Chapter 1. The Architecture of Swift

It will be useful at the outset for you to have a general sense of how the Swift language is constructed and what a Swift-based iOS program looks like. This chapter will survey the overall architecture and nature of the Swift language. Subsequent chapters will fill in the details.

Ground of Being

A complete Swift command is a *statement*. A Swift text file consists of multiple *lines* of text. Line breaks are meaningful. The typical layout of a program is one statement, one line:

```
print("hello")

print("world")
```

(The `print` command provides instant feedback in the Xcode console.)

You can combine more than one statement on a line, but then you need to put a semicolon between them:

```
print("hello"); print("world")
```

You are free to put a semicolon at the end of a statement that is last or alone on its line, but no one ever does (except out of habit, because C and Objective-C *require* the semicolon):

```
print("hello");

print("world");
```

Conversely, a single statement can be broken into multiple lines, in order to prevent long statements from becoming long lines. But you should try to do this at sensible places so as not to confuse Swift. For example, after an opening parenthesis is a good place:

```
print(
    "world")
```

Comments are everything after two slashes in a line (so-called C++-style comments):

```
print("world") // this is a comment, so Swift ignores it
```

You can also enclose comments in `/*...*/`, as in C. Unlike C, C-style comments can be nested.

Many constructs in Swift use curly braces as delimiters:

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

By convention, the contents of curly braces are preceded and followed by line breaks and are indented for clarity, as shown in the preceding code. Xcode will help impose this convention, but the truth is that Swift doesn't care, and layouts like this are legal (and are sometimes more convenient):

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

Swift is a *compiled* language. This means that your code must *build* — passing through the compiler and being turned from text into some lower-level form that a computer can understand — before it can *run* and actually do the things it says to do. The Swift compiler is very strict; in the course of writing a program, you will often try to build and run, only to discover that you can't even build in the first place, because the compiler will flag some *error*, which you will have to fix if you want the code to run. Less often, the compiler will let you off with a *warning*; the code can run, but in general you should take warnings seriously and fix whatever they are telling you about. The strictness of the compiler is one of Swift's greatest strengths, and provides your code with a large measure of audited correctness even before it ever runs.

The Swift compiler's error and warning messages, however, range from the insightful to the obtuse to the downright misleading. You will often know that *something* is wrong with a line of code, but the Swift compiler will not be telling you clearly exactly *what* is wrong or even *where* in the line to focus your attention. My advice in these situations is to pull the line apart into several lines of simpler code until you reach a point where you can guess what the issue is. Try to love the compiler despite the occasional unhelpful nature of its messages. Remember, it knows more than you do, even if it is sometimes rather inarticulate about its knowledge.

Everything Is an Object?

In Swift, "everything is an object." That's a boast common to various modern object-oriented languages, but what does it mean? Well, that depends on what you mean by "object" — and what you mean by "everything."

Let's start by stipulating that an object, roughly speaking, is something you can send a message to. A message, roughly speaking, is an imperative instruction. For example, you can give commands to a dog: "Bark!" "Sit!" In this analogy, those phrases are messages, and the dog is the object to which you are sending those messages.

In Swift, the syntax of message-sending is *dot-notation*. We start with the object; then there's a dot (a period); then there's the message. (Some messages are also followed by parentheses, but ignore them for now; the full syntax of message-sending is one of those details we'll be filling in later.) This is valid Swift syntax:

```
fido.bark()

rover.sit()
```

The idea of *everything* being an object is a way of suggesting that even "primitive" linguistic entities can be sent messages. Take, for example, `1`. It appears to be a literal digit and no more. It will not surprise you, if you've ever used any programming language, that you can say things like this in Swift:

```
let sum = 1 + 2
```

But it *is* surprising to find that `1` can be followed by a dot and a message. This is legal and meaningful in Swift (don't worry about what it actually means):

```
let s = 1.description
```

But we can go further. Return to that innocent-looking `1 + 2` from our earlier code. It turns out that this is actually a kind of syntactic trickery, a convenient way of expressing and hiding what's really going on. Just as `1` is actually an object, `+` is actually a message; but it's a message with special syntax (*operator* syntax). In Swift, every noun is an object, and every verb is a message.

Perhaps the ultimate acid test for whether something is an object in Swift is whether you can modify it. An object type can be *extended* in Swift, meaning that you can define your own messages on that type. For example, you can't normally send the `sayHello` message to a number. But you can change a number type so that you can:

```
extension Int {

    func sayHello() {

        print("Hello, I'm \(self)")

    }

}

1.sayHello() // outputs: "Hello, I'm 1"
```

In Swift, then, `1` is an object. In some languages, such as Objective-C, it clearly is not; it is a “primitive” or *scalar* built-in data type. So the distinction being drawn here is between object types on the one hand and scalars on the other. In Swift, there are no scalars; *all* types are ultimately object types. That’s what “everything is an object” really means.

