Classes and Structures

Classes and *structures* are general-purpose, flexible constructs that become the building blocks of your program's code. You define properties and methods to add functionality to your classes and structures by using exactly the same syntax as for constants, variables, and functions.

Unlike other programming languages, Swift does not require you to create separate interface and implementation files for custom classes and structures. In Swift, you define a class or a structure in a single file, and the external interface to that class or structure is automatically made available for other code to use.

> NOTE
>
> An instance of a *class* is traditionally known as an *object*. However, Swift classes and structures are much closer in functionality than in other languages, and much of this chapter describes functionality that can apply to instances of *either* a class or a structure type. Because of this, the more general term *instance* is used.

Comparing Classes and Structures

Classes and structures in Swift have many things in common. Both can:

- Define properties to store values
- Define methods to provide functionality
- Define subscripts to provide access to their values using subscript syntax
- Define initializers to set up their initial state
- Be extended to expand their functionality beyond a default implementation
- Conform to protocols to provide standard functionality of a certain kind

For more information, see Properties, Methods, Subscripts, Initialization, Extensions, and Protocols.

Classes have additional capabilities that structures do not:

- Inheritance enables one class to inherit the characteristics of another.
- Type casting enables you to check and interpret the type of a class instance at runtime.
- Deinitializers enable an instance of a class to free up any resources it has assigned.
- Reference counting allows more than one reference to a class instance.

For more information, see Inheritance, Type Casting, Deinitialization, and Automatic Reference Counting.

> NOTE
>
> Structures are always copied when they are passed around in your code, and do not use reference counting.

Definition Syntax

Classes and structures have a similar definition syntax. You introduce classes with the `class` keyword and structures with the `struct` keyword. Both place their entire definition within a pair of braces:

```
class SomeClass {
    // class definition goes here
}
struct SomeStructure {
    // structure definition goes here
}
```

> NOTE
>
> Whenever you define a new class or structure, you effectively define a brand new Swift type. Give types

> `UpperCamelCase` names (such as `SomeClass` and `SomeStructure` here) to match the capitalization of standard Swift types (such as `String`, `Int`, and `Bool`). Conversely, always give properties and methods `lowerCamelCase` names (such as `frameRate` and `incrementCount`) to differentiate them from type names.

Here's an example of a structure definition and a class definition:

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

The example above defines a new structure called `Resolution`, to describe a pixel-based display resolution. This structure has two stored properties called `width` and `height`. Stored properties are constants or variables that are bundled up and stored as part of the class or structure. These two properties are inferred to be of type `Int` by setting them to an initial integer value of `0`.

The example above also defines a new class called `VideoMode`, to describe a specific video mode for video display. This class has four variable stored properties. The first, `resolution`, is initialized with a new `Resolution` structure instance, which infers a property type of `Resolution`. For the other three properties, new `VideoMode` instances will be initialized with an `interlaced` setting of `false` (meaning "noninterlaced video"), a playback frame rate of `0.0`, and an optional `String` value called `name`. The `name` property is automatically given a default value of `nil`, or "no `name` value", because it is of an optional type.

Class and Structure Instances

The `Resolution` structure definition and the `VideoMode` class definition only describe what a `Resolution` or `VideoMode` will look like. They themselves do not describe a specific resolution or video mode. To do that, you need to create an instance of the structure or class.

The syntax for creating instances is very similar for both structures and classes:

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

Structures and classes both use initializer syntax for new instances. The simplest form of initializer syntax uses the type name of the class or structure followed by empty parentheses, such as `Resolution()` or `VideoMode()`. This creates a new instance of the class or structure, with any properties initialized to their default values. Class and structure initialization is described in more detail in Initialization.

Accessing Properties

You can access the properties of an instance using *dot syntax*. In dot syntax, you write the property name immediately after the instance name, separated by a period (`.`), without any spaces:

```
print("The width of someResolution is \(someResolution.width)")
// Prints "The width of someResolution is 0"
```

In this example, `someResolution.width` refers to the `width` property of `someResolution`, and returns its default initial value of `0`.

You can drill down into sub-properties, such as the `width` property in the `resolution` property of a `VideoMode`:

```
print("The width of someVideoMode is \(someVideoMode.resolution.width)")
// Prints "The width of someVideoMode is 0"
```

You can also use dot syntax to assign a new value to a variable property:

```
someVideoMode.resolution.width = 1280
print("The width of someVideoMode is now \(someVideoMode.resolution.width)")
// Prints "The width of someVideoMode is now 1280"
```

> NOTE
>
> Unlike Objective-C, Swift enables you to set sub-properties of a structure property directly. In the last example above, the `width` property of the `resolution` property of `someVideoMode` is set directly, without your needing to set the entire `resolution` property to a new value.

Memberwise Initializers for Structure Types

All structures have an automatically-generated *memberwise initializer*, which you can use to initialize the member properties of new structure instances. Initial values for the properties of the new instance can be passed to the memberwise initializer by name:

```
let vga = Resolution(width: 640, height: 480)
```

Unlike structures, class instances do not receive a default memberwise initializer. Initializers are described in more detail in Initialization.

Structures and Enumerations Are Value Types

A *value type* is a type whose value is *copied* when it is assigned to a variable or constant, or when it is passed to a function.

You've actually been using value types extensively throughout the previous chapters. In fact, all of the basic types in Swift—integers, floating-point numbers, Booleans, strings, arrays and dictionaries—are value types, and are implemented as structures behind the scenes.

