* of different numeric types must be *coerced explicitly* so that they are the *same* type if you want to combine them in an arithmetic operation:

```
let i = 10
let n = 3.0
let x = i / n // compile error; you need to say Double(i)
```

These rules are evidently a consequence of Swift's strict typing; but (as far as I am aware) they constitute very unusual treatment of numeric values for a modern computer language, and will probably drive you mad in short order. The examples I've given so far were easily solved, but things can become more complicated if an arithmetic expression is longer, and the problem is compounded by the existence of other numeric types that are needed for compatibility with Cocoa, as I shall now proceed to explain.

Other numeric types

If you were using Swift in some isolated, abstract world, you could probably do all necessary arithmetic with Int and Double alone. But when you're programming iOS, you encounter Cocoa, which is full of other numeric types; and Swift has types that match every one of them. In addition to Int, there are signed integer types of various sizes — Int8, Int16, Int32, Int64 — plus the unsigned integer type UInt along with UInt8, UInt16, UInt32, and UInt64. In addition to Double, there is the lower-precision Float (32-bit storage, about 6 or 7 decimal places of precision), the even lower-precision Float16, the extended-precision Float80, and (from the Core Graphics framework) CGFloat.

You may also encounter a C numeric type when trying to interface with a C API. These types, as far as Swift is concerned, are just type aliases, meaning that they are alternate names for another type: a CDouble (corresponding to C's `double`) is just a Double by another name, a CLong (C's `long`) is an Int, and so on. Many other numeric type aliases will arise in various Cocoa frameworks; for example, TimeInterval (Objective-C NSTimeInterval) is merely a type alias for Double.

Figure 3-1. Quick Help displays a variable's type

Recall that you can't assign, pass, or combine values of different numeric types using variables; you have to coerce those values explicitly to the correct type. But now it turns out that you're being flooded by Cocoa with numeric values of many types! Cocoa will often hand you a numeric value that is neither an Int nor a Double — and you won't necessarily realize this, until the compiler stops you dead in your tracks for some sort of type mismatch. You must then figure out what you've done wrong and coerce everything to the same type.

Here's a typical example from one of my apps. A slider in the interface is a UISlider, whose `minimumValue` and `maximumValue` are Floats. In this code, `s` is a UISlider, `g` is a UIGestureRecognizer, and we're trying to use the gesture recognizer to move the slider's "thumb" to wherever the user tapped within the slider:

```
let pt = g.location(in:s) ❶
let percentage = pt.x / s.bounds.size.width ❷
let delta = percentage * (s.maximumValue - s.minimumValue) // compile error ❸
```

That won't compile. Here's why:

❶ `pt` is a CGPoint, and therefore `pt.x` is a CGFloat.

❷ Luckily, `s.bounds.size.width` is also a CGFloat, so the second line compiles; the type of `percentage` is now inferred as CGFloat.

❸ We now try to combine `percentage` with `s.maximumValue` and `s.minimumValue` — and they are Floats, not CGFloats. That's a compile error.

This sort of thing is not an issue in C or Objective-C, where there is implicit coercion; but in Swift a CGFloat can't be combined with Floats. We must coerce explicitly:

```
let delta = Float(percentage) * (s.maximumValue - s.minimumValue)
```

The good news here is that if you can get enough of your code to compile, Xcode's Quick Help feature (Chapter 9) will tell you what type Swift has inferred for a selected variable (Figure 3-1). This can assist you in tracking down your issues with numeric types.

CGFloat is a special case. It resolves to the size of Float or Double, depending on the bitness of the architecture (32-bit or 64-bit); but this distinction has lost much of its importance, because nowadays 32-bit architecture is rare and in any case it handles Doubles efficiently. New in Swift 5.5, therefore, a Double and a CGFloat are

interchangeable; you can supply one where the other is expected *without coercion*, relieving iOS programmers from considerable inconvenience:

```
var cfg : CGFloat = 1
var d : Double = 2
cfg = d // legal
d = cfg + d // legal
let x = cfg + d // x is inferred as Double
```

Another problem is that not every numeric value *can* be coerced to a numeric value of a different type. In particular, integers of various sizes can be out of range with respect to integer types of other sizes. For example, `Int8.max` is 127, so attempting to assign a literal `128` or larger to an Int8 variable is illegal. Fortunately, the compiler will stop you in that case, because it knows what the literal is. But now consider *coercing* a variable value of a larger integer type to an Int8:

```
let i : Int16 = 128
let ii = Int8(i)
```

That code is legal — and will crash at runtime. One solution is to call the numeric `exactly:` initializer; this is a *failable* initializer — meaning, as I'll explain in Chapter 4, that you won't crash, but you'll have to add code to test whether the coercion succeeded (and you'll understand what the test would be when you've read the discussion of Optionals later in this chapter):

```
let i : Int16 = 128
let ii = Int8(exactly:i)
if // ... test to learn whether ii holds a real Int8
```

Yet another solution is to call the `clamping:` initializer; it *always* succeeds, because an out of range value is forced to fall within range:

```
let i : Int16 = 128
let ii = Int8(clamping:i) // 127
```

(There is also a `truncatingIfNeeded:` initializer, but you probably won't need to know about it unless you are deliberately manipulating integers as binary, so I won't describe it here.)

When a floating-point type, such as a Double, is coerced to an integer type, the stuff after the decimal point is thrown away first and then the coercion is attempted. `Int8(127.9)` succeeds, because 127 is in bounds.

Arithmetic operations

Swift's arithmetic operators are as you would expect; they are familiar from other computer languages as well as from real arithmetic:

`+` *(addition operator)*
: Add the second operand to the first and return the result.

`-` *(subtraction operator)*
: Subtract the second operand from the first and return the result. A different operator (unary minus), used as a prefix, looks the same; it returns the additive inverse of its single operand. (There is, in fact, also a unary plus operator, which returns its operand unchanged.)

`*` *(multiplication operator)*
: Multiply the first operand by the second and return the result.

`/` *(division operator)*
: Divide the first operand by the second and return the result. As in C, division of one Int by another Int yields an Int; any remaining fraction is stripped away. `10/3` is 3, not 3-and-one-third.

`%` *(remainder operator)*
: Divide the first operand by the second and return the remainder. The result can be negative, if the first operand is negative; if the second operand is negative, it is treated as positive. For floating-point operands, use a method such as `remainder(dividingBy:)`.

  Integers also have a `quotientAndRemainder(dividingBy:)` method, which returns a tuple of two integers labeled `quotient` and `remainder`. (I'll discuss tuples later in this chapter.) If the question is whether one integer evenly divides another, calling `isMultiple(of:)` may be clearer than checking for a zero remainder.

Integer types can be treated as binary bitfields and subjected to binary bitwise operations:

`&` *(bitwise-and)*
: A bit in the result is 1 if and only if that bit is 1 in both operands.

`|` *(bitwise-or)*
: A bit in the result is 0 if and only if that bit is 0 in both operands.

`^` *(bitwise-or, exclusive)*
: A bit in the result is 1 if and only if that bit is not identical in both operands.

`~` *(bitwise-not)*
: Precedes its single operand; inverts the value of each bit and returns the result.

`<<` *(shift left)*
: Shift the bits of the first operand leftward the number of times indicated by the second operand.

`>>` *(shift right)*
: Shift the bits of the first operand rightward the number of times indicated by the second operand.

Technically, the shift operators perform a logical shift if the integer is unsigned, and an arithmetic shift if the integer is signed.

Integer overflow or underflow — for example, adding two Int values so as to exceed `Int.max` — is a runtime error (your app will crash). In simple cases the compiler will stop you, but you can get away with it easily enough:

```
let i = Int.max - 2
let j = i + 12/2 // crash
```

Under certain circumstances you might want to force such an operation to succeed, so special overflow/underflow methods are supplied. These methods return a tuple; I'll show you an example even though I haven't discussed tuples yet:

```
let i = Int.max - 2
let (j, over) = i.addingReportingOverflow(12/2)
```

Now `j` is `Int.min + 3` (because the value has wrapped around from `Int.max` to `Int.min`) and `over` is a Bool reporting that overflow occurred.