Three Flavors of Object Type

If you know Objective-C or some other object-oriented language, you may be surprised by Swift’s notion of what *kind* of object `1` is. In many languages, such as Objective-C, an object is a *class* or an instance of a class (I’ll explain later what an instance is). Swift has classes, but `1` in Swift is not a class or an instance of a class: the type of `1`, namely Int, is a *struct*, and `1` is an instance of a struct. And Swift has yet another kind of thing you can send messages to, called an *enum*.

So Swift has three kinds of object type: classes, structs, and enums. I like to refer to these as the three *flavors* of object type. Exactly how they differ from one another will emerge in due course. But they are all very definitely object types, and their similarities to one another are far stronger than their differences. For now, just bear in mind that these three flavors exist.

(The fact that a struct or enum is an object type in Swift will surprise you particularly if you know Objective-C. Objective-C has structs and enums, but they are not objects. Swift structs, in particular, are much more important and pervasive than Objective-C structs. This difference between how Swift views structs and enums and how Objective-C views them can matter when you are talking to Cocoa.)

Variables

A variable is a *name* for an object. Technically, it *refers* to an object; it is an object *reference*. Nontechnically, you can think of it as a shoebox into which an object is placed. The object may undergo changes, or it may be replaced inside the shoebox by another object, but the name has an integrity all its own. The object to which the variable refers is the variable’s *value*.

In Swift, no variable comes implicitly into existence; all variables must be *declared*. If you need a name for something, you must say “I’m creating a name.” You do this with one of two keywords: `let` or `var`. In Swift, declaration is usually accompanied by *initialization* — you use an equal sign to give the variable a value immediately, as part of the declaration. These are both variable declarations (and initializations):

```
let one = 1

var two = 2
```

Once the name exists, you are free to use it. For example, we can change the value of `two` to be the same as the value of `one`:

```
let one = 1

var two = 2

two = one
```

The last line of that code uses both the name `one` and the name `two` declared in the first two lines: the name `one`, on the right side of the equal sign, is used merely to *refer* to the value inside the shoebox `one` (namely `1`); but the name `two`, on the left side of the equal sign, is used to *replace* the value inside the shoebox `two`. A statement like that, with a variable name on the left side of an equal sign, is called an *assignment*, and the equal sign is the *assignment operator*. The equal sign is not an assertion of equality, as it might be in an algebraic formula; it is a command. It means: “Get the value of what’s on the right side of me, and use it to replace the value of what’s on the left side of me.”

The two kinds of variable declaration differ in that a name declared with `let` *cannot have its value replaced*. A variable declared with `let` is a *constant*; its value is assigned once and stays. This won’t even compile:

```
let one = 1

var two = 2

one = two // compile error
```

It is always possible to declare a name with `var` to give yourself the most flexibility, but if you know you're never going to replace the initial value of a variable, it's better to use `let`, as this permits Swift to behave more efficiently — so much more efficiently, in fact, that the Swift compiler will actually call your attention to any case of your using `var` where you could have used `let`, offering to change it for you.

Variables also have a *type*. This type is established when the variable is declared and *can never change*. For example, this won't compile:

```
var two = 2

two = "hello" // compile error
```

Once `two` is declared and initialized as `2`, it is a number (properly speaking, an Int) and it must always be so. You can replace its value with `1` because that's also an Int, but you can't replace its value with `"hello"` because that's a string (properly speaking, a String) — and a String is not an Int.

Variables literally have a life of their own — more accurately, a *lifetime* of their own. As long as a variable exists, it keeps its value alive. Thus, a variable can be not only a way of conveniently *naming* something, but also a way of *preserving* it. I'll have more to say about that later.

WARNING

By convention, type names such as String or Int (or Dog or Cat) start with a capital letter; variable names start with a small letter. *Do not violate this convention.* If you do, your code might still compile and run just fine, but I will personally send agents to your house to remove your kneecaps in the dead of night.

Functions

Executable code, like `fido.bark()` or `one = two`, cannot go just anywhere in your program. (Failure to appreciate this fact is a common beginner mistake, and can result in a mysterious compile error message such as "Expected declaration.") In general, executable code must live inside the body of a *function*. A function is a batch of code that can be told, as a batch, to run. Typically, a function has a name, and it gets that name through a function declaration. Function declaration syntax is another of those details that will be filled in later, but here's an example:

```
func go() {

    let one = 1

    var two = 2

    two = one

}
```

That describes a sequence of things to do — declare `one`, declare `two`, change the value of `two` to match the value of `one` — and it gives that sequence a *name*, `go`; but it doesn't *perform* the sequence. The sequence is performed when someone *calls* the function. Thus, we might say, elsewhere:

```
go()
```

That is a command to the `go` function that it should actually run. But again, that command is itself executable code, so it cannot live on its own either. It might live in the body of a different function:

```
func doGo() {

    go()

}
```

But wait! This is getting a little nutty. That, too, is just a function declaration; to run it, someone must call `doGo` by saying `doGo()` — and that's executable code too. This seems like some kind of infinite regression; it looks like none of our code will *ever* run. If all executable code has to live in a function, who will tell *any* function to run? The initial impetus must come from somewhere.

