CHAPTER 2
Functions

Nothing is so characteristic of Swift syntax as the way you declare and call functions. Probably nothing is so important, either! As I said in Chapter 1, all your executable code is going to be in functions; they are where the action is.

Function Parameters and Return Value

Remember those imaginary machines for processing miscellaneous stuff that you drew in your math textbook in elementary school? You know the ones I mean: with a funnel-like "hopper" at the top, and then a bunch of gears and cranks, and then a tube at the bottom where something is produced. A function works like that: you feed some stuff in, the stuff is processed in accordance with what this particular machine does, and something is produced. The stuff that goes in is the input; what comes out is the output. More technically, a function that expects input has *parameters*; a function that produces output has a *result*.

Here's the declaration for a silly but valid function that expects two Int values, adds them together, and produces that sum:

```
func sum (_ x:Int, _ y:Int) -> Int {
    let result = x + y
    return result
}
```

The syntax here is very strict and well-defined, and you can't use Swift unless you understand it perfectly. Let's pause to appreciate it in full detail. I'll break the first line into pieces so that I can call them out individually:

```
func sum                        ❶
    (_ x:Int, _ y:Int)          ❷❸
    -> Int {                    ❹❺
        let result = x + y      ❻
        return result           ❼
}
```

❶ The declaration starts with the keyword `func`, followed by the *name* of this function; here, it's `sum`. This is the name that must be used in order to *call* the function — that is, in order to run the code that the function contains.

❷ The name of the function is followed by its *parameter list*. It consists, minimally, of parentheses. If this function takes parameters (input), they are listed inside the parentheses, separated by a comma. Each parameter has a strict format: the *name* of the parameter, a colon, and the *type* of the parameter.

❸ This particular function declaration also has an underscore (`_`) and a space before each parameter name in the parameter list. I'm not going to explain that underscore yet, but I need it for the example, so just trust me for now.

❹ If the function is to return a value, then after the parentheses is an arrow operator (`->`) followed by the *type* of value that this function will return.

❺ Then we have curly braces enclosing the *body* of the function — its actual code.

❻ Within the curly braces, in the function body, the variables defined as the parameter names have sprung to life, with the types specified in the parameter list.

❼ If the function is to return a value, it must do so with the keyword `return` followed by that value. And, not surprisingly, the type of that value must match the type declared earlier for the return value (after the arrow operator).

Here are some further points to note about the parameters and return type of our function:

Parameters

Our `sum` function expects two parameters — an Int, to which it gives the name `x`, and another Int, to which it gives the name `y`. The function body code won't run unless code elsewhere calls this function and actually passes values of the specified types for its parameters. If I try to call this function *without* providing a value for each of these two parameters, or if either of the values I provide is *not* an Int, the compiler will stop me with an error.

In the body of the function, therefore, we can confidently use those values, referring to them by those names, certain that they will exist as specified by our parameter list. The parameter names `x` and `y`, indeed, are defined just so that the

parameter values *can* be referred to within the function body. The parameter declaration is thus a kind of variable declaration: we are declaring variables `x` and `y` for use inside this function. With regard to their scope, these variables are local (*internal*) to the function; *only* the function body can see them, and they are different from any other `x` and `y` that may be used in other functions or at a higher level of scope.

Return type
: The last statement of our `sum` function's body returns the value of a variable called `result`; this variable was created by adding two Int values together, so it is an Int, which is what this function is supposed to produce. If I try to return a String (`return "howdy"`), or if I omit the return statement altogether, the compiler will stop me with an error.

  The keyword `return` actually does *two* things. It *returns* the accompanying value, and it also *halts* execution of the function. It is permitted for more lines of code to follow a return statement, but the compiler will warn if this means that those lines can never be executed.

  A function that returns a value must contain a return statement, so if its body consists of just a single statement, it must *be* the return statement. The keyword `return` can be omitted in that situation. This is mostly to facilitate the SwiftUI domain-specific language, and in general I like to say `return` explicitly, but in some situations it does feel more elegant to omit it.

You can view the function declaration before the curly braces as a *contract* about what kinds of values will be used as input and about what kind of output will be produced. According to this contract, the function *expects* a certain number of parameters, each of a certain type, and *yields* a certain type of result. Everything must correspond to this contract. The function body, inside the curly braces, can use the parameters as local variables. The returned value must match the declared return type.

The same contract applies to code elsewhere that *calls* this function. Here's some code that calls our `sum` function:

```
let z = sum(4,5)
```

Focus your attention on the right side of the equal sign — `sum(4,5)`. That's the function call. How is it constructed? It uses the *name* of the function; that name is followed by *parentheses*; and inside those parentheses, separated by a comma, are the *values* to be passed to each of the function's parameters. Technically, these values are called *arguments*. Here, I'm using literal Int values, but I'm perfectly free to use Int variables instead; the only requirement is that I use things that have the correct type:

```
let x = 4
let y = 5
let z = sum(y,x)
```

In that code, I purposely used the names `x` and `y` for the variables whose values are passed as arguments, and I purposely reversed them in the call, to emphasize that these names have *nothing to do* with the names `x` and `y` inside the function parameter list and the function body. Argument names do not magically make their way to the function. Their *values* are all that matter; their values are the arguments.

What about the value returned by the function? That value is magically *substituted* for the function call, at the point where the function call is made. It happens that in the preceding code, the result is 9. So the last line is exactly as if I had said:

```
let z = 9
```

The programmer and the compiler both know what type of thing this function returns, so they also know where it is and isn't legal to call this function. It's fine to call this function as the initialization part of the declaration of the variable `z`, just as it would be to use `9` as the initialization part of that declaration: in both cases, we have an Int, and so `z` ends up being declared as an Int. But it would not be legal to write this:

```
let z = sum(4,5) + "howdy" // compile error
```

Because `sum` returns an Int, that's the same as trying to add an Int to a String — and by default, you can't do that in Swift.

Observe that it is legal to ignore the value returned from a function call:

```
sum(4,5)
```

That code is rather silly in this particular situation, because we have made our `sum` function go to all the trouble of adding 4 and 5 for us and we have then thrown away the answer without capturing or using it. The compiler knows this, and will warn that we are failing to use the result of our function call. Nevertheless, a warning is not an error; that code is legal. There are, in fact, lots of situations where it is perfectly reasonable to ignore the value returned from a function call; in particular, the function may do other things (technically called *side effects*) in addition to returning a value, and the purpose of your call to that function may be those other things.

If you're ignoring a function call result deliberately, you can silence the compiler warning by assigning the function call to `_` (a variable without a name) — for example, `_ = sum(4,5)`. Alternatively, if the function being called is your own, you can prevent the warning by marking the function declaration with `@discardableResult`.

If you can call `sum` wherever you can use an Int, and if the parameters of `sum` have to be Int values, doesn't that mean you can call `sum` inside a call to `sum`? Of course it does! This is perfectly legal (and reasonable):

```
let z = sum(4,sum(5,6))
```

The only arguments against writing code like that are that you might confuse yourself and that it might make things harder to debug later. But technically it's legal and quite normal.

Void Return Type and Parameters

Let's return to our function declaration. With regard to a function's parameters and return type, there are two degenerate cases that allow us to express a function declaration more briefly:

A function without a return type
: No law says that a function *must* return a value. A function may be declared to return *no* value. In that case, there are three ways to write the declaration: you can write it as returning Void; you can write it as returning `()`, an empty pair of parentheses; or you can omit the arrow operator and the return type entirely. These are all legal:

```
func say1(_ s:String) -> Void { print(s) }
func say2(_ s:String) -> () { print(s) }
func say3(_ s:String) { print(s) }
```

If a function returns no value, then its body need not contain a return statement. If it does contain a return statement, it will consist of the word `return` alone, and its purpose will be purely to end execution of the function at that point.

A call to a function that returns no value is made purely for the function's side effects; it has no useful return value that can be made part of a larger expression, so the statement that calls the function will usually consist of the function call and nothing else.

A function without any parameters
: No law says that a function *must* take any parameters. If it doesn't, the parameter list in the function declaration can be completely empty. But you can't omit the parameter list parentheses themselves! They will be present in the function declaration, after the function's name:

```
func greet() -> String { return "howdy" }
```

Just as you cannot omit the parentheses (the parameter list) from a function *declaration*, you cannot omit the parentheses from a function *call*. Those parentheses will be empty if the function takes no parameters, but they must be present:

```
let greeting = greet()
```

Notice the parentheses!

A function can lack both a return value and parameters; these are all ways of expressing the same thing:

```
func greet1() -> Void { print("howdy") }
func greet2() -> () { print("howdy") }
func greet3() { print("howdy") }
```

Function Signature

If we ignore the parameter names in the function declaration, we can completely characterize a function by the *types* of its inputs and its output. To do so, we write the parameter types in parentheses, followed by the arrow operator and the output type, like this:

```
(Int, Int) -> Int
```

That is a legal expression in Swift; it is the *signature* of a function. In this case, it's the signature of a function that takes two Int parameters and returns an Int. In fact, it's the signature of our `sum` function! Of course, there can be other functions that take two Int parameters and return an Int — and that's just the point. This signature characterizes *all* functions that have this number of parameters, of these types, and that return a result of this type. A function's signature is, in effect, *its* type — the type *of the function*. The fact that functions have types will be of great importance later on.