If you don't care to hear about whether or not there was an overflow/underflow, special arithmetic operators let you suppress the error: `&+`, `&-`, `&*`.

You will frequently want to combine the value of an existing variable arithmetically with another value and store the result in the same variable. To do so, you will need to have declared the variable as a `var`:

```
var i = 1
i = i + 7
```

As a shorthand, operators are provided that perform the arithmetic operation and the assignment all in one move:

```
var i = 1
i += 7
```

The shorthand (*compound*) assignment arithmetic operators are `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`.

Operation precedence is largely intuitive: for example, `*` has a higher precedence than `+`, so `x+y*z` multiplies `y` by `z` first, and then adds the result to `x`. Use parentheses to dictate precedence explicitly: `(x+y)*z` performs the addition first.

Global functions from the Swift standard library include `abs` (absolute value), `max`, and `min`:

```
let i = -7
let j = 6
print(abs(i)) // 7
print(max(i,j)) // 6
```

Doubles are also stocked with mathematical methods. If `d` is a Double, you can say `d.squareRoot()` or `d.rounded()`; if `dd` is also a Double, you can say `Double.maximum(d,dd)`. Other global mathematical functions, such as trigonometric `sin` and `cos`, come from the C standard libraries that are visible because you've imported UIKit. In addition, the Swift Numerics package unites the floating-point types under the Real protocol and supplies most of the mathematical functions you could ever want (`import Numerics`; see Chapter 7).

Numeric types have a `random(in:)` static method allowing generation of a random number. The parameter is a range representing the bounds within which the random number should fall. (Ranges are discussed later in this chapter.) This method is much easier to use correctly than the C library methods such as `arc4random_uniform`, which should be avoided:

```
// pick a number from 1 to 10
let i = Int.random(in: 1...10)
```

Comparison

Numbers are compared using the comparison operators, which return a Bool. The expression `i==j` tests whether `i` and `j` are equal; when `i` and `j` are numbers, "equal" means numerically equal. So `i==j` is `true` only if `i` and `j` are "the same number," in exactly the sense you would expect.

The comparison operators are:

`==` *(equality operator)*
: Returns `true` if its operands are equal.

`!=` *(inequality operator)*
: Returns `false` if its operands are equal.

`<` *(less-than operator)*
: Returns `true` if the first operand is less than the second operand.

`<=` *(less-than-or-equal operator)*
: Returns `true` if the first operand is less than or equal to the second operand.

`>` *(greater-than operator)*
: Returns `true` if the first operand is greater than the second operand.

`>=` *(greater-than-or-equal operator)*
: Returns `true` if the first operand is greater than or equal to the second operand.

Curiously, you can compare values of different integer types, even though you can't combine them in an arithmetic operation:

```
let i:Int = 1
let i2:UInt8 = 2
let ok = i < i2 // true
let ok2 = i == i2 // false
let sum = i + i2 // error
```

Because of the way computers store numbers, equality comparison of Double values may not succeed where you would expect. To give a classic example, adding `0.1` to `0` ten times does not give the same result as multiplying `0.1` by ten:

```
let f = 0.1
var sum = 0.0
for _ in 0..<10 { sum += f }
let product = f * 10
let ok = sum == product // false
```

Working around this sort of thing is not easy. The usual approach is to check whether two values are sufficiently close to one another, but this begs the question of what constitutes sufficient closeness. A useful formula is:

```
let ok = sum >= product.nextDown && sum <= product.nextUp // true
```

String

The String object type (a struct) represents text. The simplest way to represent a String value is with a literal, delimited by double quotes:

```
let greeting = "hello"
```

A Swift string is thoroughly modern; under the hood, it's Unicode, and you can include any character (such as an emoji) directly in a string literal. If you don't want to bother typing a Unicode character whose codepoint you know, use the notation `\u{...}`, where what's between the curly braces is up to eight hex digits:

```
let leftTripleArrow = "\u{21DA}"
```

The backslash in that string representation is the *escape* character; it means, "I'm not really a backslash; I indicate that the next character gets special treatment." Various nonprintable and ambiguous characters are entered as escaped characters; the most important are:

`\n`
: A Unix newline character

`\t`
: A tab character

`\"`
: A quotation mark (escaped to show that this is not the end of the string literal)

`\\`
: A backslash (escaped because a lone backslash is the escape character)

Escaped quotation marks and backslashes can quickly make your string literals ugly and illegible. The issue arises particularly in contexts such as regular expression patterns. For example, the pattern `\b\d\d\b` (a word consisting of two digits) must be written `"\\b\\d\\d\\b"`. But you can omit the escape character before quotes and backslashes by surrounding your literal with one or more hash characters (`#`); these are all identical strings:

```
let pattold = "\\b\\d\\d\\b"
let pattnew = #"\b\d\d\b"# // same thing
let pattnew2 = ##"\b\d\d\b"## // same thing
```

That's called a *raw* string literal. The downside is that if you *want* to use a backslash as an escape character in a raw string literal, you must follow it with the same number of `#` characters you are using to surround your literal. The string `#"hello\nthere"#` does not contain a newline character (`\n`), but `#"hello\#nthere"#` does.

A string literal containing newline characters can be entered as multiple lines (rather than a single-line expression containing `\n` characters). This is called a *multiline* string literal. The rules are:

- The multiline string literal must be delimited by a triple of double quotes (`"""`) at start and end.
- No material may follow the opening delimiter on the same line.
- No material other than whitespace may appear on the same line as the closing delimiter.
- The last implicit newline character before the closing delimiter is ignored.
- The indentation of the closing delimiter dictates the indentation of the lines of text, which must be indented at least as far as the closing delimiter (except for completely empty lines).

For example:

```
func f() {
    let s = """
    Line 1
        Line 2
    Line 3
    """
    // ...
}
```

In that code, the string `s` consists of three lines of text; lines 1 and 3 start with no whitespace; line 2 starts with four spaces; and there are two newline characters, namely after lines 1 and 2. To add a newline after line 3, you could enter a blank line, or write it as an escaped `\n`.

Quotation marks do *not* have to be escaped. A line ending with a backslash is joined with the following line. In this code, the string `s` consists of just two lines of text; the second line consists of four spaces followed by "Line 2 and this is still line 2":

```
func f() {
    let s = """
    Line "1"
        Line 2 \
    and this is still Line 2
    """
    // ...
}
```

(You can surround a multiline string literal with # characters, making a raw multiline string literal; but you are unlikely to do so.)

String interpolation permits you to embed any value that can be output with `print` inside a string literal *as a string*, even if it is not itself a string. The notation is escaped parentheses: `\(...)`:

```
let n = 5
let s = "You have \(n) widgets."
```

Now `s` is the string `"You have 5 widgets."` The example is not very compelling, because we know what `n` is and could have typed `5` directly into our string; but imagine that we *don't* know what `n` is! Moreover, the stuff in escaped parentheses doesn't have to be the name of a variable; it can be almost any expression that evaluates as legal Swift:

```
let m = 4
let n = 5
let s = "You have \(m + n) widgets."
```

Inside a string interpolation, the quotation marks delimiting a literal string do not need to be escaped:

```
let s = "You have \("numerous".uppercased()) widgets"
```

String interpolation is legal inside a multiline string literal. It is also legal inside a raw string literal surrounded with # characters, but then the backslash must be followed by the same number of # characters, to indicate that it is the escape character.

String interpolation syntax can be customized to accept additional parameters, refining how the first parameter should be transformed. Expressions of this form can be legal:

```
let s = "You have \(n, roman:true) widgets"
```

(I'm imagining that if `n` is 5, this would yield `"You have V widgets"`.) I'll say more in Chapter 5 about how that is achieved.

To combine (concatenate) two strings, the simplest approach is to use the `+` operator:

```
let s = "hello"
let s2 = " world"
let greeting = s + s2
```

This convenient notation is possible because the `+` operator is *overloaded*: it does one thing when the operands are numbers (numeric addition) and another when the operands are strings (concatenation). As I'll explain in Chapter 5, *all* operators can be overloaded, and you can overload them to operate in some appropriate way on your own types.