In real life, fortunately, this regression problem doesn't arise. Remember that your goal is ultimately to write an iOS app. Thus, your app will be run on an iOS device (or the Simulator) by a runtime that already wants to call certain functions. So you start by writing special functions that you know the runtime itself will call. That gives your app a way to get started and gives you places to put functions that will be called by the runtime at key moments — such as when the app launches, or when the user taps a button in your app's interface.

TIP

Swift also has a special rule that a file called *main.swift*, exceptionally, *can* have executable code at its top level, outside any function body, and this is the code that actually runs when the program runs. You can construct your app with a *main.swift* file, but in general you won't need to.

The Structure of a Swift File

A Swift program can consist of one file or many files. In Swift, a file is a meaningful unit, and there are definite rules about the structure of the Swift code that can go inside it. (I'm assuming that we are *not* in a *main.swift* file.) Only certain things can go at the top level of a Swift file — chiefly the following:

Module `import` statements

> A module is an even higher-level unit than a file. A module can consist of multiple files, and these can all see each other automatically; but a module can't see another module without an `import` statement. For example, that is how you are able to talk to

Cocoa in an iOS program: the first line of your file says `import UIKit`.

Variable declarations

A variable declared at the top level of a file is a *global* variable: all code will be able to see and access it, without explicitly sending a message to any object, and it lives as long as the program runs.

Function declarations

A function declared at the top level of a file is a *global* function: all code will be able to see and call it, without explicitly sending a message to any object.

Object type declarations

The declaration for a class, a struct, or an enum.

For example, this is a legal Swift file containing (just to demonstrate that it can be done) an `import` statement, a variable declaration, a function declaration, a class declaration, a struct declaration, and an enum declaration:

```
import UIKit

var one = 1

func changeOne() {

}

class Manny {

}

struct Moe {

}

enum Jack {

}
```

That's a very silly and mostly empty example, but remember, our goal is to survey the parts of the language and the structure of a file, and the example shows them.

Furthermore, the curly braces for each of the things in that example can all have variable declarations, function declarations, and object type declarations within them! Indeed, *any* structural curly braces can contain such declarations.

You'll notice that I did *not* say that executable code can go at the top level of a file. That's because it can't! *Only a function body can contain executable code.* A statement like `one = two` or `print(name)` is executable code, and can't go at the top level of a file. But in our previous example, `func changeOne()` is a function declaration, so executable code *can* go inside its curly braces, because they constitute a function body:

```
var one = 1

// executable code can't go here

func changeOne() {

    let two = 2 // executable code

    one = two   // executable code

}
```

Executable code also can't go directly inside the curly braces that accompany the `class Manny` declaration; that's the top level of a class declaration, not a function body. But a class declaration *can* contain a function declaration, and that function declaration *can* contain executable code:

```
class Manny {

    let name = "manny"

    // executable code can't go here
```

```
        func sayName() {

            print(name) // executable code

        }

    }
```

To sum up, Example 1-1 is a legal Swift file, schematically illustrating the structural possibilities. (Ignore the hanky-panky with the `name` variable declaration inside the enum declaration for Jack; enum top-level variables have some special rules that I'll explain later.)

Example 1-1. Schematic structure of a legal Swift file

```
import UIKit
var one = 1
func changeOne() {
    let two = 2
    func sayTwo() {
        print(two)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
    one = two
}
class Manny {
    let name = "manny"
    func sayName() {
        print(name)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
}
struct Moe {
    let name = "moe"
    func sayName() {
        print(name)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
}
enum Jack {
    var name : String {
        return "jack"
    }
    func sayName() {
        print(name)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
}
```

Obviously, we can recurse down as far we like: we could have a class declaration containing a class declaration containing a class declaration, and so on. But there’s no point illustrating *that*.

Scope and Lifetime

In a Swift program, things have a *scope*. This refers to their ability to be seen by other things. Things are nested inside of other things, making a nested hierarchy of things. The rule is that things can see things *at their own level and at a higher level containing them*. The levels are:

- A module is a scope.
- A file is a scope.
- Curly braces are a scope.

When something is declared, it is declared at some level within that hierarchy. Its place in the hierarchy — its scope — determines whether it can be seen by other things.