The signature of a function must include both the parameter list (without parameter names) and the return type, even if one or both of those is empty; the signature of a function that takes no parameters and returns no value may be written `() -> Void` or `() -> ()`.

External Parameter Names

A function can *externalize* the names of its parameters. The external parameter names become part of the function's name, and must appear in a call to the function as *labels* to the arguments. There are several reasons why this is a good thing:

- It clarifies the purpose of each argument; an argument label can give a clue as to how that argument contributes to the behavior of the function.
- It distinguishes one function from another; two functions with the same name before the parentheses and the same signature, but with different external parameter names, are two distinct functions.
- It helps Swift to interface with Objective-C and Cocoa, where method parameters nearly always have external names.

It will be useful to have a term for the part of a function's name that precedes the parentheses, so I don't have to keep calling it "the name before the parentheses" to distinguish it from the external parameter names. Let's call it the function's *base name*.

External parameter names are so standard in Swift that there's a rule: by default, *all* parameter names are externalized *automatically*, using the internal name as the external name. If you want a parameter name to be externalized, and if you want the external name to be the same as the internal name, *do nothing* — that will happen all by itself.

If you want to depart from the default behavior, you can do either of the following in your function declaration:

Change the name of an external parameter
: If you want the external name of a parameter to be different from its internal name, precede the internal name with the external name and a space.

Suppress the externalization of a parameter
: To suppress a parameter's external name, precede the internal name with an underscore and a space.

That explains my declaration `func sum (_ x:Int, _ y:Int) -> Int` at the start of this chapter: I was suppressing the externalization of the parameter names, so as not to have to explain argument labels at the outset.

Here's the declaration for a function that concatenates a string with itself a given number of times:

```
func echoString(_ s:String, times:Int) -> String {
    var result = ""
    for _ in 1...times { result += s }
    return result
}
```

That function's first parameter has an internal name only, but its second parameter has an external name, which will be the same as its internal name, namely `times`. And here's how to call it:

```
let s = echoString("hi", times:3)
```

In the call, as you can see, the external name precedes the argument as a label, separated by a colon.

Now let's say that in our `echoString` function we prefer to use `times` purely as an external name for the second parameter, with a different name — say, `n` — as the internal name. And let's strip the `string` off the function's base name and make it the external name of the first parameter. Then the declaration would look like this:

```
func echo(string s:String, times n:Int) -> String {
    var result = ""
    for _ in 1...n { result += s }
    return result
}
```

In the body of that function, there is now no `times` variable available; `times` is purely an external name, for use as a label in the call. The internal name is `n`, and that's the name the code refers to. And here's how to call it:

```
let s = echo(string:"hi", times:3)
```

The existence of external names doesn't mean that the call can use a different parameter order from the declaration. Our `echo(string:times:)` expects a String parameter and an Int parameter, *in that order*. The order can't be different in the call, even though the label might appear to disambiguate which argument goes with which parameter.

Overloading

In Swift, function *overloading* is legal (and common). This means that two functions with exactly the same name, *including* their external parameter names, can coexist as long as they have different signatures. (Two functions with the same base name but *different* external parameter names do *not* constitute a case of overloading; they are simply two different functions with two different names.) These two functions can coexist:

```
func say (_ what:String) {
}
func say (_ what:Int) {
}
```

The reason overloading works is that Swift has strict typing. A String parameter is not an Int parameter. Swift can tell them apart both in the declaration and in a function call. So Swift knows unambiguously that `say("what")` is different from `say(1)`, and it knows *which* `say` function each is calling.

Overloading works for the return type as well. Two functions with the same name and parameter types can have different return types. But the context of the call must disambiguate; that is, it must be clear what return type the caller is expecting.

For example, these two functions can coexist:

```
func say() -> String {
    return "one"
}
func say() -> Int {
    return 1
}
```

But now you can't call `say` like this:

```
let result = say() // compile error
```

The call is ambiguous, and the compiler tells you so. The call must be used in a context where the expected return type is clear. One solution, as I'll describe in Chapter 3, is to state the problematic type explicitly (rather than relying on type inference):

```
let result: String = say()
```

Alternatively, the context itself might disambiguate. Suppose we have another function that is not overloaded, and that expects a String parameter:

```
func giveMeAString(_ s:String) {
    print("thanks!")
}
```

Then `giveMeAString(say())` is legal, because only a String can go in this spot, so we must be calling the `say` that returns a String. Similarly:

```
let result = say() + "two"
```

Only a String can be "added" to a String, so this must be the `say` that returns a String.

You can also disambiguate explicitly between overloads in a method call using the name of the method, the keyword `as`, and the signature of the desired method. The syntax is a little odd, because the entire expression must be enclosed in parentheses, followed immediately by the parentheses signifying that this is a method call:

```
let result = (say as () -> String)()
```

Default Parameter Values

A parameter can have a default value. This means that the caller can omit the parameter entirely, supplying no argument for it; the value will then be the default.

To specify a default value in a function declaration, append `=` and the default value after the parameter type:

```
class Dog {
    func say(_ s:String, times:Int = 1) {
        for _ in 1...times {
            print(s)
        }
    }
}
```

In effect, there are now *two* functions, one with a single unlabeled parameter, the other with an additional `times:` parameter. If you just want to say something once, you can call the `say` that takes a single unlabeled argument:

```
let d = Dog()
d.say("woof") // same as d.say("woof", times:1)
```

If you want repetition, call the `say` that takes a `times:` parameter:

```
let d = Dog()
d.say("woof", times:3)
```

Variadic Parameters

A parameter can be *variadic*. This means that the caller can supply as many argument values of this parameter's type as desired, separated by a comma; the function body will receive these values as an array.

To indicate in a function declaration that a parameter is variadic, follow it with three dots, like this:

```
func sayStrings(_ array:String ...) {
    for s in array { print(s) }
}
```

And here's how to call it:

```
sayStrings("hey", "ho", "nonny nonny no")
```

Another parameter may follow a variadic parameter, but it must have an external label to show where the variadic arguments end in the call. The global `print` function works like that. It takes a variadic first parameter, so that you can output multiple values with a single command; the remaining parameters have default values and can be omitted:

```
print("Manny", 3, true) // Manny 3 true
```

The `print` function's remaining parameters dictate further details of the output. The default `separator:` (for when you provide multiple values) is a space, and the default `terminator:` is a newline; you can change either or both:

```
print("Manny", "Moe", separator:", ", terminator:", ")
print("Jack")
// output is "Manny, Moe, Jack" on one line
```

In Swift 5.3 and before, a function can declare a maximum of one variadic parameter. Starting in Swift 5.4, that restriction has been lifted. Even two variadic parameters in a row are now legal, though you still need an external label on the second one:

```
func sayStrings(_ array:String ..., and array2:String ...) {
    for s in array { print(s) }
    for s in array2 { print(s) }
}
```

A second variadic parameter that isn't adjacent to the first variadic parameter doesn't require an external label:

Initializers Are Functions

In Chapter 1, I declared a Dog class and instantiated it, creating a Dog instance, by saying `Dog()`. When you do this, you are actually calling a function — a special function called an *initializer*. This particular initializer function takes no parameters; hence the parentheses are empty. But an initializer is a function like any other, so it can have parameters, external parameter names, default parameter values, overloading, and so forth. And the syntax for calling an initializer (and thus for creating an instance) is the same as the syntax for any other function call. What's special is only that the parentheses follow the name of the type we're instantiating.

I'll explain how initializers are declared and implemented later (Chapter 4). Swift built-in types tend to have multiple initializers. For example, here I'll initialize two String objects:

```
let s1 = String(42)
let s2 = String(repeating: "ho", count: 2)
```

After that, `s1` is the string `"42"` and `s2` is the string `"hoho"`. And there are many other String initializers, each providing a different way to make a String.

```
func sayStrings(_ array:String ..., other:String, _ array2:String ...) {
    for s in array { print(s) }
    print(other)
    for s in array2 { print(s) }
}
```

Arguably, that function declaration is silly, as it results in a confusing call:

```
sayStrings("Manny", "Moe", "Jack", other: "Matt", "Groucho", "Harpo", "Chico")
```

In that call, `"Matt"` is a parameter on its own, while `"Groucho"` and the rest are the arguments to an additional (variadic) parameter. But there's no indication of that, so a human being might think that `other:` is a variadic. Nevertheless, the compiler isn't confused, so it's legal!

Unfortunately, there's a hole in the Swift language: there's no way to convert an array into a comma-separated list of arguments (comparable to splatting in Ruby). If what you're starting with is an array of some type, you can't use it where a variadic of that type is expected.

Ignored Parameters

A parameter whose local name is an underscore is ignored. The caller must supply an argument, but it has no name within the function body and cannot be referred to there. For example:

```
func say(_ s:String, times:Int, loudly _:Bool) {
```

No `loudly` parameter makes its way into the function body, but the caller must still provide it:

```
say("hi", times:3, loudly:true)
```

What's the purpose of this feature? It isn't to satisfy the compiler, because the compiler doesn't complain if a parameter is never referred to in the function body. I use it primarily as a kind of note to myself, a way of saying, "Yes, I know there is a parameter here, and I am deliberately not using it for anything."