The `+` operator comes with a `+=` assignment shortcut; naturally, the variable on the left side must have been declared with `var`:

```
var s = "hello"
let s2 = " world"
s += s2
```

As an alternative to `+=`, you can call the `append(_:)` instance method:

```
var s = "hello"
let s2 = " world"
s.append(s2)
```

Another way of concatenating strings is with the `joined(separator:)` method. You start with an array of strings to be concatenated, and hand it the string that is to be inserted between all of them:

```
let s = "hello"
let s2 = "world"
let space = " "
let greeting = [s,s2].joined(separator:space)
```

The comparison operators are also overloaded so that they work with String operands. Two String values are equal (`==`) if they are, in the natural sense, the same text. A String is less than another if it is alphabetically prior.

Some additional convenient instance methods and properties are provided. `isEmpty` returns a Bool reporting whether this string is the empty string (`""`). `hasPrefix(_:)` and `hasSuffix(_:)` report whether this string starts or ends with another string; `"hello".hasPrefix("he")` is `true`. The `uppercased` and `lowercased` methods provide uppercase and lowercase versions of the original string.

Coercion between a String and an Int is possible. To make a string that represents an Int, it is sufficient to use string interpolation; alternatively, use a String initializer taking the Int:

```
let i = 7
let s = String(i) // "7"
```

Your string can also represent an Int in some other base; in the initializer, supply a `radix:` argument expressing the base:

```
let i = 31
let s = String(i, radix:16) // "1f"
```

A String that might represent a number can be coerced to a numeric type; an integer type will accept a `radix:` argument expressing the base. The coercion might fail, because the String might *not* represent a number of the specified type; so the result is not a number but an Optional wrapping a number (I haven't talked about Optionals yet, so you'll have to trust me for now; failable initializers are discussed in Chapter 4):

```
let s = "31"
let i = Int(s) // Optional(31)
let s2 = "1f"
let i2 = Int(s2, radix:16) // Optional(31)
```

Similarly, you can coerce a Bool to a String, which will be `"true"` or `"false"`. Going the other way, you can coerce the string `"true"` to the Bool `true` and the string `"false"` to the Bool `false`; again, this is a failable initializer, and any other string will fail.

The length of a String, in characters, is given by its `count` property:

```
let s = "hello"
let length = s.count // 5
```

This property is called `count` rather then `length` because a String doesn't really have a simple length. The String comprises a sequence of Unicode codepoints, but multiple Unicode codepoints can combine to form a single character; so, in order to know how many characters are represented by such a sequence, we actually have to walk through the sequence and resolve it into the characters that it represents.

You, too, can walk through a String's characters. The simplest way is with the `for...in` construct (see Chapter 5). What you get when you do this are Character objects; I'll talk more about Character objects later:

```
let s = "hello"
for c in s {
    print(c) // print each Character on its own line
}
```

At an even deeper level, you can decompose a String into its UTF-8 codepoints or its UTF-16 codepoints, using the `utf8` and `utf16` properties:

```
let s = "\u{BF}Qui\u{E9}n?"
for i in s.utf8 {
    print(i) // 194, 191, 81, 117, 105, 195, 169, 110, 63
}
for i in s.utf16 {
    print(i) // 191, 81, 117, 105, 233, 110, 63
}
```

There is also a `unicodeScalars` property representing a collection (a String.UnicodeScalarView) of the String's UTF-32 codepoints expressed as UnicodeScalar structs. To illustrate, here's a utility function that turns a two-letter country abbreviation into an emoji representation of its flag:

```
func flag(country:String) -> String {
    let base : UInt32 = 127397
    var s = ""
    for v in country.unicodeScalars {
        s.unicodeScalars.append(UnicodeScalar(base + v.value)!)
    }
    return String(s)
}
// and here's how to use it:
let s = flag(country:"DE")
```

The curious thing is that there aren't more methods for standard string manipulation. How do you capitalize a string, or find out whether a string contains a given substring? Most modern programming languages have a compact, convenient way of doing things like that; Swift doesn't. The reason appears to be that missing features are provided by the Foundation framework, to which you'll always be linked in real life (importing UIKit imports Foundation). A Swift String is bridged to a Foundation NSString. This means that, to a large extent, Foundation NSString properties and methods magically spring to life whenever you are using a Swift String:

```
let s = "hello world"
let s2 = s.capitalized // "Hello World"
```

The `capitalized` property comes from the Foundation framework; it's provided by Cocoa, not by Swift. It's an NSString property; it appears tacked onto String "for free." Similarly, here's how to locate a substring of a string:

```
let s = "hello"
let range = s.range(of:"ell") // Optional(Range(...)) [details omitted]
```

I haven't explained yet what an Optional is or what a Range is (I'll talk about them later in this chapter), but that innocent-looking code has made a remarkable round-trip from Swift to Cocoa and back again: the Swift String `s` becomes an NSString, an NSString method is called, a Foundation NSRange struct is returned, and the NSRange is converted to a Swift Range and wrapped up in an Optional.

The String–NSString Element Mismatch

Swift and Cocoa have different ideas of what the elements of a string are. The Swift conception involves characters. The NSString conception involves UTF-16 codepoints. Each approach has its advantages. The NSString way makes for great speed and efficiency in comparison to Swift, which must walk the string to investigate how the characters are constructed; but the Swift way gives what you would intuitively think of as the right answer. To emphasize this difference, a nonliteral Swift string has no `length` property; its analog to an NSString's `length` is its `utf16.count`.

Fortunately, the element mismatch doesn't arise very often in practice; but it can arise. Here's a good test case:

```
let s = "Ha\u{030A}kon"
print(s.count) // 5
let length = (s as NSString).length // or: s.utf16.count
print(length) // 6
```

We've created our string (the Norwegian name Håkon) using a Unicode codepoint that combines with the previous codepoint to form a character with a ring over it. Swift walks the whole string, so it normalizes the combination and reports five characters. Cocoa just sees at a glance that this string contains six 16-bit codepoints.

Character and String Index

You are more likely to be interested in a string's characters than its codepoints. Codepoints are numbers, but what we naturally think of as characters are effectively minimal strings: a character is a single "letter" or "symbol" — formally, a *grapheme*. The equivalence between numeric codepoints and symbolic graphemes is provided, in Unicode, by the notion of a grapheme cluster. To embody this equivalence, Swift provides the Character object type (a struct), representing a single grapheme cluster.

A String (in Swift 4 and later) simply *is* a character sequence — quite literally, a Sequence of the Character objects that constitute it. That is why, as I mentioned earlier, you can walk through a string with `for...in` to obtain the String's Characters, one by one; when you do that, you're walking through the string *qua* character sequence:

```
let s = "hello"
for c in s {
    print(c) // print each Character on its own line
}
```

It isn't common to encounter Character objects outside of some character sequence of which they are a part. There isn't even a way to write a literal Character. To make a Character from scratch, initialize it from a single-character String:

```
let c = Character("h")
```

Similarly, you can pass a one-character String literal where a Character is expected, and many examples in this section will do so.

By the same token, you can initialize a String from a Character:

```
let c = Character("h")
let s = (String(c)).uppercased()
```

Characters can be compared for equality; "less than" means what you would expect it to mean.

Formally, a String is both a Sequence of Characters and a Collection of Characters. Sequence and Collection are protocols; I'll discuss protocols in Chapter 4, but what's important for now is that a String is endowed with methods and properties that it gets by virtue of being a Sequence and a Collection.

A String has a `first` and `last` property; the resulting Character is wrapped in an Optional because the string might be empty:

```
let s = "hello"
let c1 = s.first // Optional("h")
let c2 = s.last // Optional("o")
```

The `firstIndex(of:)` method locates the first occurrence of a given character within the sequence and returns its index. Again, this is an Optional, because the character might be absent:

```
let s = "hello"
let firstL = s.firstIndex(of:"l") // Optional(2)
```

All Swift indexes are numbered starting with `0`, so `2` means the third character. The index value here, however, is not an Int; I'll explain in a moment what it is and what it's good for.