Look again at Example 1-1. Inside the declaration of Manny is a `name` variable declaration and a `sayName` function declaration; the code *inside* `sayName`’s curly braces can see things *outside* those curly braces *at a higher containing level*, and can therefore see the `name` variable. Similarly, the code inside the body of the `changeOne` function can see the `one` variable declared at the top level of the file; indeed, *everything* throughout this file can see the `one` variable declared at the top level of the file.

Scope is thus a very important way of *sharing information*. Two different functions declared inside Manny would *both* be able to see the `name` declared at Manny’s top level. Code inside Jack and code inside Moe can *both* see the `one` declared at the file’s top level.

Things also have a *lifetime*, which is effectively equivalent to their scope. A thing lives as long as its surrounding scope lives. Thus, in Example 1-1, the variable `one` lives as long as the file lives — namely, as long the program runs. It is global *and persistent*. But the variable `name` declared at the top level of Manny exists only so long as a Manny instance exists (I'll talk in a moment about what that means).

Things declared at a deeper level live even shorter lifetimes. Consider this code:

```
func silly() {
    if true {
        class Cat {}
        var one = 1
        one = one + 1
    }
}
```

That code is silly, but it's legal: remember, I said that variable declarations, function declarations, and object type declarations can appear in *any* structural curly braces. In that code, the class Cat and the variable `one` will not even come into existence until someone calls the `silly` function, and even then they will exist only during the brief instant that the path of code execution passes through the if construct. So, suppose the function `silly` is called; the path of execution then enters the if construct. Here, Cat is declared and comes into existence; then `one` is declared and comes into existence; then the executable line `one = one + 1` is executed; and then the scope ends and both Cat and `one` vanish in a puff of smoke. And throughout their brief lives, Cat and `one` were completely invisible to the rest of the program. (Do you see why?)

Object Members

Inside the three object types (class, struct, and enum), things declared at the top level have special names, mostly for historical reasons. Let's use the Manny class as an example:

```
class Manny {

    let name = "manny"

    func sayName() {

        print(name)

    }

}
```

In that code:

- `name` is a variable declared at the top level of an object declaration, so it is called a *property* of that object.
- `sayName` is a function declared at the top level of an object declaration, so it is called a *method* of that object.

Things declared at the top level of an object declaration — properties, methods, and any objects declared at that level — are collectively the *members* of that object. Members have a special significance, because they define the *messages* you are allowed to send to that object!

Namespaces

A *namespace* is a named region of a program. The names of things inside a namespace cannot be reached by things outside it without somehow first passing through the barrier of *saying* that region's name. This is a good thing because it allows the same name to be used in different places without a conflict. Clearly, namespaces and scopes are closely related notions.

Namespaces help to explain the significance of declaring an object at the top level of an object, like this:

```
class Manny {
    class Klass {}
}
```

This way of declaring Klass makes Klass a *nested type*. It effectively “hides” Klass inside Manny. Manny is a namespace! Code *inside* Manny can see (and say) Klass directly. But code outside Manny can’t do that. It has to specify the namespace *explicitly* in order to pass through the barrier that the namespace represents. To do so, it must say Manny’s name first, followed by a dot, followed by the term Klass. In short, it has to say `Manny.Klass`.

The namespace does not, of itself, provide secrecy or privacy; it’s a convenience. Thus, in Example 1-1, I gave Manny a Klass class, and I also gave Moe a Klass class. But they don’t conflict, because they are in different namespaces, and I can differentiate them, if necessary, as `Manny.Klass` and `Moe.Klass`.

It will not have escaped your attention that the syntax for diving explicitly into a namespace is the message-sending dot-notation syntax. They are, in fact, the same thing.

In effect, message-sending allows you to see into scopes you can’t see into otherwise. Code inside Moe can’t *automatically* see the Klass declared inside Manny, but it *can* see it by taking one easy extra step, namely by speaking of `Manny.Klass`. It can do *that* because it *can* see Manny (because Manny is declared at a level that code inside Moe can see).

Modules

The top-level namespaces are *modules*. By default, your app is a module and hence a namespace; that namespace’s name is, roughly speaking, the name of the app. For example, if my app is called `MyApp`, then if I declare a class Manny at the top level of a file, that class’s *real* name is `MyApp.Manny`. But I don’t usually need to use that real name, because my code is already inside the same namespace, and can see the name `Manny` directly.

Frameworks are also modules, and hence they are also namespaces. When you import a module, all the top-level declarations of that module become visible to your code, without your having to use the module's namespace explicitly to refer to them.

For example, Cocoa's Foundation framework, where NSString lives, is a module. When you program iOS, you will say `import Foundation` (or, more likely, you'll say `import UIKit`, which itself imports Foundation), thus allowing you to speak of NSString without saying `Foundation.NSString`. But you *could* say `Foundation.NSString`, and if you were so silly as to declare a different NSString in your own module, you would *have* to say `Foundation.NSString`, in order to differentiate them. You can also create your own frameworks, and these, too, will be modules.