Modifiable Parameters

In the body of a function, a parameter is essentially a local variable. By default, it's a variable implicitly declared with `let`. You can't assign to it:

```
func say(_ s:String, times:Int, loudly:Bool) {
    loudly = true // compile error
}
```

If your code needs to assign to a parameter name within the body of a function, declare a `var` local variable inside the function body and assign the parameter value to it; your local variable can even have the same name as the parameter:

```
func say(_ s:String, times:Int, loudly:Bool) {
    var loudly = loudly
    loudly = true // no problem
}
```

In that code, `loudly` is a local variable; assigning to it doesn't change the value of any variable outside the function body. However, it is also possible to configure a parameter in such a way that assigning to it *does* modify the value of a variable outside the function body! One typical use case is that you want your function to return more than one result. Here I'll write a rather advanced function that removes all occurrences of a given character from a given string and returns the number of occurrences that were removed:

```
func removeCharacter(_ c:Character, from s:String) -> Int {
    var s = s
    var howMany = 0
    while let ix = s.firstIndex(of:c) {
        s.remove(at:ix)
        howMany += 1
    }
    return howMany
}
```

And you call it like this:

```
let s = "hello"
let result = removeCharacter("l", from:s) // 2
```

That's nice, but the *original* string, `s`, is still `"hello"`! In the function body, we removed all occurrences of the character from the *local* copy of the String parameter.

If we want a function to alter the *original* value of an argument passed to it, we must do the following:

- The type of the parameter we intend to modify must be declared `inout`.
- When we call the function, the variable holding the value to be modified must be declared with `var`, not `let`.
- Instead of passing the variable as an argument, we must pass its *address*. This is done by preceding its name with an ampersand (`&`).

Our `removeCharacter(_:from:)` now looks like this:

```
func removeCharacter(_ c:Character, from s: inout String) -> Int {
    var howMany = 0
    while let ix = s.firstIndex(of:c) {
        s.remove(at:ix)
        howMany += 1
    }
    return howMany
}
```

And our call to `removeCharacter(_:from:)` now looks like this:

```
var s = "hello"
let result = removeCharacter("l", from:&s)
```

After the call, `result` is 2 and `s` is `"heo"`. Notice the ampersand before the name `s` when we pass it as the `from:` argument. It is required; if you omit it, the compiler will stop you. I like this requirement, because it forces us to acknowledge explicitly to the compiler, and to ourselves, that we're about to do something potentially dangerous: we're letting this function, as a side effect, modify a value outside of itself.

When a function with an `inout` parameter is called, the variable whose address was passed as argument to that parameter is *always* set, even if the function makes no changes to that parameter.

Calling Objective-C with Modifiable Parameters

You may encounter variations on that pattern when you're using Cocoa. The Cocoa APIs are written in C and Objective-C, so instead of the Swift term `inout`, you'll probably see some mysterious type such as UnsafeMutablePointer. From your point of view as the caller, however, it's the same thing: you'll prepare a `var` variable and pass its address.

For instance, consider the problem of learning a UIColor's RGBA components. There are four such components: the color's red, green, blue, and alpha values. A function that, given a UIColor, returned the components of that color, would need to return four values at once — and that is something that Objective-C cannot do. So a different strategy is used. The UIColor method `getRed(_:green:blue:alpha:)` returns only a Bool reporting whether the component extraction succeeded. Instead of returning the actual components, it says: "You hand me four CGFloats *as arguments*, and I will *modify* them for you so that they are the results of this operation." Here's roughly how the declaration for `getRed(_:green:blue:alpha:)` appears in Swift:

```
func getRed(_ red: UnsafeMutablePointer<CGFloat>,
    green: UnsafeMutablePointer<CGFloat>,
    blue: UnsafeMutablePointer<CGFloat>,
    alpha: UnsafeMutablePointer<CGFloat>) -> Bool
```

How would you call this function? The parameters are each an UnsafeMutablePointer to a CGFloat. You'll create four `var` CGFloat variables beforehand, giving them each some value even though that value will be replaced when you call `getRed(_:green:blue:alpha:)`. The arguments you'll pass will be the *addresses* of those variables. Those variables are where the component values will be after the call; and you'll probably be so sure that the component extraction will succeed, that you won't even bother to capture the call's actual result:

```
let c = UIColor.purple
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)
// now r, g, b, a are 0.5, 0.0, 0.5, 1.0
```

Called by Objective-C with Modifiable Parameters

Sometimes, Cocoa will call *your* function with an UnsafeMutablePointer parameter, and *you* will want to change its value. To do this, you cannot assign directly to it, as we did earlier with the `inout` parameter `s` that we declared in our implementation of `remove(from:character:)`. You're talking to Objective-C, not to Swift, and this is an UnsafeMutablePointer, not an `inout` parameter. The technique here is to assign to the UnsafeMutablePointer's `pointee` property. Here (without further explanation) is an example from my own code:

```
func popoverPresentationController(
    _ popoverPresentationController: UIPopoverPresentationController,
    willRepositionPopoverTo rect: UnsafeMutablePointer<CGRect>,
    in view: AutoreleasingUnsafeMutablePointer<UIView>) {
        view.pointee = self.button2
        rect.pointee = self.button2.bounds
}
```

Reference Type Modifiable Parameters

There is one very common situation where your function can modify a parameter *without* declaring it as `inout` — namely, when the parameter is an *instance of a class*. This is a special feature of classes, as opposed to the other two object type flavors, enum and struct. String isn't a class; it's a struct. That's why we had to use `inout` in order to modify a String parameter. So I'll illustrate by declaring a Dog class with a `name` property:

```
class Dog {
    var name = ""
}
```

Here's a function that takes a Dog instance parameter and a String, and sets that Dog instance's `name` to that String. Notice that no `inout` is involved:

```
func changeName(of d:Dog, to newName:String) {
    d.name = newName
}
```

Here's how to call it. We pass a Dog instance *directly*:

```
let d = Dog()
d.name = "Fido"
print(d.name) // "Fido"
changeName(of:d, to:"Rover")
print(d.name) // "Rover"
```

We were able to change a property of our Dog instance `d`, even though it wasn't passed as an `inout` parameter, and even though it was declared originally with `let`, not `var`. This appears to be an exception to the rules about modifying parameters — but it isn't. It's a feature of class instances, namely that they are themselves mutable. In `changeName(of:to:)`, we didn't actually attempt to assign *a different Dog instance* to the parameter. To do that, the Dog parameter *would* need to be declared `inout` (and `d` would have to be declared with `var` and we would have to pass its address as argument).

Technically, we say that classes are *reference types*, whereas the other object type flavors are *value types*. When you pass an instance of a struct as an argument to a function, you effectively wind up with a *separate copy* of the struct instance. But when you pass an instance of a class as an argument to a function, you pass a reference to the

class instance *itself*. I'll discuss this topic in more detail later ("Value Types and Reference Types" on page 157).

Function in Function

A function can be declared anywhere, including inside the body of a function. A function declared in the body of a function (also called a *local function*) is available to be called by later code within the same scope, but is completely invisible elsewhere.

This feature is an elegant architecture for functions whose sole purpose is to assist another function. If only function A ever needs to call function B, function B might as well be packaged inside function A.

Here's a typical example from one of my apps (I've omitted everything except the structure):

```
func checkPair(_ p1:Piece, and p2:Piece) -> Path? {
    // ...
    func addPathIfValid(_ midpt1:Point, _ midpt2:Point) {
        // ...
    }
    for y in -1..._yct {
        addPathIfValid((pt1.x,y),(pt2.x,y))
    }
    for x in -1..._xct {
        addPathIfValid((x,pt1.y),(x,pt2.y))
    }
    // ...
}
```

What I'm doing in the first for loop (`for y`) and what I'm doing in the second for loop (`for x`) are the same — but with a different set of starting values. We could write out the functionality in full inside each for loop, but that would be an unnecessary and confusing repetition. (Such a repetition would violate the principle often referred to as *DRY*, for "Don't Repeat Yourself.") To prevent that repetition, we could refactor the repeated code into an instance method to be called by both for loops, but that exposes this functionality more broadly than we need, as it is called *only* by these two for loops inside `checkPair`. A local function is the perfect compromise.

Recursion

A function can call itself. This is called *recursion*. Recursion seems a little scary, rather like jumping off a cliff, because of the danger of creating an infinite loop; but if you write the function correctly, you will always have a "stopper" condition that handles the degenerate case and prevents the loop from being infinite:

```
func countDownFrom(_ ix:Int) {
    print(ix)
    if ix > 0 { // stopper
        countDownFrom(ix-1) // recurse!
    }
}
countDownFrom(5) // 5, 4, 3, 2, 1, 0
```

Function as Value

If you've never used a programming language where functions are first-class citizens, perhaps you'd better sit down now, because what I'm about to tell you might make you feel a little faint: In Swift, a function *is* a first-class citizen. This means that a function can be used wherever a value can be used. A function can be assigned to a variable; a function can be passed as an argument in a function call; a function can be returned as the result of a function.