A related method, `firstIndex(where:)`, takes a function that takes a Character and returns a Bool. This code locates the first character smaller than `"f"`:

```
let s = "hello"
let firstSmall = s.firstIndex {$0 < "f"} // Optional(1)
```

Those methods are matched by `lastIndex(of:)` and `lastIndex(where:)`.

A String has a `contains(_:)` method that returns a Bool, reporting whether a certain character is present:

```
let s = "hello"
let ok = s.contains("o") // true
```

Alternatively, `contains(where:)` takes a function that takes a Character and returns a Bool. This code reports whether the target string contains a vowel:

```
let s = "hello"
let ok = s.contains {"aeiou".contains($0)} // true
```

The `filter(_:)` method, too, takes a function that takes a Character and returns a Bool, effectively eliminating those characters for which `false` is returned. Here, we delete all consonants from a string:

```
let s = "hello"
let s2 = s.filter {"aeiou".contains($0)} // "eo"
```

The `dropFirst` and `dropLast` methods return, in effect, a new string without the first or last character, respectively:

```
let s = "hello"
let s2 = s.dropFirst() // "ello"
```

I say "in effect" because a method that extracts a substring returns, in reality, a Substring instance. The Substring struct is an efficient way of pointing at part of some original String, rather than having to generate a new String. When we call `s.dropFirst()` on the string `"hello"`, the resulting Substring points at the `"ello"` part of `"hello"`, which continues to exist; there is still only one string, and no new string storage memory is required.

In general, the difference between a String and a Substring will make little practical difference to you, because what you can do with a String, you can usually do also with a Substring. Nevertheless, they are different classes; this code won't compile:

```
var s = "hello"
let s2 = s.dropFirst()
s = s2 // compile error
```

To pass a Substring where a String is expected, coerce the Substring to a String explicitly:

```
var s = "hello"
let s2 = s.dropFirst()
s = String(s2)
```

You can coerce the other way, too, from a String to a Substring.

`prefix(_:)` and `suffix(_:)` extract a Substring of a given length from the start or end of the original string:

```
var s = "hello"
s = String(s.prefix(4)) // "hell"
```

`split` breaks a string up into an array, according to a function that takes a Character and returns a Bool. In this example, I obtain the words of a String, where a "word" is simplemindedly defined as a run of Characters other than a space:

```
let s = "hello world"
let arr = s.split {$0 == " "} // ["hello", "world"]
```

The result is actually an array of Substrings. If we needed to get String objects, we could apply the `map(_:)` function and coerce them all to Strings. I'll talk about `map(_:)` in Chapter 4, so you'll have to trust me for now:

```
let s = "hello world"
let arr = s.split {$0 == " "}.map {String($0)} // ["hello", "world"]
```

A String, *qua* character sequence, can also be manipulated similarly to an array. For example, you can use subscripting to obtain the character at a certain position. Unfortunately, this isn't as easy as it might be. What's the second character of `"hello"`? This doesn't compile:

```
let s = "hello"
let c = s[1] // compile error
```

The reason is that the indexes on a String are not Int values, but rather a nested type, String.Index (actually a type alias for String.CharacterView.Index). To make an object of this type is rather tricky. Start with a String's `startIndex` or `endIndex`, or with the return value from `firstIndex` or `lastIndex`; you can then call the `index(_:offsetBy:)` method to derive the index you want:

```
let s = "hello"
let ix = s.startIndex
let ix2 = s.index(ix, offsetBy:1)
let c = s[ix2] // "e"
```

The reason for this clumsy circumlocution is that Swift doesn't know where the characters of a character sequence are until it actually walks the sequence; calling `index(_:offsetBy:)` is how you make Swift do that.

To offset an index by a single position, you can obtain the next or preceding index value with the `index(after:)` and `index(before:)` methods. I could have written the preceding example like this:

```
let s = "hello"
let ix = s.startIndex
let c = s[s.index(after:ix)] // "e"
```

Another reason why it's necessary to think of a string index as an offset from the `startIndex` or `endIndex` is that those values may not be what you think they are — in particular, when you're dealing with a Substring. Consider, once again, the following:

```
let s = "hello"
let s2 = s.dropFirst()
```

Now `s2` is `"ello"`. What, then, is `s2.startIndex` (as an Int)? Not 0, but 1 — because `s2` is a Substring pointing into the original `"hello"`, where the index of the `"e"` is 1. Similarly, `s2.firstIndex(of:"o")` is not 3, but 4, because the index value is reckoned with respect to the original `"hello"`.

Once you've obtained a desired character index value, you can use it to modify the String. The `insert(contentsOf:at:)` method inserts a string into a string:

```
var s = "hello"
let ix = s.index(s.startIndex, offsetBy:1)
s.insert(contentsOf: "ey, h", at: ix) // s is now "hey, hello"
```

Similarly, `remove(at:)` deletes a single character, and also returns that character. (Manipulations involving longer character stretches require use of a Range, which is the subject of the next section.)

On the other hand, a character sequence can be coerced directly to an array of Character objects — `Array("hello")` creates an array of the characters `"h"`, `"e"`, and so on — and array indexes *are* Ints and are easy to work with. Once you've manipulated the array of Characters, you can coerce it directly to a String. I'll give an example in the next section (and I'll discuss arrays, and say more about collections and sequences, in Chapter 4).

Range

The Range object type (a struct) represents a pair of endpoints. There are two operators for forming a Range literal; you supply a start value and an end value, with one of the Range operators between them:

`...` *(closed range operator)*
: The notation `a...b` means "everything from `a` up to `b`, *including* `b`."

`..<` *(half-open range operator)*
: The notation `a..<b` means "everything from `a` up to but *not* including `b`."

Spaces around a Range operator are legal.

The types of a Range's endpoints will typically be some kind of number — most often, Ints:

```
let r = 1...3
```

If the end value is a negative literal, it has to be enclosed in parentheses or preceded by whitespace:

```
let r = -1000 ... -1
```

A very common use of a Range is to loop through numbers with `for...in`:

```
for ix in 1...3 {
    print(ix) // 1, then 2, then 3
}
```

There are no reverse Ranges: the start value of a Range can't be greater than the end value (the compiler won't stop you, but you'll crash at runtime). In practice, you can use Range's `reversed()` method to iterate from a higher value to a lower one:

```
for ix in (1...3).reversed() {
    print(ix) // 3, then 2, then 1
}
```

In Chapter 5 I'll show how to create a custom operator that effectively generates a reverse Range.

You can also use a Range's `contains(_:)` instance method to test whether a value falls within given limits:

```
let ix = // ... an Int ...
if (1...3).contains(ix) { // ...
```

For purposes of testing containment, a Range's endpoints can be Doubles:

```
let d = // ... a Double ...
if (0.1...0.9).contains(d) { // ...
```

There are also methods for learning whether two ranges overlap, and for clamping one range to another.

Another common use of a Range is to index into a sequence. Here's one way to get the second, third, and fourth characters of a String. As I suggested at the end of the preceding section, if we coerce the String to an array of Character, we can then use an Int Range as an index into that array, and coerce back to a String:

```
let s = "hello"
let arr = Array(s)
let result = arr[1...3]
let s2 = String(result) // "ell"
```

A String is itself a sequence — a character sequence — so you can use a Range to index directly into a String; but then it has to be a Range of String.Index, which, as I've already pointed out, is rather tricky to obtain. By manipulating String.Index values, you can form a Range of the proper type and use it to extract a substring by subscripting:

```
let s = "hello"
let ix1 = s.index(s.startIndex, offsetBy:1)
let ix2 = s.index(ix1, offsetBy:2)
let s2 = s[ix1...ix2] // "ell"
```

The `replaceSubrange(_:with:)` method splices into a range, modifying the string:

```
var s = "hello"
let ix = s.startIndex
let r = s.index(ix, offsetBy:1)...s.index(ix, offsetBy:3)
s.replaceSubrange(r, with: "ipp") // s is now "hippo"
```

Similarly, you can delete a stretch of characters with the `removeSubrange(_:)` method:

```
var s = "hello"
let ix = s.startIndex
let r = s.index(ix, offsetBy:1)...s.index(ix, offsetBy:3)
s.removeSubrange(r) // s is now "ho"
```

It is possible to omit one of the endpoints from a Range literal, specifying a *partial range*. There are three kinds of partial range expression, corresponding to three types of Range-like struct. To illustrate, the following expressions are identical ways of specifying the range of an entire String `s`:

```
let range1 = s.startIndex..<s.endIndex      // Range
let range2 = ..<s.endIndex                  // PartialRangeUpTo
let range3 = ...s.index(before: s.endIndex) // PartialRangeUpThrough
let range4 = s.startIndex...                // PartialRangeFrom
```

If you need to convert a partial range to a range, call `relative(to:)`. In the preceding code, `range1` and `range2.relative(to:s)` are identical. But in general you won't need to do that, because a partial range literal can be used wherever you would use a range literal. For instance, a partial range is a legal String subscript value:

```
let s = "hello"
let ix2 = s.index(before: s.endIndex)
let s2 = s[..<ix2] // "hell"
```

I'll show further practical examples later on.