Swift itself is defined in a module — the Swift module. Your code *always implicitly imports the Swift module*. You could make this explicit by starting a file with the line `import Swift`; there is no need to do this, but it does no harm either.

That fact is important, because it solves a major mystery: where do things like `print` come from, and why is it possible to use them outside of any message to any object? `print` is in fact a function declared at the top level of the Swift module, and your code can see the Swift module's top-level declarations because it imports Swift. The `print` function thus becomes, as far as your code is concerned, an ordinary top-level function like any other; it is global to your code, and your code can speak of it without specifying its namespace. You *can* specify its namespace — it is perfectly legal to say things like `Swift.print("hello")` — but you probably never will, because there's no name conflict to resolve.

TIP

You can actually *see* the Swift top-level declarations and read and study them, and this can be a useful thing to do. For example, to see the declaration of `print`, Command-Control-click the term `print` in your code. Alternatively, explicitly `import Swift` and Command-Control-click the term `Swift`. Behold, there are the Swift top-level declarations! You won't see any executable Swift *code* here, but you will see the declarations for all the available Swift terms, including top-level functions like `print`, operators like `+`, and built-in types such as Int and String (look for `struct Int`, `struct String`, and so on).

Instances

Object types — class, struct, and enum — have an important feature in common: they can be *instantiated*. In effect, when you declare an object type, you are only defining a *type*. To instantiate a type is to make a thing — an *instance* — of that type.

So, for example, I can declare a Dog class, and I can give my class a method:

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

But I don't actually have any Dog objects in my program yet. I have merely described the *type* of thing a Dog *would* be if I had one. To get an actual Dog, I have to *make* one. The process of making an actual Dog object whose type is the Dog class is the process of instantiating Dog. The result is a new object — a Dog *instance*.

In Swift, instances can be created by using the object type's name as a function name and calling the function. This involves using parentheses. When you append parentheses to the name of an object type, you are sending a very special kind of message to that object type: Instantiate yourself!

So now I'm going to make a Dog instance:

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

There’s a lot going on in that code! I did two things. I instantiated Dog, thus causing me to end up with a Dog instance. I also put that Dog instance into a shoebox called `fido` — I declared a variable and initialized the variable by assigning my new Dog instance to it. Now `fido` *is a Dog instance*. (Moreover, because I used `let`, `fido` will always be this same Dog instance. I could have used `var` instead, but even then, initializing `fido` as a Dog instance would have meant `fido` could only be some Dog instance after that.)

Now that I have a Dog instance, I can send *instance messages* to it. And what do you suppose they are? They are Dog’s properties and methods! For example:

```
let fido = Dog()

fido.bark()
```

That code is legal. Not only that, it is effective: it actually does cause `"woof"` to appear in the console. I made a Dog and I made it bark! (See Figure 1-1.)

Figure 1-1. Making an instance and calling an instance method

There’s an important lesson here, so let me pause to emphasize it. By default, properties and methods are *instance* properties and methods. You can’t use them as messages to the object type itself; you have to have an *instance* to send those messages to. As things stand, this is illegal and won’t compile:

```
Dog.bark() // compile error
```

It is possible to declare a function `bark` in such a way that saying `Dog.bark()` *is* legal, but that would be a different kind of function — a *class* function or a *static* function — and you would need to say so when you declare it.

The same thing is true of properties. To illustrate, let’s give Dog a `name` property:

```
class Dog {

    var name = ""

}
```

That allows me to set a Dog’s `name`, but it needs to be an *instance* of Dog:

```
let fido = Dog()

fido.name = "Fido"
```

It is possible to declare a property `name` in such a way that saying `Dog.name` is legal, but that would be a different kind of property — a *class* property or a *static* property — and you would need to say so when you declare it.

Why Instances?

Even if there were no such thing as an instance, an object type is itself an object. We know this because it is possible to send a message to an object type (the phrase `Manny.Klass` is a case in point). Why, then, do instances exist at all?

The answer has mostly to do with the nature of instance properties. The value of an instance property is defined with respect to *a particular instance*. This is where instances get their real usefulness and power.

Consider again our Dog class. I’ll give it a `name` property and a `bark` method; remember, these are an instance property and an instance method:

```
class Dog {

    var name = ""
    func bark() {

        print("woof")

    }

}
```

A Dog instance comes into existence with a blank `name` (an empty string). But its `name` property is a `var`, so once we have any Dog instance, we can assign to its `name` a new String value:

```
let dog1 = Dog()

dog1.name = "Fido"
```

We can also ask for a Dog instance’s `name`:

```
let dog1 = Dog()

dog1.name = "Fido"

print(dog1.name) // "Fido"
```