Swift has strict typing. You can assign a value to a variable or pass a value into or out of a function only if it is the right *type* of value. In order for a function to be used as a value, the function needs to *have* a type. And indeed it does: a function's *signature* is its type.

The chief purpose of using a function as a value is so that this function can later be called without a definite knowledge of *what* function it is. Here's the world's simplest (and silliest) example, just to show the syntax and structure:

```
func doThis(_ f:() -> ()) {
    f()
}
```

That is a function `doThis` that takes one parameter (and returns no value). The parameter, `f`, is itself a function! How do we know? Well, look at its type (after the colon): it is `() -> ()`. That's a function signature; in particular, it is the signature of a function that takes no parameters and returns no value.

The function `doThis`, then, expects as its parameter a function, which it names `f`. Then, within its body, `doThis` *calls* the function `f` that it received as its parameter, by saying `f()`. So `doThis` is merely a function that trivially calls another function. But it does this without knowing in advance *what* function it is going to call. That's the power of functions being first-class citizens.

Having declared the function `doThis`, how would you call it? You'd need to pass it a function as argument. Here's one way to do that:

```
func doThis(_ f:() -> ()) {
    f()
}
func whatToDo() { ❶
    print("I did it")
}
doThis(whatToDo) ❷
```

❶ First, we declare a function (`whatToDo`) *of the proper type* — a function that takes no parameters and returns no value.

❷ Then, we call `doThis`, passing as argument a *function reference* — in effect, the bare name of the function. Notice that we are not *calling* `whatToDo` here; we are *passing* it.

Sure enough, this works: we pass `whatToDo` as argument to `doThis`; `doThis` calls the function that it receives as its parameter; and the string `"I did it"` appears in the console.

Obviously, that example, while demonstrating a remarkable ability of the Swift language, is far from compelling, because the outcome in practice is no different from what would have happened if we had simply called `whatToDo` directly. But in real life, encapsulating function-calling in a function can reduce repetition and opportunity for error. Moreover, a function may call its parameter function in some special way; it might call it after doing other things, or at some later time.

Here's a case from my own code. A common thing to do in Cocoa is to draw an image, directly, in code. One way of doing this involves four steps:

```
let size = CGSize(width:45, height:20)
UIGraphicsBeginImageContextWithOptions(size, false, 0) ❶
let p = UIBezierPath(
    roundedRect: CGRect(x:0, y:0, width:45, height:20), cornerRadius: 8)
p.stroke() ❷
let result = UIGraphicsGetImageFromCurrentImageContext()! ❸
UIGraphicsEndImageContext() ❹
```

❶ Open an image context.

❷ Draw into the context.

❸ Extract the image.

❹ Close the image context.

That works — in this case, it generates an image of a rounded rectangle — but it's ugly. The sole purpose of all that code is to obtain `result`, the image; but that purpose is buried in all the other code. At the same time, everything except for the two lines involving the UIBezierPath `p` (step 2) is boilerplate; every time I do this in any

app, step 1, step 3, and step 4 are exactly the same. Moreover, I live in mortal fear of forgetting a step; if I were to omit step 4 by mistake, the universe would explode.

Since the only thing that's different every time I draw is step 2, step 2 is the only part I should have to write out! The entire problem is solved by writing a utility function expressing the boilerplate:

```
func imageOfSize(_ size:CGSize, _ whatToDraw:() -> ()) -> UIImage {
    UIGraphicsBeginImageContextWithOptions(size, false, 0)
    whatToDraw()
    let result = UIGraphicsGetImageFromCurrentImageContext()!
    UIGraphicsEndImageContext()
    return result
}
```

My `imageOfSize` utility is so useful that I declare it at the top level of a file, where all my files can see it. To make an image, I can perform step 2 (the actual drawing) in a function and pass that function as argument to the `imageOfSize` utility:

```
func drawing() {
    let p = UIBezierPath(
        roundedRect: CGRect(x:0, y:0, width:45, height:20),
        cornerRadius: 8)
    p.stroke()
}
let image = imageOfSize(CGSize(width:45, height:20), drawing)
```

Now *that* is a beautifully expressive and clear way to turn drawing instructions into an image.

The Cocoa API is full of situations where you'll pass a function to be called by the runtime in some special way or at some later time. Some common Cocoa situations even involve passing *two* functions. For instance, when you perform view animation, you'll often pass one function prescribing the action to be animated and another function saying what to do afterward:

```
func whatToAnimate() { // self.myButton is a button in the interface
    self.myButton.frame.origin.y += 20
}
func whatToDoLater(finished:Bool) {
    print("finished: \(finished)")
}
UIView.animate(withDuration:0.4,
    animations: whatToAnimate, completion: whatToDoLater)
```

That means: Change the frame origin (that is, the position) of this button in the interface, but do it over time (four-tenths of a second); and then, when that's finished, print a log message in the console saying whether the animation was performed or not.

Type Aliases Can Clarify Function Types

To make function type specifiers clearer, we can take advantage of Swift's type alias feature to give a function type a name. The name can be descriptive, and the possibly confusing arrow operator notation is avoided:

```
typealias VoidVoidFunction = () -> ()
func dothis(_ f:VoidVoidFunction) {
    f()
}
```

The Cocoa documentation will often describe a function to be passed in this way as a *handler*, and will refer it as a *block*, because that's the Objective-C syntactic construct needed here. In Swift, it's a function.

Anonymous Functions

Consider once again this example:

```
func whatToAnimate() { // self.myButton is a button in the interface
    self.myButton.frame.origin.y += 20
}
func whatToDoLater(finished:Bool) {
    print("finished: \(finished)")
}
UIView.animate(withDuration:0.4,
    animations: whatToAnimate, completion: whatToDoLater)
```

There's a slight bit of ugliness in that code. I'm declaring functions `whatToAnimate` and `whatToDoLater`, just because I want to pass those functions in the last line. But I don't really need the *names* `whatToAnimate` and `whatToDoLater` for anything, except to refer to them in the last line; neither the names nor the functions will ever be used again. In my call to `UIView.animate(withDuration:animations:completion:)`, it would be nice to be able to pass just the *body* of those functions *without* a declared name.

We can do that. A nameless function body is called an *anonymous* function, and is legal and common in Swift. To form an anonymous function, you do two things:

1. Create the function body itself, including the surrounding curly braces, but with no function declaration.
2. If necessary, express the function's parameter list and return type as the first thing *inside* the curly braces, followed by the keyword `in`.

Let's practice by transforming our named function declarations into anonymous functions. Here's the named function declaration for `whatToAnimate`:

```
func whatToAnimate() {
    self.myButton.frame.origin.y += 20
}
```

Here's an anonymous function that does the same thing. Notice how I've moved the parameter list and return type inside the curly braces:

```
{
    () -> () in
    self.myButton.frame.origin.y += 20
}
```

Here's the named function declaration for `whatToDoLater`:

```
func whatToDoLater(finished:Bool) {
    print("finished: \(finished)")
}
```

Here's an anonymous function that does the same thing:

```
{
    (finished:Bool) -> () in
    print("finished: \(finished)")
}
```

Using Anonymous Functions Inline

Now that we know how to make anonymous functions, let's use them. The point where we need the functions is the point where we're passing the second and third arguments to `animate(withDuration:animations:completion:)`. We can create and pass anonymous functions *right at that point*, like this:

```
UIView.animate(withDuration:0.4,
    animations: {
        () -> () in
        self.myButton.frame.origin.y += 20
    },
    completion: {
        (finished:Bool) -> () in
        print("finished: \(finished)")
    }
)
```

We can make the same improvement in the way we call the `imageOfSize` function from the preceding section. Earlier, we called that function like this:

```
func drawing() {
    let p = UIBezierPath(
        roundedRect: CGRect(x:0, y:0, width:45, height:20),
        cornerRadius: 8)
    p.stroke()
}
let image = imageOfSize(CGSize(width:45, height:20), drawing)
```

We now know, however, that we don't need to declare the `drawing` function separately. We can call `imageOfSize` with an anonymous function:

```
let image = imageOfSize(CGSize(width:45, height:20), {
    () -> () in
    let p = UIBezierPath(
        roundedRect: CGRect(x:0, y:0, width:45, height:20),
        cornerRadius: 8)
    p.stroke()
})
```

Anonymous functions are very commonly used in Swift, so make sure you can read and write that code!

Anonymous Function Abbreviated Syntax

Anonymous functions are so common and so important in Swift that some shortcuts for writing them are provided:

Omission of the return type
: If the anonymous function's return type is already known to the compiler, you can omit the arrow operator and the specification of the return type:

```
UIView.animate(withDuration:0.4,
    animations: {
        () in // *
        self.myButton.frame.origin.y += 20
    }, completion: {
        (finished:Bool) in // *
        print("finished: \(finished)")
})
```

(Occasionally the compiler will fail to infer the anonymous function's return type, even though you think it should be obvious, and will give a compile error. If that happens, just don't use this shortcut: supply an `in` expression with an explicit return type.)