Tuple

A *tuple* is a lightweight custom ordered collection of multiple values. As a type, it is expressed by surrounding the types of the contained values with parentheses, separated by a comma. Here's a declaration for a variable whose type is a tuple of an Int and a String:

```
var pair : (Int, String)
```

The literal value of a tuple is expressed in the same way — the contained values, surrounded with parentheses and separated by a comma:

```
var pair : (Int, String) = (1, "Two")
```

Those types can be inferred, so there's no need for the explicit type in the declaration:

```
var pair = (1, "Two")
```

Tuples are a pure Swift language feature; they are not compatible with Cocoa and Objective-C, so you'll use them only for values that Cocoa never sees. Within Swift, however, they have many uses. For example, a tuple is an obvious solution to the problem that a function can return only one value; a tuple *is* one value, but it *contains*

multiple values, so using a tuple as the return type of a function permits that function to return multiple values.

Tuples come with numerous linguistic conveniences. You can assign to a tuple of variable names as a way of assigning to multiple variables simultaneously:

```
let ix: Int
let s: String
(ix, s) = (1, "Two")
```

That's such a convenient thing to do that Swift lets you do it in one line, declaring and initializing multiple variables simultaneously:

```
let (ix, s) = (1, "Two")
```

To ignore one of the assigned values, use an underscore to represent it in the receiving tuple:

```
let pair = (1, "Two")
let (_, s) = pair // now s is "Two"
```

Assigning variable values to one another through a tuple swaps them safely:

```
var s1 = "hello"
var s2 = "world"
(s1, s2) = (s2, s1) // now s1 is "world" and s2 is "hello"
```

The `enumerated` method lets you walk a sequence with `for...in` and receive, on each iteration, each successive element's index number along with the element itself; this double result comes to you as — you guessed it — a tuple:

```
let s = "hello"
for (ix,c) in s.enumerated() {
    print("character \(ix) is \(c)")
}
```

I also pointed out earlier that numeric instance methods such as `addingReportingOverflow` return a tuple.

You can refer to the individual elements of a tuple directly, in two ways. The first way is by index number, using a *literal number* (not a variable value) as the name of a message sent to the tuple with dot-notation:

```
let pair = (1, "Two")
let ix = pair.0 // now ix is 1
```

If you have a `var` reference to a tuple, you can assign into it by the same means:

```
var pair = (1, "Two")
pair.0 = 2 // now pair is (2, "Two")
```

The second way to access tuple elements is to give them labels. The notation is like that of function parameters, and must appear as part of the explicit or implicit type declaration. Here's one way to establish tuple element labels:

```
let pair : (first:Int, second:String) = (1, "Two")
```

And here's another way:

```
let pair = (first:1, second:"Two")
```

The labels are now part of the type of this value, and travel with it through subsequent assignments. You can then use them as literal messages, just like (and together with) the numeric literals:

```
var pair = (first:1, second:"Two")
let x = pair.first // 1
pair.first = 2
let y = pair.0 // 2
```

The tuple generated by the `enumerated` method has labels `offset` and `element`, so we can rewrite an earlier example like this:

```
let s = "hello"
for t in s.enumerated() {
    print("character \(t.offset) is \(t.element)")
}
```

You can assign from a tuple without labels into a corresponding tuple with labels (and vice versa):

```
let pair = (1, "Two")
let pairWithNames : (first:Int, second:String) = pair
let ix = pairWithNames.first // 1
```

You can also pass, or return from a function, a tuple without labels where a corresponding tuple with labels is expected:

```
func tupleMaker() -> (first:Int, second:String) {
    return (1, "Two") // no labels here
}
let ix = tupleMaker().first // 1
```

If you're going to be using a certain type of tuple consistently throughout your program, it might be useful to give it a type name. To do so, define a type alias. In my LinkSame app, I have a Board class describing and manipulating the game layout. The board is a grid of Piece objects. I need a way to describe positions of the grid. That's a pair of integers, so I define my own type as a tuple:

```
class Board {
    typealias Point = (x:Int, y:Int)
    // ...
}
```

The advantage of that notation is that it now becomes easy to use Points throughout my code. Given a Point, I can fetch the corresponding Piece:

```
func piece(at p:Point) -> Piece? {
    let (i,j) = p
    // ... error-checking goes here ...
    return self.grid[i][j]
}
```

Still, one should not overuse tuples. In a very real sense, they are not a full-fledged type. Keep your tuples small, light, and temporary.

Void, the type of value returned by a function that doesn't return a value, is actually a type alias for an empty tuple. That's why it is also notated as `()`.

Optional

The Optional object type (an enum) wraps another object of any type. What makes an Optional optional is this: it *might* wrap another object, but then again it might not. Think of an Optional as being itself a kind of shoebox — a shoebox which can quite legally be empty.

Let's start by creating an Optional that does wrap an object. Suppose we want an Optional wrapping the String `"howdy"`. One way to create it is with the Optional initializer:

```
var stringMaybe = Optional("howdy")
```

If we log `stringMaybe` to the console with `print`, we'll see an expression identical to the corresponding initializer: `Optional("howdy")`.

After that declaration and initialization, `stringMaybe` is typed, not as a String, nor as an Optional plain and simple, but as an Optional wrapping a String. This means that any other Optional wrapping a String can be assigned to it — but not an Optional wrapping some other type. This code is legal:

```
var stringMaybe = Optional("howdy")
stringMaybe = Optional("farewell")
```

This code, however, is not legal:

```
var stringMaybe = Optional("howdy")
stringMaybe = Optional(123) // compile error
```

`Optional(123)` is an Optional wrapping an Int, and you can't assign an Optional wrapping an Int where an Optional wrapping a String is expected.

Optionals are so important to Swift that special syntax for working with them is baked into the language. The usual way to make an Optional is not to use the Optional initializer (though you can certainly do that), but to assign or pass a value of some type to a reference that is already typed as an Optional wrapping that type. This seems as if it should not be legal — but it is. Once `stringMaybe` is typed as an

Optional wrapping a String, it is legal to assign a String directly to it. The outcome is that the assigned String is wrapped in an Optional for us, automatically:

```
var stringMaybe = Optional("howdy")
stringMaybe = "farewell" // now stringMaybe is Optional("farewell")
```

We also need a way of typing something *explicitly* as an Optional wrapping a String. Otherwise, we cannot declare a variable or parameter with an Optional type. Formally, an Optional is a generic, so an Optional wrapping a String is an `Optional<String>`. (I'll explain that syntax in Chapter 4.) However, you don't have to write that. The Swift language supports syntactic sugar for expressing an Optional type: use the name of the wrapped type followed by a question mark:

```
var stringMaybe : String?
```

Thus I don't need to use the Optional initializer at all. I can type the variable as an Optional wrapping a String and assign a String into it for wrapping, all in one move:

```
var stringMaybe : String? = "howdy"
```

That, in fact, is the normal way to make an Optional in Swift.