The important thing is that we can make more than one Dog instance, and that two different Dog instances can have two different `name` property values (Figure 1-2):

```
let dog1 = Dog()

dog1.name = "Fido"

let dog2 = Dog()

dog2.name = "Rover"

print(dog1.name) // "Fido"

print(dog2.name) // "Rover"
```

Figure 1-2. Two dogs with different property values

Note that a Dog instance’s `name` property has nothing to do with the name of the variable to which a Dog instance is assigned. The variable is just a shoebox. You can pass an instance from one shoebox to another. But the instance itself maintains its own internal integrity:

```
let dog1 = Dog()

dog1.name = "Fido"

var dog2 = Dog()

dog2.name = "Rover"

print(dog1.name) // "Fido"

print(dog2.name) // "Rover"

dog2 = dog1

print(dog2.name) // "Fido"
```

That code didn’t change Rover’s `name`; it changed which dog was inside the `dog2` shoebox, replacing Rover with Fido.

The full power of object-based programming has now emerged. There is a Dog object type which defines *what it is to be a Dog*. Our declaration of Dog says that a Dog instance — *any* Dog instance, *every* Dog instance — has a `name` property and a `bark` method. But *each* Dog instance can have its own `name` property *value*. They are *different* instances and maintain their own internal *state*. So multiple instances of the same object type *behave* alike — both Fido and Rover can bark, and will do so when they are sent the `bark` message — but they are different instances and can have different property values:
Fido’s `name` is `"Fido"` while Rover’s `name` is `"Rover"`.

So an instance is a reflection of the instance methods of its type, but that isn’t *all* it is; it’s also a collection of instance properties. The object type is responsible for *what* properties the instance has, but not necessarily for the *values* of those properties. The values can change as the program runs, and apply only to a particular instance. An instance is a cluster of particular property values.

An instance is responsible not only for the values but also for the *lifetimes* of its properties. Suppose we bring a Dog instance into existence and assign to its `name` property the value `"Fido"`. Then this Dog instance is keeping the string `"Fido"` alive just so long as we do not replace the value of its `name` with some other value — and just so long as this instance lives.

In short, an instance is both code and data. The code it gets from its type and in a sense is shared with all other instances of that type, but the data belong to it alone. The data can persist as long as the instance persists. The instance has, at every moment, a state — the complete collection of its own personal property values. An instance is a device for *maintaining state*. It's a box for storage of data.

The Keyword self

An instance is an object, and an object is the recipient of messages. Thus, an instance needs a way of sending a message to itself. This is made possible by the keyword `self`. This word can be used wherever an instance of the appropriate type is expected.

For example, let's say I want to keep the thing that a Dog says when it barks, such as `"woof"`, in a property. Then in my implementation of `bark` I need to refer to that property. I can do it like this:

```
class Dog {

    var name = ""

    var whatADogSays = "woof"

    func bark() {

        print(self.whatADogSays)
    }

}
```

Similarly, let’s say I want to write an instance method `speak` which is merely a synonym for `bark`. My `speak` implementation can consist of simply calling my own `bark` method. I can do it like this:

```
class Dog {

    var name = ""

    var whatADogSays = "woof"

    func bark() {

        print(self.whatADogSays)

    }

    func speak() {

        self.bark()

    }

}
```

Observe that the term `self` in that example appears only in instance methods. When an instance’s code says `self`, it is referring to *this* instance. If the expression `self.name` appears in a Dog instance method’s code, it means the `name` of *this* Dog instance, the one whose code is running at that moment.

It turns out that every use of the word `self` I’ve just illustrated is completely optional. You can omit it and all the same things will happen:

```
class Dog {
        var name = ""
        var whatADogSays = "woof"
        func bark() {
            print(whatADogSays)
        }
        func speak() {
            bark()
        }
    }
```

The reason is that if you omit the message recipient and the message you're sending can be sent to `self`, the compiler supplies `self` as the message's recipient under the hood. However, I *never* do that (except by mistake). As a matter of style, I like to be explicit in my use of `self`. I find code that omits `self` harder to read and understand. And there are situations where you *must* say `self`, so I prefer to use it whenever I'm allowed to.

Privacy

Earlier, I said that a namespace is not, of itself, an insuperable barrier to accessing the names inside it. But such a barrier is sometimes desirable. For example, not all data stored by an instance is intended for alteration by, or even visibility to, another instance. And not every instance method is intended to be called by other instances. Any decent object-based programming language needs a way to endow its object members with *privacy* — a way of making it harder for other objects to see those members if they are not supposed to be seen.