Omission of the `in` *expression when there are no parameters*
: If the anonymous function takes no parameters, and if the return type can be omitted, the `in` expression itself can be omitted:

```
UIView.animate(withDuration:0.4,
    animations: { // *
        self.myButton.frame.origin.y += 20
    }, completion: {
        (finished:Bool) in
        print("finished: \(finished)")
})
```

Omission of the parameter types
: If the anonymous function takes parameters and their types are already known to the compiler, the types can be omitted:

```
UIView.animate(withDuration:0.4,
    animations: {
        self.myButton.frame.origin.y += 20
    }, completion: {
        (finished) in // *
        print("finished: \(finished)")
})
```

Omission of the parentheses
: If the parameter types are omitted, the parentheses around the parameter list can be omitted:

```
UIView.animate(withDuration:0.4,
    animations: {
        self.myButton.frame.origin.y += 20
    }, completion: {
        finished in // *
        print("finished: \(finished)")
})
```

Omission of the `in` *expression even when there are parameters*
: If the return type can be omitted, and if the parameter types are already known to the compiler, you can omit the `in` expression and refer to the parameters directly within the body of the anonymous function by using the magic names `$0`, `$1`, and so on, in order:

```
UIView.animate(withDuration:0.4,
    animations: {
        self.myButton.frame.origin.y += 20
    }, completion: {
        print("finished: \($0)") // *
})
```

Omission of the parameter names
: If the anonymous function body doesn't need to refer to a parameter, you can substitute an underscore for its name in the parameter list in the `in` expression:

```
UIView.animate(withDuration:0.4,
    animations: {
        self.myButton.frame.origin.y += 20
    }, completion: {
        _ in // *
        print("finished!")
})
```

If an anonymous function takes parameters, you *must* acknowledge them somehow. You can omit the `in` expression and use the parameters by the magic names `$0` and so on. Or you can keep the `in` expression and give the parameters names or ignore them with underscores. But you can't omit the `in` expression and *not* use the parameters' magic names! If you do, your code won't compile.

Omission of the function argument label

If your anonymous function is the last argument being passed in this function call — which will just about always be the case — you can close the function call with a right parenthesis *before* this last argument, and then put just the anonymous function body *without a label.* This is called *trailing closure syntax* (I'll explain in a moment what a closure is):

```
UIView.animate(withDuration:0.4,
    animations: {
        self.myButton.frame.origin.y += 20
    }) { // *
        _ in
        print("finished!")
}
```

In that code, the `completion:` parameter is last, so the call can pass the anonymous function argument outside the call's parentheses, using trailing closure syntax with no label.

But there's a curious asymmetry in that particular example, because this is a method that takes *two* function parameters. The `animations:` parameter is an anonymous function too, but it still sits inside the parentheses. A Swift 5.3 innovation resolved that asymmetry: now, *multiple* anonymous function arguments can be expressed using trailing closure syntax. When you do that, the *first* anonymous function takes no label; the remaining functions *do* each have their labels, with *no comma:*

```
UIView.animate(withDuration:0.4) { // *
    self.myButton3.frame.origin.y += 20
} completion: { // *
    _ in
    print("finished")
}
```

Omission of the calling function parentheses

If you use a trailing closure, and if the function you are calling takes no parameters other than the function you are passing to it, you can omit the empty parentheses from the call. This is the *only* situation in which you can omit the parentheses from a function call! To illustrate, I'll declare and call a different function:

```
func doThis(_ f:() -> ()) {
    f()
}
doThis { // no parentheses!
    print("Howdy")
}
```

Omission of the keyword `return`

If the anonymous function body consists of exactly one statement consisting of returning a value with the keyword `return`, the keyword `return` can be omitted (and in this situation, I like to do so):

```
func greeting() -> String {
    return "Howdy"
}
func performAndPrint(_ f:()->String) {
    let s = f()
    print(s)
}
performAndPrint {
    greeting() // meaning: return greeting()
}
```

When writing anonymous functions, you will frequently find yourself taking advantage of all the omissions you are permitted. In addition, you'll sometimes shorten the *layout* of the code (though not the code itself) by putting the whole anonymous function together with the function call *on one line*. Thus, Swift code involving anonymous functions can be extremely compact.

Here's a typical example. We start with an array of Int values and generate a new array consisting of all those values multiplied by 2, by calling the `map(_:)` instance method. The `map(_:)` method of an array takes a function. That function takes one parameter of the same type as the array's elements and returns a new value; here, our array is made of Int values, and we are passing to the `map(_:)` method a function that takes one Int parameter and returns an Int. We could write out the whole function, like this:

```
let arr = [2, 4, 6, 8]
func doubleMe(i:Int) -> Int {
    return i*2
}
let arr2 = arr.map(doubleMe) // [4, 8, 12, 16]
```

That, however, is not very Swifty. We don't need the name `doubleMe` for anything else, so this may as well be an anonymous function:

```
let arr = [2, 4, 6, 8]
let arr2 = arr.map ({
    (i:Int) -> Int in
    return i*2
})
```

Now let's abbreviate our anonymous function. Its parameter type is known in advance, so we don't need to specify that. Its return type is known by inspection of the function body, so we don't need to specify that. There's just one parameter and we are going to use it, so we don't need the `in` expression as long we refer to the parameter as `$0`. Our function body consists of just one statement, and it is a return statement, so we can omit `return`. And `map(_:)` doesn't take any other parameters, so we can omit the parentheses and follow the name directly with a trailing closure:

```
let arr = [2, 4, 6, 8]
let arr2 = arr.map {$0*2}
```

It doesn't get any Swiftier than that!

Define-and-Call

A pattern that's surprisingly common in Swift is to define an anonymous function and call it, all in one move:

```
{
    // ... code goes here
}()
```

Notice the parentheses after the curly braces! The curly braces *define* an anonymous function body; the parentheses *call* that anonymous function. I call this construct *define-and-call*.

Using define-and-call, an action can be taken at the point where it is needed, rather than in a series of preparatory steps. For instance, there's a common Cocoa situation where we create and configure an NSMutableParagraphStyle and then use it as an argument in a call to the NSMutableAttributedString method `addAttribute(_:value:range:)`, like this:

```
let para = NSMutableParagraphStyle()
para.headIndent = 10
para.firstLineHeadIndent = 10
// ... more configuration of para ...
content.addAttribute( // content is an NSMutableAttributedString
    .paragraphStyle,
    value:para,
    range:NSRange(location:0, length:1))
```

I find that code ugly. We don't need `para` except to pass it as the `value:` argument within the call to `addAttribute(_:value:range:)`, so it would be much nicer to create and configure it right there within the call, as the `value:` argument itself. That

sounds like an anonymous function — except that the `value:` parameter is not a function, but an NSMutableParagraphStyle object.

We can solve the problem by providing, as the `value:` argument, an anonymous function that *produces* an NSMutableParagraphStyle object and calling it so that it *does* produce an NSMutableParagraphStyle object:

```
content.addAttribute(
    .paragraphStyle,
    value: {
        let para = NSMutableParagraphStyle()
        para.headIndent = 10
        para.firstLineHeadIndent = 10
        // ... more configuration of para ...
        return para
    }(),
    range:NSRange(location:0, length:1))
```

I'll demonstrate some further uses of define-and-call in Chapter 3.

Closures

Swift functions are *closures*. This means they can *capture* references to external variables in scope within the body of the function. What do I mean by that? Well, recall from Chapter 1 that code in curly braces constitutes a scope, and this code can "see" variables and functions declared in a surrounding scope. For example:

```
class Dog {
    var whatThisDogSays = "woof" ❶
    func bark() {
        print(self.whatThisDogSays) ❷
    }
}
```

In that code:

❶ The variable `whatThisDogSays` is *external* to the function: it is *declared outside* the body of the function, and yet is *in scope* for the body of the function, so that the code inside the body of the function can see it.

❷ The code inside the body of the function *refers* to the external variable `whatThisDogSays` — it says, explicitly, `whatThisDogSays`.

So far, so good; but we now know that the function `bark` can be passed as a value. In effect, it can travel from one environment to another. When it does, what happens to that reference to `whatThisDogSays`? Let's find out:

```
func doThis(_ f : () -> ()) {
    f()
}
let d = Dog()
d.whatThisDogSays = "arf"
let barkFunction = d.bark
doThis(barkFunction) // arf
```

We run that code, and `"arf"` appears in the console!

Perhaps that result doesn't seem very surprising to you. But think about it. We do not directly *call* `d.bark()`. We make a Dog instance and *pass* its `bark` function as a value into the function `doThis`. There, it is called. Now, `whatThisDogSays` is an instance property of a particular Dog. Inside the function `doThis` there is no `whatThisDogSays`. Indeed, inside the function `doThis` there is no Dog instance! Nevertheless the call `f()` still works. The function `d.bark`, as it is passed around, evidently *carries* the variable `whatThisDogSays` along with itself.

But there's more. I'll change the example by moving the line where we set `d.whatThisDogSays` to *after* we assign `d.bark` into our variable `barkFunction`:

```
func doThis(_ f : () -> ()) {
    f()
}
let d = Dog()
let barkFunction = d.bark      ❶
doThis(barkFunction) // woof   ❷
d.whatThisDogSays = "arf"      ❸
doThis(barkFunction) // arf    ❹
```

What just happened?