Once you've got an Optional wrapping a particular type, you can use it wherever an Optional wrapping that type is expected — just like any other value. If a function expects an Optional wrapping a String as its parameter, you can pass `stringMaybe` as the argument:

```
func optionalExpecter(_ s:String?) {}
let stringMaybe : String? = "howdy"
optionalExpecter(stringMaybe)
```

Moreover, where an Optional wrapping a certain type of value is expected, you can pass a value of that wrapped type instead. That's because parameter passing is just like assignment: an unwrapped value will be wrapped implicitly for you. If a function expects an Optional wrapping a String, you can pass a String argument, which will be wrapped into an Optional in the received parameter:

```
func optionalExpecter(_ s:String?) {
    // ... here, s will be an Optional wrapping a String ...
    print(s)
}
optionalExpecter("howdy") // console prints: Optional("howdy")
```

But you cannot do the opposite — you cannot use an Optional wrapping a type where the wrapped type is expected. This won't compile:

```
func realStringExpecter(_ s:String) {}
let stringMaybe : String? = "howdy"
realStringExpecter(stringMaybe) // compile error
```

The error message reads: "Value of Optional type `String?` must be unwrapped." You're going to be seeing that sort of message a lot in Swift, so get used to it! If you want to use an Optional where the type of thing it wraps is expected, you must *unwrap* the Optional — that is, you must reach inside it and *retrieve* the actual thing that it wraps. Now I'm going to talk about how to do that.

Unwrapping an Optional

We have seen more than one way to wrap an object in an Optional. But what about the opposite procedure? How do we unwrap an Optional to get at the object wrapped inside it? One way is to use the *unwrap operator* (or *forced unwrap operator*), which is a postfixed exclamation mark:

```
func realStringExpecter(_ s:String) {}
let stringMaybe : String? = "howdy"
realStringExpecter(stringMaybe!)
```

In that code, the `stringMaybe!` syntax expresses the operation of reaching inside the Optional `stringMaybe`, grabbing the wrapped value, and substituting it at that point. Since `stringMaybe` is an Optional wrapping a String, the thing inside it is a String. That is exactly what the `realStringExpecter` function wants as its parameter! `stringMaybe` is an Optional *wrapping* the String `"howdy"`, but `stringMaybe!` *is* the String `"howdy"`.

If an Optional wraps a certain type, you cannot send it a message expected by that type. You must unwrap it first. Let's try to get an uppercase version of `stringMaybe`:

```
let stringMaybe : String? = "howdy"
let upper = stringMaybe.uppercased() // compile error
```

The solution is to unwrap `stringMaybe` to get at the String inside it. We can do this directly, in place, using the unwrap operator:

```
let stringMaybe : String? = "howdy"
let upper = stringMaybe!.uppercased()
```

If an Optional is to be used several times where the unwrapped type is expected, and if you're going to be unwrapping it with the unwrap operator each time, your code can quickly start to look like the dialog from a 1960s Batman comic. For example, an app's window is an Optional UIWindow property (`self.window`):

```
// self.window is an Optional wrapping a UIWindow
self.window!.rootViewController = RootViewController()
self.window!.backgroundColor = UIColor.white
self.window!.makeKeyAndVisible()
```

That sort of thing soon gets old (or silly). One obvious alternative is to assign the unwrapped value *once* to a variable of the wrapped type and then use that variable:

```
// self.window is an Optional wrapping a UIWindow
let window = self.window!
// now window (not self.window) is a UIWindow, not an Optional
window.rootViewController = RootViewController()
window.backgroundColor = UIColor.white
window.makeKeyAndVisible()
```

Implicitly unwrapped Optional

Swift provides another way of using an Optional where the wrapped type is expected: you can declare the Optional *type* as being *implicitly unwrapped*. An implicitly unwrapped Optional is an Optional, but the compiler permits some special magic associated with it: its value can be used *directly* where the wrapped type is expected. You *can* unwrap an implicitly unwrapped Optional explicitly, but you don't have to, because it will be unwrapped for you, automatically, if you try to use it where the wrapped type is expected. Moreover, Swift provides syntactic sugar for expressing an implicitly unwrapped Optional type: use the name of the wrapped type followed by an exclamation mark:

```
func realStringExpecter(_ s:String) {}
var stringMaybe : String! = "howdy"
realStringExpecter(stringMaybe) // no problem
```

Bear in mind that *an implicitly unwrapped Optional is still an Optional.* It's just a convenience. By declaring something as an implicitly unwrapped Optional, you are asking the compiler, if you happen to use this value where the wrapped type is expected, to forgive you and to unwrap the value for you.

In reality, an implicitly unwrapped Optional type is not really a distinct type; it is merely an Optional marked in a special way that allows it to be used where the unwrapped type is expected. For this reason, implicit unwrapping does not propagate by assignment. Here's a case in point. If `self` is a UIViewController, then `self.view` is typed as `UIView!`. As a result, this expression is legal (assume `v` is a UIView):

```
self.view.addSubview(v)
```

But this is not legal:

```
let mainview = self.view
mainview.addSubview(v) // compile error
```

The problem is that, although `self.view` is an implicitly unwrapped Optional wrapping a UIView, `mainview` is a *normal* Optional wrapping a UIView, and so it would have to be unwrapped explicitly before you could send it the `addSubview` message. Alternatively, you could unwrap the implicitly unwrapped Optional explicitly at the outset:

```
let mainview = self.view!
mainview.addSubview(v)
```

In real life, the primary situation in which you're likely to declare an implicitly unwrapped Optional is when an instance property's initial value can't be provided until after the instance itself is created. I'll give some examples at the end of this chapter.

The keyword nil

I have talked so far about Optionals that contain a wrapped value. But what about an Optional that *doesn't* contain any wrapped value? Such an Optional is, as I've already said, a perfectly legal entity; that, indeed, is the whole point of Optionals.

You are going to need a way to *ask* whether an Optional contains a wrapped value, and a way to *specify* an Optional *without* a wrapped value. Swift makes both of those things easy, through the use of a special keyword, `nil`:

To learn whether an Optional contains a wrapped value
: Test the Optional for equality against `nil`. If the test succeeds, the Optional is empty. An empty Optional is also reported in the console as `nil`.

To specify an Optional with no wrapped value
: Assign or pass `nil` where the Optional type is expected. The result is an Optional of the expected type, containing no wrapped value.

To illustrate:

```
var stringMaybe : String? = "Howdy"
print(stringMaybe) // Optional("Howdy")
if stringMaybe == nil {
    print("it is empty") // does not print
}
stringMaybe = nil
print(stringMaybe) // nil
if stringMaybe == nil {
    print("it is empty") // prints
}
```

The keyword `nil` lets you express the concept, "an Optional wrapping the appropriate type, but not actually containing any object of that type." Clearly, that's very convenient magic; you'll want to take advantage of it. It is very important to understand, however, that it *is* magic: `nil` in Swift is *not* a thing and is *not* a value. *It is a shorthand.* It is natural to think and speak as if this shorthand were real. I will often say that something "is `nil`." But in reality, nothing "is `nil`"; `nil` isn't a thing. What I really mean is that this thing is equatable with `nil`, because it is an Optional not wrapping anything. (I'll explain in Chapter 4 how `nil`, and Optionals in general, really work.)

Because a variable typed as an Optional can be `nil`, Swift follows a special initialization rule: a variable (`var`) typed as an Optional *is* `nil`, automatically:

```
func optionalExpecter(_ s:String?) {}
var stringMaybe : String?
optionalExpecter(stringMaybe)
```

That code looks as if it should be illegal. We declared a variable `stringMaybe`, but we never assigned it a value. Nevertheless we are now passing it around as if it were an actual thing. That's because it *is* an actual thing. This variable has been *implicitly initialized* — to `nil`. A variable (`var`) typed as an Optional is the *only* sort of variable that gets implicit initialization in Swift.