Consider, for example:

```
class Dog {

    var name = ""

    var whatADogSays = "woof"

    func bark() {

        print(self.whatADogSays)

    }

    func speak() {

        print(self.whatADogSays)

    }

}
```

Here, other objects can come along and change my property `whatADogSays`. Since that property is used by both `bark` and `speak`, we could easily end up with a Dog that, when told to `bark`, says `"meow"`. That seems somehow undesirable:

```
let dog1 = Dog()

dog1.whatADogSays = "meow"

dog1.bark() // meow
```

You might reply: Well, silly, why did you declare `whatADogSays` with `var`? Declare it with `let` instead. Make it a constant! Now no one can change it:

```
class Dog {

    var name = ""

    let whatADogSays = "woof"
    func bark() {

        print(self.whatADogSays)

    }

    func speak() {

        print(self.whatADogSays)

    }

}
```

That is a good answer, but it is not quite good enough. There are two problems. Suppose I want a Dog instance *itself* to be able to change *its own* `whatADogSays` — by assigning to `self.whatADogSays`. Then `whatADogSays` *has* to be a `var`; otherwise, even the instance itself can't change it. Also, suppose I don't want any other object to *know* what this Dog says, except by calling `bark` or `speak`. Even when declared with `let`, other objects can still *read* the value of `whatADogSays`. Maybe I don't like that.

To solve this problem, Swift provides the `private` keyword. I'll talk later about all the ramifications of this keyword, but for now it's enough to know that it solves the problem:

```
class Dog {

    var name = ""

    private var whatADogSays = "woof"

    func bark() {

        print(self.whatADogSays)

    }

    func speak() {
        print(self.whatADogSays)
    }
}
```

Now `name` is a public property, but `whatADogSays` is a private property: it can't be seen by other types of object. A Dog instance can speak of `self.whatADogSays`, but a Cat instance with a reference to a Dog instance as `fido` cannot say `fido.whatADogSays`. The important lesson here is that object members are public by default, and if you want privacy, you have to ask for it.

To sum up: A class declaration defines a namespace. This namespace requires that other objects use an extra level of dot-notation to refer to what's inside the namespace, but other objects *can* still refer to what's inside the namespace; the namespace does not, in and of itself, close any doors of visibility. The `private` keyword lets you close those doors.

RESERVED WORDS

Certain terms, like `class` and `func` and `var` and `let` and `if` and `private` and `import`, are *reserved* in Swift; they are part of the language. That means you can't use them as *identifiers* — as the name of a class, a function, or a variable, for example. If you try to do so, you'll get a compile error.

To force a reserved word to be an identifier, surround it by backticks (`` ` ``). This (extraordinarily confusing) code is legal:

```
class `func` {
    func `if`() {
        let `class` = 1
    }
}
```

Design

What object types will your program need, what methods and properties should they have, when and how will they be instantiated, and what should you do with those instances when you have them? Those aren't easy decisions, and there are no clear-cut answers. Object-based programming is an art.

In real life, when you're programming iOS, many object types you'll be working with will not be yours but Apple's. Swift itself comes with a few useful object types, such as String and Int; you'll also `import UIKit`, which includes a *huge* number of object types, all of which spring to life in your program. You didn't create any of those object types, so their design is not your problem; instead, you must learn to use them. Apple's object types are aimed at enabling the *general* functionality that any app might need. At the same time, your app will probably have *specific* functionality, unique to its purpose, and you will have to design object types to serve that purpose.

Object-based program design must be founded upon a secure understanding of the nature of objects. You want to design object types that encapsulate the right sort of functionality (methods) accompanied by the right set of data (properties). Then, when you instantiate those object types, you want to make sure that your instances have the right lifetimes, sufficient exposure to one another, and an appropriate ability to communicate with one another.

Object Types and APIs

Your program files will have very few, if any, top-level functions and variables. Methods and properties of object types — in particular, instance methods and instance properties — will be where most of the action is. Object types give each actual instance its specialized abilities. They also help to organize your program's code meaningfully and maintainably.

We may summarize the nature of objects in two phrases: encapsulation of functionality, and maintenance of state. (I first used this summary many years ago in my book *REALbasic: The Definitive Guide*.)

Encapsulation of functionality

Each object does its own job, and presents to the rest of the world — to other objects, and indeed in a sense to the programmer — an opaque wall whose only entrances are the methods to which it promises to respond and the actions it promises to perform when the corresponding messages are sent to it. The details of how, behind the scenes, it actually implements those actions are secreted within itself; no other object needs to know them.

Maintenance of state

Each individual instance is a bundle of data that it maintains. Often that data is private, so it's encapsulated as well; no other object knows what that data is or in what form it is kept. The only way to discover from outside what private data an object is maintaining is if there's a public method or property that reveals it.

As an example, imagine an object whose job is to implement a stack — it might be an instance of a Stack class. A *stack* is a data structure that maintains a set of data in LIFO order (last in, first out). It responds to just two messages: `push` and `pop`. Push means to add a given piece of data to the set. Pop means to remove from the set the piece of data that was most recently pushed and hand it out. It's like a stack of plates: plates are placed onto the top of the stack or removed from the top of the stack one by one, so the first plate to go onto the stack can't be retrieved until all other subsequently added plates have been removed (Figure 1-3).