❶ We assigned `d.bark` to `barkFunction`, once and for all; after that, we never changed `barkFunction`.

❷ At that time, `d.whatThisDogSays` was `"woof"`, so we passed `barkFunction` into `doThis` and got `"woof"`.

❸ We then changed `d.whatThisDogSays` to `"arf"`. We didn't change `barkFunction`.

❹ We passed `barkFunction` into `doThis` again, and this time we got `"arf"`!

After creating both `d` and `barkFunction`, changing a property of the Dog `d` changes the output of calling `barkFunction`! How can this be? Evidently, after step 1, when `d.bark` has been assigned to `barkFunction`, *both* our Dog variable `d` *and* the function `barkFunction` are holding references to the same Dog instance. This is because `d.bark`, which we assigned to `barkFunction`, refers to `self`, which *is* the Dog

instance. That's what we mean when we say that a function is a closure and that it *captures* external variables referred to in its body.

How Closures Improve Code

You can use the fact that functions are closures to make your code more general, and hence more useful. To illustrate, here, once again, is my earlier example of a function that accepts drawing instructions and performs them to generate an image:

```
func imageOfSize(_ size:CGSize, _ whatToDraw:() -> ()) -> UIImage {
    UIGraphicsBeginImageContextWithOptions(size, false, 0)
    whatToDraw()
    let result = UIGraphicsGetImageFromCurrentImageContext()!
    UIGraphicsEndImageContext()
    return result
}
```

As you know, we can call `imageOfSize` with a trailing closure:

```
let image = imageOfSize(CGSize(width:45, height:20)) {
    let p = UIBezierPath(
        roundedRect: CGRect(x:0, y:0, width:45, height:20),
        cornerRadius: 8)
    p.stroke()
}
```

That code, however, contains an annoying repetition. This is a call to create an image of a given size consisting of a rounded rectangle of that size. We are repeating the size; the pair of numbers `45,20` appears twice. That's silly. Let's prevent the repetition by putting the size into a variable at the outset:

```
let sz = CGSize(width:45, height:20)
let image = imageOfSize(sz) {
    let p = UIBezierPath(
        roundedRect: CGRect(origin:CGPoint.zero, size:sz),
        cornerRadius: 8)
    p.stroke()
}
```

The variable `sz`, declared outside our anonymous function at a higher level, is visible inside it. Thus we can refer to it inside the anonymous function — and we do so. The anonymous function is just a function body; it won't be *executed* until `imageOfSize` calls it. Nevertheless, when we refer to `sz` from inside the function body in the expression `CGRect(origin:CGPoint.zero, size:sz)`, we capture its value *now*, because the function body is a closure. When `imageOfSize` calls `whatToDraw`, and `whatToDraw` turns out to be a function whose body refers to a variable `sz`, there's no problem, even though there is no `sz` anywhere in the neighborhood of `imageOfSize`.

Now let's go further. So far, we've been hard-coding the size of the desired rounded rectangle. Imagine, though, that creating images of rounded rectangles of various

sizes is something we do often. It would make sense to package this code up as a function, where `sz` is not a fixed value but a parameter; the function will then return the image:

```
func makeRoundedRectangle(_ sz:CGSize) -> UIImage {
    let image = imageOfSize(sz) {
        let p = UIBezierPath(
            roundedRect: CGRect(origin:CGPoint.zero, size:sz),
            cornerRadius: 8)
        p.stroke()
    }
    return image
}
```

Incredibly, that works! The parameter `sz` that arrives into `makeRoundedRectangle` is no longer a hard-coded value; we don't know what it will be. Nevertheless, when `makeRoundedRectangle` is called, `sz` *will* have a value, and the anonymous function captures `sz`, so when `imageOfSize` calls the anonymous function, `sz` inside that function will have the `sz` value that was passed to `makeRoundedRectangle`.

Our code is becoming beautifully compact. To call `makeRoundedRectangle`, supply a size; an image is returned. I can perform the call, obtain the image, and display that image, all in one move, like this (`self.iv` is a UIImageView in the interface):

```
self.iv.image = makeRoundedRectangle(CGSize(width:45, height:20))
```

Function Returning Function

But now let's go even further! Instead of returning an image, our function can return *a function* that makes rounded rectangles *of the specified size*. If you've never seen a function returned as a value from a function, you may now be gasping for breath. But a function, after all, can be used as a value. We have already passed a function *into* a function as an argument in the function call; now we are going to receive a function *from* a function call as its result:

```
func makeRoundedRectangleMaker(_ sz:CGSize) -> () -> UIImage { ❶
    func f () -> UIImage { ❷
        let im = imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPoint.zero, size:sz),
                cornerRadius: 8)
            p.stroke()
        }
        return im
    }
    return f ❸
}
```

Let's analyze that code slowly:

❶ The declaration is the hardest part. What on earth is the type (signature) of this function `makeRoundedRectangleMaker`? It is `(CGSize) -> () -> UIImage`. That expression has *two* arrow operators. To understand it, keep in mind that everything after each arrow operator is the type of a returned value. So `makeRoundedRectangleMaker` is a function that takes a CGSize parameter and returns a `() -> UIImage`. Okay, and what's a `() -> UIImage`? We already know that: it's a function that takes no parameters and returns a UIImage. So `makeRoundedRectangleMaker` is a function that takes a CGSize parameter and returns *a function* — a function that itself, when called with *no* parameters, will return a UIImage.

❷ Now here we are in the body of the function `makeRoundedRectangleMaker`, and our first step is to declare a function (a function-in-function, or local function) of precisely the type we intend to return, namely, one that takes no parameters and returns a UIImage. Here, we're naming this function `f`. The way this function works is simple and familiar: it calls `imageOfSize`, passing it an anonymous function that makes an image of a rounded rectangle (`im`) — and then it returns the image.

❸ Finally, we *return* the function we just made (`f`). We have fulfilled our contract: we said we would return a function that takes no parameters and returns a UIImage, and we do so.

But perhaps you are still gazing open-mouthed at `makeRoundedRectangleMaker`, wondering how you would ever call it and what you would get if you did. Let's try it:

```
let maker = makeRoundedRectangleMaker(CGSize(width:45, height:20))
```

What is the variable `maker` after that code runs? It's a *function* — a function that takes no parameters and that, when called, produces the image of a rounded rectangle of size `45,20`. You don't believe me? I'll prove it — by *calling* the function that is now the value of `maker`:

```
let maker = makeRoundedRectangleMaker(CGSize(width:45, height:20))
self.iv.image = maker()
```

Now that you've gotten over your stunned surprise at the notion of a function that produces a function as its result, turn your attention once again to the implementation of `makeRoundedRectangleMaker` and let's analyze it again, a different way. Remember, I didn't write that function to show you that a function can produce a function. I wrote it to illustrate closures! Let's think about how the environment gets captured:

```
func makeRoundedRectangleMaker(_ sz:CGSize) -> () -> UIImage {
    func f () -> UIImage {
        let im = imageOfSize(sz) { // *
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPoint.zero, size:sz), // *
                cornerRadius: 8)
            p.stroke()
        }
        return im
    }
    return f
}
```

The function `f` takes no parameters. Yet, twice within the function body of `f` (I've marked the places with asterisk comments), there are references to a size value `sz`. The body of the function `f` can see `sz`, the parameter of the surrounding function `makeRoundedRectangleMaker`, because it is in a surrounding scope. The function `f` *captures* the reference to `sz` at the time `makeRoundedRectangleMaker` is called, and *keeps* that reference when `f` is returned and assigned to `maker`:

```
let maker = makeRoundedRectangleMaker(CGSize(width:45, height:20))
```

That is why `maker` is now a function that, when it is called, creates and returns an image of the particular size `45,20` even though it itself will be called *with no parameters*. The knowledge of what size of image to produce has been *baked into* the function referred to by `maker`. Looking at it another way, `makeRoundedRectangleMaker` is a *factory* for creating a whole family of functions similar to `maker`, each of which produces an image of one particular size. That's a dramatic illustration of the power of closures.

Before I leave `makeRoundedRectangleMaker`, I'd like to rewrite it in a Swiftier fashion. Within `f`, there is no need to create `im` and then return it; we can return the result of calling `imageOfSize` directly:

```
func makeRoundedRectangleMaker(_ sz:CGSize) -> () -> UIImage {
    func f () -> UIImage {
        return imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPoint.zero, size:sz),
                cornerRadius: 8)
            p.stroke()
        }
    }
    return f
}
```

But there is no need to declare `f` and then return it either; it can be an anonymous function and we can return it directly:

```
func makeRoundedRectangleMaker(_ sz:CGSize) -> () -> UIImage {
    return {
        return imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPoint.zero, size:sz),
                cornerRadius: 8)
            p.stroke()
        }
    }
}
```

But our anonymous function consists of nothing but a `return` statement; the anonymous function parameter to `imageOfSize` consists of multiple statements, but the `imageOfSize` call itself is still just one Swift statement. We can omit the keyword `return` (and we could omit the remaining `return` too, but I prefer not to):

```
func makeRoundedRectangleMaker(_ sz:CGSize) -> () -> UIImage {
    return {
        imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPoint.zero, size:sz),
                cornerRadius: 8)
            p.stroke()
        }
    }
}
```

Closure Setting a Captured Variable

The power that a closure gets through its ability to capture its environment is even greater than I've shown so far. If a closure captures a reference to a variable outside itself, and if that variable is settable, then *the closure can set the variable*.