We come now to perhaps the most important rule in all of Swift: You *cannot unwrap an Optional containing nothing* (an Optional equatable with `nil`). Such an Optional contains nothing; there's nothing to unwrap. Like Oakland, there's no there there. In fact, explicitly unwrapping an Optional containing nothing will *crash your program* at runtime:

```
var stringMaybe : String?
let s = stringMaybe! // crash
```

The crash message reads: "Fatal error: unexpectedly found `nil` while unwrapping an Optional value." Get used to it, because you're going to be seeing it a lot. This is an easy mistake to make. Unwrapping an Optional that contains no value is, in fact, probably the most common way to crash a Swift program. You should look upon this kind of crash as a blessing. Very often, in fact, you will *want* to crash if your Optional contains no value, because it *should* contain a value, and the fact that it doesn't indicates that you've made a mistake elsewhere.

In the long run, however, crashing is bad. To eliminate this kind of crash, you need to ensure that your Optional contains a value, and *don't* unwrap it if it doesn't! Ensuring that an Optional contains a value before attempting to unwrap it is clearly a very important thing to do. Accordingly, Swift provides several convenient ways of doing it. I'll describe some of them now, and I'll discuss others in Chapter 5.

One obvious approach is to test your Optional against `nil` explicitly before you unwrap it:

```
var stringMaybe : String?
// ... stringMaybe might be assigned a real value here ...
if stringMaybe != nil {
    let s = stringMaybe!
    // ...
}
```

But there's a more elegant way, as I shall now explain.

Optional chains

A common situation is that you want to send a message to the value wrapped inside an Optional. You *cannot* send such a message to the Optional *itself*. If you try to do so, you will get an error message from the compiler:

```
let stringMaybe : String? = "howdy"
let upper = stringMaybe.uppercased() // compile error
```

You must unwrap the Optional first, so that you can send that message to the *actual* thing wrapped inside. Conveniently, you can unwrap the Optional *in place*. I gave an example earlier:

```
let stringMaybe : String? = "howdy"
let upper = stringMaybe!.uppercased()
```

That form of code is called an *Optional chain*. In the middle of a chain of dot-notation, you have unwrapped an Optional.

However, if you unwrap an Optional that contains no wrapped object, you'll crash. So what if you're *not sure* whether this Optional contains a wrapped object? How can you send a message to the value inside an Optional in that situation?

Swift provides a special shorthand for exactly this purpose. To send a message *safely* to the value wrapped inside an Optional that might be empty, you can *unwrap the Optional optionally*. To do so, unwrap the Optional with the question mark postfix operator instead of the exclamation mark:

```
var stringMaybe : String?
// ... stringMaybe might be assigned a real value here ...
let upper = stringMaybe?.uppercased()
```

That's an Optional chain in which you used a question mark to unwrap the Optional. By using that notation, you have unwrapped the Optional optionally — meaning conditionally. The condition in question is one of safety; a test for `nil` is performed for us. Our code means: "If `stringMaybe` contains a String, unwrap it and send that String the `uppercased` message. If it doesn't (that is, if it equates to `nil`), *do not* unwrap it and *do not* send it any messages!"

Such code is a double-edged sword. On the one hand, if `stringMaybe` is `nil`, you won't crash at runtime. On the other hand, if `stringMaybe` is `nil`, that line of code won't do anything useful — you won't get any uppercase string.

But now there's a new question. In that code, we initialized a variable `upper` to an expression that involves sending the `uppercased` message. Now it turns out that the `uppercased` message might not even be sent. So what, exactly, is `upper` initialized *to*?

To handle this situation, Swift has a special rule. If an Optional chain contains an optionally unwrapped Optional, and if this Optional chain produces a value, that value is itself *wrapped in an Optional*. Thus, `upper` is typed as an Optional wrapping a

String. This works brilliantly, because it covers both possible cases. Let's say, first, that `stringMaybe` contains a String:

```
var stringMaybe : String?
stringMaybe = "howdy"
let upper = stringMaybe?.uppercased()
```

After that code, `upper` is *not* a String; it is *not* `"HOWDY"`. It is an Optional wrapping `"HOWDY"`.

On the other hand, if the attempt to unwrap the Optional fails, the Optional chain can return `nil` instead:

```
var stringMaybe : String?
let upper = stringMaybe?.uppercased()
```

After that code, `upper` is typed as an Optional wrapping a String, but it wraps no string; its value is `nil`.

Unwrapping an Optional optionally in this way is elegant and safe; even if `stringMaybe` is `nil`, we won't crash at runtime. On the other hand, we've ended up with yet another Optional on our hands! `upper` is typed as an Optional wrapping a String, and in order to use that String, we're going to have to unwrap `upper`. And we don't know whether `upper` is `nil`, so we have exactly the same problem we had before — we need to make sure that we unwrap `upper` safely, and that we don't accidentally unwrap an empty Optional.

Longer Optional chains are legal. No matter how many Optionals are unwrapped in the course of the chain, if any of them is unwrapped optionally, the entire expression produces an Optional wrapping the type it would have produced if the Optionals were unwrapped normally, and is free to fail safely at any point along the way:

```
// self is a UIViewController
let f = self.view?.window?.rootViewController?.view?.frame
```

The `frame` property of a view is a CGRect. But after that code, `f` is *not* a CGRect. It's an Optional wrapping a CGRect. If *any* of the optional unwrapping along the chain fails (because the Optional we propose to unwrap is `nil`), `f` will be `nil` to indicate failure.

(Observe that the preceding code does *not* end up nesting Optionals; it doesn't produce a CGRect wrapped in an Optional wrapped in an Optional, and so on, merely because there are multiple Optionals being optionally unwrapped in the chain! However, it is possible, for other reasons, to end up with an Optional wrapped in an Optional, and I'll call out some examples as we proceed.)

If a function call returns an Optional, you can unwrap the result and use it. You don't necessarily have to capture the result in order to do that; you can unwrap it in place, by putting an exclamation mark or a question mark after the function call (that is,

after the closing parenthesis). That's really no different from what we've been doing all along, except that instead of an Optional property or variable, this is a function call that returns an Optional:

```
class Dog {
    var noise : String?
    func speak() -> String? {
        return self.noise
    }
}
let d = Dog()
let bigname = d.speak()?.uppercased()
```

After that, don't forget, `bigname` is not a String — it's an Optional wrapping a String.

You can also assign safely into an Optional chain. If any of the optionally unwrapped Optionals in the chain turns out to be `nil`, nothing happens:

```
// self is a UIViewController
self.navigationController?.hidesBarsOnTap = true
```

A view controller might or might not have a navigation controller, so its `navigationController` property is an Optional. In that code, we are setting our navigation controller's `hidesBarsOnTap` property safely; if we happen to have no navigation controller, no harm is done — because nothing happens.

When assigning into an Optional chain, if you also want to know whether the assignment succeeded, you can capture the result of the assignment as an Optional wrapping a Void and test it for `nil`:

```
let ok : Void? = self.navigationController?.hidesBarsOnTap = true
```

Now, if `ok` is not `nil`, `self.navigationController` was safely unwrapped and the assignment succeeded.

It is also possible to assign into an Optional chain where the chain ends with a question mark; if the chain is `nil`, no assignment takes place. For example, a UILabel has an Optional String `text` property:

```
let label = UILabel() // label.text is nil
label.text? = "testing" // label.text is _still_ nil!
```

But I can think of no practical use case, so perhaps this should be regarded as a curiosity.

The `!` and `?` postfix operators, which are used to unwrap an Optional, have basically *nothing* to do with the `!` and `?` used with type names as syntactic sugar for expressing Optional types (such as `String?` and `String!`). The outward similarity has confused many a beginner.

Optional map and flatMap

When you want to do something to an Optional's wrapped value more elaborate than sending it a simple message, such as calling `uppercased()`, while keeping the advantages of Optional chaining, Swift provides a method that elegantly and safely permits you to do so: `map(_:)`. This is a method of Optional itself, so it's fine to send it to an Optional. The parameter is a function that you supply (usually an anonymous function) that takes whatever type is wrapped in the Optional; the *unwrapped* value is passed to this function, and now you can manipulate it in any desired manner. The result of the function is then wrapped as an Optional. If the original Optional was `nil`, the whole thing produces `nil`, safely:

```
let s : String? = "howdy"
let s2 = s.map {($0 + ", world").uppercased()}
```

In that example, we start with an Optional wrapping a String; we append a string to the string, and uppercase the result. You can't apply the `+` operator to an Optional string, but inside the `map` function, the string is *not* Optional. Afterward, `s2` is an Optional wrapping a String. If `s` had turned out to be `nil`, there would be no crash, and `s2` would be set to `nil` as well.