Figure 1-3. A stack

The stack object illustrates encapsulation of functionality because the outside world knows nothing of how the stack is actually implemented. It might be an array, it might be a linked list, it might be any of a number of other implementations. But a client object — an object that actually sends a `push` or `pop` message to the stack object — knows nothing of this and cares less, provided the stack object adheres to its contract of behaving like a stack. This is also good for the programmer, who can, as the program develops, safely substitute one implementation for another without harming the vast machinery of the program as a whole.

The stack object illustrates maintenance of state because it isn't just the gateway to the stack data — it *is* the stack data. Other objects can get access to that data, but only by virtue of having access to the stack object itself, and only in the manner that the stack object permits. The stack data is effectively inside the stack object; no one else can see it. All that another object can do is push or pop.

The sum total of messages that each object type is eligible to be sent by other objects — its *API* (application programming interface) — is like a list or menu of things you can ask this type of object to do. Your object types divide up your code; their APIs form the basis of communication between those divisions. The same is true of objects that you didn't design. Apple's Cocoa documentation consists largely of lists of object APIs. For example, to know what messages you can send to an NSString instance, you'd start by studying the NSString class documentation. That page is really just a big list of properties and methods, so it tells you what an NSString object can do — and thus constitutes the bulk of what you need to know in order to use NSStrings in your program.

Instance Creation, Scope, and Lifetime

The important moment-to-moment entities in a Swift program are mostly instances. Object types define what *kinds* of instances there can be and how each kind of instance behaves. But the actual instances of those types are the state-carrying individual "things" that the program is all about, and instance methods and properties are messages that can be sent to instances. So there need to *be* instances in order for the program to *do* anything.

By default, however, there are *no* instances! Looking back at Example 1-1, we defined some object types, but we made no instances of them. If we were to run this program, our object types would exist from the get-go, but that's all that would exist. We've created a world of pure potentiality — some types of object that *might* exist. In that world, nothing would actually *happen*.

Instances do not come into being by magic. You have to instantiate a type in order to obtain an instance. Much of the action of your program, therefore, will consist of instantiating types. And of course you will want those instances to persist, so you will also assign each newly created instance to a variable as a shoebox to hold it, name it, and give it a lifetime. The instance will *persist* according to the lifetime of the variable that refers to it. And the instance will be *visible* to other instances according to the scope of the variable that refers to it.

Much of the art of object-based programming involves giving instances a sufficient lifetime and making them visible to one another. You will often put an instance into a particular shoebox — assigning it to a particular variable, declared at a certain scope — exactly so that, thanks to the rules of variable lifetime and scope, this instance will *persist* long enough to keep being useful to your program while it will still be needed, and so that other code can *get a reference* to this instance and talk to it later.

Planning how you’re going to create instances, and working out the lifetimes and communication between those instances, may sound daunting. Fortunately, in real life, when you’re programming iOS, the Cocoa framework itself will provide an initial scaffolding for you. Before you write a single line of code, the framework ensures that your app, as it launches, is given some instances that will persist for the lifetime of the app, providing the basis of your app’s visible interface and giving you an initial place to put your own instances and give them sufficiently long lifetimes.

Summary and Conclusion

As we imagine constructing an object-based program for performing a particular task, we bear in mind the nature of objects. There are types and instances. A type is a set of methods describing what all instances of that type can do (encapsulation of functionality). Instances of the same type differ only in the value of their properties (maintenance of state). We plan accordingly. Objects are an organizational tool, a set of boxes for encapsulating the code that accomplishes a particular task. They are also a conceptual tool. The programmer, being forced to think in terms of discrete objects, must divide the goals and behaviors of the program into discrete tasks, each task being assigned to an appropriate object.

At the same time, no object is an island. Objects can cooperate with one another, namely by communicating with one another — that is, by sending messages to one another. The ways in which appropriate lines of communication can be arranged are innumerable. Coming up with an appropriate arrangement — an *architecture* — for the cooperative and orderly relationship between objects is one of the most challenging aspects of object-based programming. In iOS programming, you get a boost from the Cocoa framework, which provides an initial set of object types and a practical basic architectural scaffolding.

Using object-based programming effectively to make a program do what you want it to do while keeping it clear and maintainable is itself an art; your abilities will improve with experience. Eventually, you may want to do some further reading on effective planning and construction of the architecture of an object-based program. I recommend in particular two classic, favorite books. *Refactoring*, by Martin Fowler (Addison-Wesley, 1999), describes why you might need to rearrange what methods belong to what classes (and how to conquer your fear of doing so). *Design Patterns*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (also known as "the Gang of Four"), is the bible on architecting object-based programs, listing all the ways you can arrange objects with the right powers and the right knowledge of one another (Addison-Wesley, 1994).