Let's say I've declared this simple function. All it does is to accept a function that takes an Int parameter, and to call that function with an argument of 100:

```
func pass100 (_ f:(Int) -> ()) {
    f(100)
}
```

Now, look closely at this code and try to guess what will happen when we run it:

```
var x = 0
print(x) // ?
func setX(newX:Int) {
    x = newX
}
pass100(setX)
print(x) // ?
```

The first `print(x)` call obviously produces `0`. The second `print(x)` call produces `100`! The `pass100` function has reached into my code and *changed* the value of my variable

x. That's because the function setX that I passed to pass100 contains a reference to x; not only does it contain it, but it captures it; not only does it capture it, but it sets its value. That x is my x. The pass100 function was able to set my x just as readily as I would have set it by calling setX directly.

Closure Preserving Captured Environment

When a closure captures its environment, it *preserves* that environment *even if nothing else does*. Here's an example calculated to blow your mind — a function that modifies a function:

```
func countAdder(_ f: @escaping () -> ()) -> () -> () {
    var ct = 0
    return {
        ct = ct + 1
        print("count is \(ct)")
        f()
    }
}
```

The function countAdder accepts a function as its parameter and returns a function as its result. (I'll explain the @escaping attribute in the next section.) The function that it returns calls the function that it accepts, with a little bit added: it increments a variable and reports the result. So now try to guess what will happen when we run this code:

```
func greet () {
    print("howdy")
}
let countedGreet = countAdder(greet)
countedGreet() // ?
countedGreet() // ?
countedGreet() // ?
```

What we've done here is to take a function greet, which prints "howdy", and pass it through countAdder. What comes out the other side of countAdder is a new function, which we've named countedGreet. We then call countedGreet three times. Here's what appears in the console:

```
count is 1
howdy
count is 2
howdy
count is 3
howdy
```

Clearly, countAdder has added to the functionality of the function that was passed into it *the ability to report how many times it is called*. Now ask yourself: Where on earth is the variable that maintains this count? Inside countAdder, it was a local

variable `ct`. But it isn't declared inside the anonymous function that `countAdder` returns. That's deliberate! If it *were* declared inside the anonymous function, we would be setting `ct` to `0` every time `countedGreet` is called — we wouldn't be counting. Instead, `ct` is initialized to `0` once *and then captured* by the anonymous function. This variable is preserved as part of the *environment* of `countedGreet` — it is *outside* `countedGreet` in some mysterious environment-preserving world, so that it can be incremented every time `countedGreet` is called.

Escaping Closures

If a function passed around as a value will be preserved for later execution, rather than being called directly, it is a closure that captures and preserves its environment *over time*. That's called an *escaping* closure. In some situations, the function's type must be explicitly marked `@escaping`. The compiler will detect violations of this rule, so if you find the rule confusing, don't worry about it; just let the compiler enforce it for you.

This function is legal because it receives a function and calls it directly:

```
func funcCaller(f:() -> ()) {
    f()
}
```

And this function is legal, even though it returns a function to be executed later, because it also *creates* that function internally. The function that it returns *is* an escaping closure, but the type of the function's returned value does not have to be marked as `@escaping`:

```
func funcMaker() -> () -> () {
    return { print("hello world") }
}
```

But this function is illegal. It receives a function as a parameter *and* returns that function to be executed later:

```
func funcPasser(f:() -> ()) -> () -> () { // compile error
    return f
}
```

The solution is to mark the type of the incoming parameter `f` as `@escaping`, and the compiler will prompt you to do so:

```
func funcPasser(f:@escaping () -> ()) -> () -> () {
    return f
}
```

A secondary feature of escaping closures is that, when you refer to a property or method of `self` within the function body, the compiler may insist that you say `self`

explicitly. That's because such a reference *captures* `self`, and the compiler wants you to acknowledge this fact by *saying* `self`:

```
let f1 = funcPasser {
    print(view.bounds) // compile error, because self.view is implied
}
let f2 = funcPasser {
    print(self.view.bounds) // ok
}
```

I'll say more about *why* you must acknowledge a captured `self` when I talk about memory management in Chapter 5.

Capture Lists

Sometimes, you might want a function to refer to a variable outside itself, just in order to get its value, but *without* capturing the variable. Swift provides an ingenious syntax for doing that — but only when the function is an anonymous function. At the start of the anonymous function body, you put square brackets containing a comma-separated list of references to variables in the surrounding environment. This is called a *capture list*.

If you have a capture list, you must follow it with the keyword `in` if there's no `in` expression already. If there *is* an `in` expression already, the capture list precedes any parameter names.

The result is as if the value had been passed into the anonymous function *as a parameter* rather than by closure capture; like a parameter, it behaves as if it had been declared with `let` ("Modifiable Parameters" on page 36).

To illustrate, let's start with a simple example that captures a reference in the usual way. Remember, I'm going to need an anonymous function here, so I'll declare a variable and assign the anonymous function to it. Test yourself by guessing what the output is:

```
var x = 0
let f : () -> () = {
    print(x)
}
f() // ?
x = 1
f() // ?
```

The first call to `f()` prints `0`. The second call to `f()` prints `1`. That's similar to the very first closure example I gave earlier. The variable `x` is declared outside the function `f`. The function `f` captures the variable `x` and prints it out. If we *change* the value of `x` and call `f`, it prints out the *new* value of `x`.

Now let's see what happens if we include `x` in a capture list:

```
var x = 0
let f : () -> () = { [x] in // *
    print(x)
}
f()
x = 1
f()
```

This time, the call to `f()` prints out `0` — *both times.* We changed `x` to `1`, but `f` doesn't care. Because of the capture list, `f` had already captured the *value* of `x` as `0` at the time it was declared. In effect, `x` behaves like an ordinary parameter passed into `f`; it's a constant. Another way to see this is that if `f` tries to set the value of `x`, the compiler will complain: "Cannot assign to value: `x` is an immutable capture."

It is also legal to assign a capture list expression to another name. For example, a capture list can say `[y=x]`; that's as if you were passing `x` into the anonymous function as a parameter, but now the parameter is called `y`. This is useful when there is something unfortunate about the original name, or when the value to be passed in is an expression that needs to be evaluated at the time of assignment. Here's an example from my own code (without explanation of the context):

```
self.undoer.registerUndo(withTarget: self) {
    [oldCenter = self.center] myself in
    myself.setCenterUndoably(oldCenter)
}
```

The anonymous function passed as a parameter to `registerUndo` is an escaping function. My capture list declares a constant `oldCenter` and sets its value to `self.center`. As a result, the value of `self.center` is evaluated *now*, when `registerUndo` is called, rather than later, when the anonymous function itself is called (at which time the value of `self.center` may have changed).

Another use of capture lists is to work around the requirement to say `self` explicitly in an escaping closure. Recall this example from the preceding section:

```
let f2 = funcPasser {
    print(self.view.bounds)
}
```

We are forced to say `self` explicitly because this is an escaping closure. But many people don't like saying `self` explicitly; and if this anonymous function implies `self` many times, it may be necessary to say `self` explicitly many times. Starting in Swift 5.3, there's a stylistic alternative: if you put `self` in the capture list, that satisfies the compiler, and you don't have to write `self` as the implicit target in the function body:

```
let f2 = funcPasser { [self] in
    print(view.bounds) // ok
}
```

Curried Functions

Return once more to `makeRoundedRectangleMaker`:

```
func makeRoundedRectangleMaker(_ sz:CGSize) -> () -> UIImage {
    return {
        imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPoint.zero, size:sz),
                cornerRadius: 8)
            p.stroke()
        }
    }
}
```

There's something I still don't like about this method: the size of the rounded rectangle that it creates is a parameter (`sz`), but the `cornerRadius` of the rounded rectangle is hard-coded as `8`. I'd like the ability to pass a value for the corner radius as part of the call. I can think of two ways to do that. One is to give `makeRoundedRectangleMaker` itself another parameter:

```
func makeRoundedRectangleMaker(_ sz:CGSize, _ r:CGFloat) -> () -> UIImage {
    return {
        imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPoint.zero, size:sz),
                cornerRadius: r)
            p.stroke()
        }
    }
}
```

And we would then call it like this:

```
let maker = makeRoundedRectangleMaker(CGSize(width:45, height:20), 8)
```

But there's another way. The function that we are returning from `makeRoundedRectangleMaker` takes no parameters. Instead, *it* could take the extra parameter:

```
func makeRoundedRectangleMaker(_ sz:CGSize) -> (CGFloat) -> UIImage {
    return { r in
        imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPoint.zero, size:sz),
                cornerRadius: r)
            p.stroke()
        }
    }
}
```

Now `makeRoundedRectangleMaker` returns a function that, itself, takes one parameter, so we must remember to supply that when we call it:

```
let maker = makeRoundedRectangleMaker(CGSize(width:45, height:20))
self.iv.image = maker(8)
```

If we don't need to conserve `maker` for anything, we can of course do all that in one line — a function call that yields a function which we immediately call to obtain our image:

```
self.iv.image = makeRoundedRectangleMaker(CGSize(width:45, height:20))(8)
```

When a function returns a function that takes a parameter in this way, it is called a *curried* function (after the computer scientist Haskell Curry).