The output Optional type doesn't have to be the same as the input Optional type. To illustrate, I'll use a closely related Optional method, `flatMap(_:)`. Here's an elegant way to coerce an Optional String to an (Optional) Int:

```
let s : String? = // whatever
let i = s.flatMap {Int($0)}
```

In that code, we attempt to unwrap an Optional String and coerce it to an Int. The result is an Optional Int, which will be `nil` if `s` is `nil`, or if `s` isn't `nil` but the coercion fails because the string wrapped by `s` doesn't represent an integer.

That example also illustrates the difference between `map` and `flatMap`. If the map function itself produces an Optional — as coercing a String to an Int does — `flatMap` unwraps it before wrapping the result in an Optional. `map` doesn't do that, so if we had used `map` here, we would have ended up with a double-wrapped Optional (an `Int??`).

Comparison with Optional

In an equality comparison with something other than `nil`, an Optional gets special treatment: the wrapped value, not the Optional itself, is compared. This works:

```
let s : String? = "Howdy"
if s == "Howdy" { // ... they _are_ equal!
```

That shouldn't work — how can an Optional be the same as a String? — but it does. Instead of comparing the Optional itself with `"Howdy"`, Swift automagically (and

safely) compares its wrapped value (if there is one) with `"Howdy"`. If the wrapped value is `"Howdy"`, the comparison succeeds. If the wrapped value is not `"Howdy"`, the comparison fails. If there is *no* wrapped value (`s` is `nil`), the comparison fails too — safely! You can compare `s` to `nil` or to a String, and the comparison works correctly in all cases.

(This feature depends upon the wrapped type itself being usable with ==. This means that the wrapped type must adopt the Equatable protocol; otherwise, the compiler will stop you from using == with an Optional wrapping it. I'll talk about protocols and Equatable in Chapters 4 and 5.)

Direct comparison of Optionals does *not* work for an inequality comparison, using the greater-than and less-than operators:

```
let i : Int? = 2
if i < 3 { // compile error
```

To perform that sort of comparison, you can unwrap safely and perform the comparison directly on the unwrapped value:

```
if i != nil && i! < 3 { // ... it _is_ less
```

Do not compare an implicitly unwrapped Optional with anything; you can crash at runtime.

Why Optionals?

Now that you know *how* to use an Optional, you are probably wondering *why* to use an Optional. Why does Swift have Optionals at all? What are they good for?

One important use of Optionals is to permit a value to be *marked as empty or erroneous*. Many built-in Swift functions use an Optional this way:

```
let arr = [1,2,3]
let ix = arr.firstIndex(of:4)
if ix == nil { // ...
```

Swift's `firstIndex(of:)` method returns an Optional because the object sought might not be present, in which case it has *no* index. The type returned cannot be an Int, because there is no Int value that can be taken to mean, "I didn't find this object at all." Returning an Optional solves the problem neatly: `nil` means "I didn't find the object," and otherwise the actual Int result is sitting there wrapped up in the Optional.

Another purpose of Optionals is to provide *interchange of object values with Objective-C*. In Objective-C, *any* object reference can be `nil`. You need a way to send `nil` to Objective-C and to receive `nil` from Objective-C. Swift Optionals provide your only way to do that.

Swift will typically assist you by a judicious use of appropriate types in the Cocoa APIs. Consider a UIView's `backgroundColor` property. It's a UIColor, but it can be `nil`, and you are allowed to set it to `nil`. Thus, it is typed as `UIColor?`. You don't need to work directly with Optionals in order to *set* such a value! Remember, assigning the wrapped type to an Optional is legal, as the assigned value will be wrapped for you. You can set `myView.backgroundColor` to a UIColor — or to `nil`. If you *get* a UIView's `backgroundColor`, you now have an Optional wrapping a UIColor, *and you must be conscious of that fact*, for all the reasons I've already discussed: if you're not, surprising things can happen:

```
let v = UIView()
let c = v.backgroundColor
let c2 = c.withAlphaComponent(0.5) // compile error
```

You're trying to send the `withAlphaComponent` message to `c`, as if it were a UIColor. It *isn't* a UIColor. It's an Optional wrapping a UIColor. Xcode will try to help you in this situation; if you use code completion (Chapter 10) to enter the name of the `withAlphaComponent` method, Xcode will insert a question mark after `c`, (optionally) unwrapping the Optional and giving you legal code:

```
let v = UIView()
let c = v.backgroundColor
let c2 = c?.withAlphaComponent(0.5)
```

In the vast majority of situations, however, a Cocoa object type will *not* be marked as an Optional. That's because, although in theory it *could* be `nil` (because any Objective-C object reference can be `nil`), in practice it won't be. Swift saves you a step by treating the value as the object type itself. This magic is performed by hand-tweaking the Cocoa APIs (also called *auditing*). In the very first public version of Swift (in June of 2014), *all* object values received from Cocoa were typed as Optionals (usually implicitly unwrapped Optionals); but then Apple embarked on the massive project of hand-tweaking the APIs to eliminate Optionals that didn't need to be Optionals, and that project is now essentially complete.

Finally, an important use of Optionals is to *defer initialization* of an instance property. If a variable (declared with `var`) is typed as an Optional, it has a value even if you don't initialize it — namely `nil`. That comes in very handy in situations where you know something *will* have a value, but not right away.

One way this can happen is that a property represents data that will take time to acquire. In my Albumen app, as we launch, I create an instance of my root view controller. I also want to gather a bunch of data about the user's music library and store that data in instance properties of the root view controller instance. But gathering that data will take time. Therefore I must instantiate the root view controller *first* and gather the data *later*, because if we pause to gather the data *before* instantiating the root view controller, the app will take too long to launch — the delay will be

perceptible, and we might even crash (because iOS forbids long launch times). Therefore the data properties are all typed as Optionals; they are `nil` until the data are gathered, at which time they are assigned their "real" values:

```
class RootViewController : UITableViewController {
    var albums : [MPMediaItemCollection]? // initialized to nil
    // ...
}
```

This approach has a second advantage: as with `firstIndex`, the initial `nil` value of `albums` is a signal to the rest of my code that we don't yet have a real value. When my Albumen app launches, it displays a table listing all the user's music albums. At launch time, however, that data has not yet been gathered. My table-display code tests `albums` to see whether it's `nil` and, if it is, displays an empty table. After gathering the data, I tell my table to display its data *again*. This time, the table-display code finds that `albums` is *not* `nil`, but rather consists of actual data — and it now displays that data. The use of an Optional allows one and the same value, `albums`, to store the data or to state that there is no data.

Sometimes, a property's value isn't time-consuming to acquire, but it *still* won't be ready at initialization time. A common case in real life is an outlet, which is a reference to something in your interface such as a button:

```
class ViewController: UIViewController {
    @IBOutlet var myButton: UIButton! // initialized to nil
    // ...
}
```

Ignore, for now, the `@IBOutlet` designation, which is an internal hint to Xcode (as I'll explain in Chapter 8). The important thing is that this property, `myButton`, won't have a value when our ViewController instance first comes into existence, but shortly thereafter the view controller's view will be loaded and `myButton` will be set so that it points to an actual UIButton object in the interface. Therefore, the variable is typed as an implicitly unwrapped Optional:

- It's an Optional because we need a placeholder value (namely `nil`) for `myButton` when the ViewController instance first comes into existence.
- It's implicitly unwrapped so that in our code, once `self.myButton` has been assigned a UIButton value, we can treat it as a reference to an actual UIButton, passing through the Optional without noticing that it *is* an Optional.

In real life, nearly all of this view controller's code will run after the view is loaded and the actual button is assigned to `myButton`. So the implicitly unwrapped Optional is generally safe: code can confidently refer to `myButton` without fear that it might be `nil`. So an implicitly unwrapped Optional is appropriate here.