Function References and Selectors

When you want to refer to a function by name — perhaps in order to pass it as argument to another function — you can often use its bare name. That's what I've been doing throughout this chapter, in examples like this:

```
func whatToAnimate() { // self.myButton is a button in the interface
    self.myButton.frame.origin.y += 20
}
func whatToDoLater(finished:Bool) {
    print("finished: \(finished)")
}
UIView.animate(withDuration:0.4,
    animations: whatToAnimate, completion: whatToDoLater) // *
```

A bare name like `whatToAnimate` or `whatToDoLater` is a *function reference*. It consists of the base name alone, the part of the function's name that precedes the parentheses. The lack of parentheses makes it clear that this is a reference, not a call. Use of the bare name as a function reference is legal when it's unambiguous: in this particular context, there's only one function called `whatToDoLater` in scope, and I'm using its name as argument in a function call where the parameter type is known (namely, `(Bool) -> ()`).

But now consider the following situation. Just as I can pass a function as an argument, I can assign a function as a value to a variable. And suppose I have *two* functions with the same name, one that takes a parameter, and one that doesn't:

```
class Dog {
    func bark() {
        print("woof")
    }
    func bark(_ loudly:Bool) {
        if loudly {
            print("WOOF")
        } else {
            self.bark()
        }
    }
```

```
        func test() {
            let barkFunction = bark // compile error
            // ...
        }
    }
```

That code won't compile, because the bare name `bark` is ambiguous in this context: which `bark` method does it refer to? To solve this problem, Swift provides a notation allowing you to refer to a function more precisely. This notation has two parts:

Full name
: The full name of a Swift function is the base name along with parentheses containing the external names of its parameters, each followed by colon (and no commas or spaces). If the external name of a parameter is suppressed, we represent its external name as an underscore.

Signature
: The signature of a Swift function may be appended to its bare name (or full name) with the keyword `as`.

So, for example:

```
func say(_ s:String, times:Int) {
```

That method has full name `say(_:times:)` but may be referred to using a bare name and signature as `say as (String, Int) -> ()`.

In our `bark` example, use of the full name solves the problem if the function to which we want a reference is the one that takes a parameter:

```
class Dog {
    func bark() {
        // ... as before ...
    }
    func bark(_ loudly:Bool) {
        // ... as before ...
    }
    func test() {
        let barkFunction = bark(_:) // fine
    }
}
```

But use of the full name *doesn't* solve the problem if the function to which we want a reference is the one that takes *no* parameters, because in that case the full name is the bare name, which is exactly what's ambiguous in this context. Use of the signature solves the problem:

```
class Dog {
    func bark() {
        // ... as before ...
    }
```

```
        func bark(_ loudly:Bool) {
            // ... as before ...
        }
        func test() {
            let barkFunction = bark as () -> () // fine
        }
    }
```

Obviously, an explicit signature is needed also when a function is *overloaded*:

```
    class Dog {
        func bark() {
        }
        func bark(_ loudly:Bool) {
        }
        func bark(_ times:Int) {
        }
        func test() {
            let barkFunction = bark(_:) // compile error
        }
    }
```

Here, we have said that we want the `bark` that takes one parameter, but there are *two* such `bark` functions, one whose parameter is a Bool, the other whose parameter is an Int. The signature disambiguates (and we can use the bare name):

```
    let barkFunction = bark as (Int) -> () // "times", not "loudly"
```

Here's an overloaded method where one overload has a default parameter:

```
    class Dog {
        func bark(loudly:Bool = false) {
        }
        func bark(softly:Bool) {
        }
        func test() {
            self.bark() // fine
            let barkFunction = bark // compile error
        }
    }
```

A *call* to `self.bark()` is resolved to the `bark` method with a default parameter, but a function *reference* `bark` is rejected as ambiguous; if the function reference means `bark(loudly:)`, it must say so.

Function Reference Scope

In the foregoing examples of function references, there was no need to tell the compiler *where* the function is defined. That's because the function is already in scope at the point where the function reference appears. If you can *call* the function without supplying further information, you can form the function *reference* without supplying further information.

However, a function reference *can* supply further information about where a function is defined; and sometimes it *must* do so. This is done by prefixing an instance or class to the function reference, using dot-notation. There are situations where the compiler would force you to use `self` to call a function; in those situations, you will have to use `self` to refer to the function as well:

```
class Dog {
    func bark() {
    }
    func bark(_ loudly:Bool) {
    }
    func test() {
        let f = {
            return self.bark(_:) // self required here
        }
    }
}
```

To form a function reference to an instance method of another type, you have two choices. If you have on hand an instance of that type, you can use dot-notation with a reference to that instance:

```
class Cat {
    func purr() {
    }
}
class Dog {
    let cat = Cat()
    func test() {
        let purrFunction = cat.purr
    }
}
```

The other possibility is to use *the type* with dot-notation (this works even if the function is an instance method):

```
class Cat {
    func purr() {
    }
}
class Dog {
    func bark() {
    }
    func test() {
        let barkFunction = Dog.bark // legal but not necessary
        let purrFunction = Cat.purr
    }
}
```

If you use the type with dot-notation and you need to disambiguate the function reference by giving its signature, the signature must describe the curried static/class version of the instance method (see "The Secret Life of Instance Methods" on page 138):

```
class Cat {
    func purr() {
    }
    func purr(_ loudly:Bool) {
    }
}
class Dog {
    func test() {
        let purrFunction = Cat.purr as (Cat) -> () -> Void
    }
}
```

Selectors

In Objective-C, a selector is a kind of method reference. In iOS programming, you might have to call a Cocoa method that wants a selector as one of its parameters; typically, this parameter will be named either `selector:` or `action:`. Usually, such a method also requires that you provide a *target* (an object reference); the idea is that the runtime can later call the method by turning the selector into a message and sending that message to that target.

Unfortunately, this architecture can be extremely risky. The reason is that to form the selector, it is necessary to construct a string representing a method's Objective-C name. If you construct that string incorrectly, then when the time comes to send the message to the target, the runtime will find that the target can't handle that message, because it has no such method, and the app comes to a violent and premature halt, dumping into the console the dreaded phrase "unrecognized selector." Here's a typical recipe for failure:

```
class ViewController : UIViewController {
    @IBOutlet var button : UIButton!
    func viewDidLoad() {
        super.viewDidLoad()
        self.button.addTarget( // prepare to crash!
            self, action: "buttonPressed", for: .touchUpInside)
    }
    @objc func buttonPressed(_ sender: Any) {
        // ...
    }
}
```

In that code, `self.button` is a button in the interface, and we are configuring it by calling `addTarget(_:action:for:)`, so that when the button is tapped, our `buttonPressed` method will be called. But we are configuring it incorrectly! Unfortunately, `"buttonPressed"` is *not* the Objective-C name of our `buttonPressed` method; the correct name would have been `"buttonPressed:"`, with a colon. (I'll explain why in the Appendix.) Therefore, our app will crash when the user taps that button.

The point is that if you don't know the rules for forming a selector string — or even if you do, but you make a typing mistake — an "unrecognized selector" crash is likely to result. Humans are fallible, and therefore "unrecognized selector" crashes have historically been extremely common among iOS programmers. The Swift compiler, however, is *not* fallible in this way. Therefore, Swift provides a way to let the compiler form the selector for you, by means of `#selector` syntax.

To ask the compiler to form an Objective-C selector for you, you use `#selector(...)` with a function reference inside the parentheses. We would rewrite our button action example like this:

```
class ViewController : UIViewController {
    @IBOutlet var button : UIButton!
    func viewDidLoad() {
        super.viewDidLoad()
        self.button.addTarget(
            self, action: #selector(buttonPressed), for: .touchUpInside)
    }
    @objc func buttonPressed(_ sender: Any) {
        // ...
    }
}
```

When you use that notation, two wonderful things happen:

The compiler validates the function reference
: If your function reference isn't valid, your code won't even compile. The compiler also checks that this function is exposed to Objective-C; there's no point forming a selector for a method that Objective-C can't see, as your app would crash if Objective-C were to try to call such a method. To ensure Objective-C visibility, the method may need to be marked with the `@objc` attribute; the compiler will enforce this requirement.

The compiler forms the Objective-C selector for you
: If your code compiles, the actual selector that will be passed into this parameter is guaranteed to be correct. *You* might form the selector incorrectly, but the compiler won't! It is impossible that the resulting selector should fail to match the method, and there is no chance of an "unrecognized selector" crash.

Very rarely, you still might need to create a selector manually. To do so, you can use a string, or you can instantiate Selector with the string as argument — for example, `Selector("woohoo:")`.

You can still crash, even with `#selector` syntax, by sending an action message to the *wrong target*. In the preceding example, if you changed `self`, the first argument of the `addTarget` call, to `self.button`, you'd crash at runtime with "unrecognized selector" — because the `buttonPressed` method is declared in ViewController, not in UIButton. Unfortunately, the compiler won't help you with this kind of mistake.