All structures and enumerations are value types in Swift. This means that any structure and enumeration instances you create—and any value types they have as properties—are always copied when they are passed around in your code.

Consider this example, which uses the `Resolution` structure from the previous example:

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

This example declares a constant called `hd` and sets it to a `Resolution` instance initialized with the width and height of full HD video (`1920` pixels wide by `1080` pixels high).

It then declares a variable called `cinema` and sets it to the current value of `hd`. Because `Resolution` is a structure, a *copy* of the existing instance is made, and this new copy is assigned to `cinema`. Even though `hd` and `cinema` now have the same width and height, they are two completely different instances behind the scenes.

Next, the `width` property of `cinema` is amended to be the width of the slightly-wider 2K standard used for digital cinema projection (`2048` pixels wide and `1080` pixels high):

```
cinema.width = 2048
```

Checking the `width` property of `cinema` shows that it has indeed changed to be `2048`:

```
print("cinema is now \(cinema.width) pixels wide")
// Prints "cinema is now 2048 pixels wide"
```

However, the `width` property of the original `hd` instance still has the old value of `1920`:

```
print("hd is still \(hd.width) pixels wide")
// Prints "hd is still 1920 pixels wide"
```

When `cinema` was given the current value of `hd`, the *values* stored in `hd` were copied into the new `cinema` instance. The end result is two completely separate instances, which just happened to contain the same numeric values. Because they are separate instances, setting the width of `cinema` to `2048` doesn't affect the width stored in `hd`.

The same behavior applies to enumerations:

```
enum CompassPoint {
    case north, south, east, west
}
var currentDirection = CompassPoint.west
let rememberedDirection = currentDirection
currentDirection = .east
if rememberedDirection == .west {
    print("The remembered direction is still .west")
}
// Prints "The remembered direction is still .west"
```

When `rememberedDirection` is assigned the value of `currentDirection`, it is actually set to a copy of that value. Changing the value of `currentDirection` thereafter does not affect the copy of the original value that was stored in `rememberedDirection`.

Classes Are Reference Types

Unlike value types, *reference types* are *not* copied when they are assigned to a variable or constant, or when they are passed to a function. Rather than a copy, a reference to the same existing instance is used instead.

Here's an example, using the `VideoMode` class defined above:

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

This example declares a new constant called `tenEighty` and sets it to refer to a new instance of the `VideoMode` class. The video mode is assigned a copy of the HD resolution of `1920` by `1080` from before. It is set to be interlaced, and is given a name of `"1080i"`. Finally, it is set to a frame rate of `25.0` frames per second.

Next, `tenEighty` is assigned to a new constant, called `alsoTenEighty`, and the frame rate of `alsoTenEighty` is modified:

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

Because classes are reference types, `tenEighty` and `alsoTenEighty` actually both refer to the *same* `VideoMode` instance. Effectively, they are just two different names for the same single instance.

Checking the `frameRate` property of `tenEighty` shows that it correctly reports the new frame rate of `30.0` from the underlying `VideoMode` instance:

```
print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// Prints "The frameRate property of tenEighty is now 30.0"
```

Note that `tenEighty` and `alsoTenEighty` are declared as *constants*, rather than variables. However, you can still change `tenEighty.frameRate` and `alsoTenEighty.frameRate` because the values of the `tenEighty` and `alsoTenEighty` constants themselves do not actually change. `tenEighty` and `alsoTenEighty` themselves do not "store" the `VideoMode` instance—instead, they both *refer* to a `VideoMode` instance behind the scenes. It is

the `frameRate` property of the underlying `VideoMode` that is changed, not the values of the constant references to that `VideoMode`.

Identity Operators

Because classes are reference types, it is possible for multiple constants and variables to refer to the same single instance of a class behind the scenes. (The same is not true for structures and enumerations, because they are always copied when they are assigned to a constant or variable, or passed to a function.)

It can sometimes be useful to find out if two constants or variables refer to exactly the same instance of a class. To enable this, Swift provides two identity operators:

- Identical to (`===`)
- Not identical to (`!==`)

Use these operators to check whether two constants or variables refer to the same single instance:

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same VideoMode instance.")
}
// Prints "tenEighty and alsoTenEighty refer to the same VideoMode instance."
```

Note that “identical to” (represented by three equals signs, or `===`) does not mean the same thing as “equal to” (represented by two equals signs, or `==`):

- “Identical to” means that two constants or variables of class type refer to exactly the same class instance.
- “Equal to” means that two instances are considered “equal” or “equivalent” in value, for some appropriate meaning of “equal”, as defined by the type’s designer.

When you define your own custom classes and structures, it is your responsibility to decide what qualifies as two instances being “equal”. The process of defining your own implementations of the “equal to” and “not equal to” operators is described in Equivalence Operators.

Pointers

If you have experience with C, C++, or Objective-C, you may know that these languages use *pointers* to refer to addresses in memory. A Swift constant or variable that refers to an instance of some reference type is similar to a pointer in C, but is not a direct pointer to an address in memory, and does not require you to write an asterisk (`*`) to indicate that you are creating a reference. Instead, these references are defined like any other constant or variable in Swift.

Choosing Between Classes and Structures

You can use both classes and structures to define custom data types to use as the building blocks of your program’s code.

However, structure instances are always passed by *value*, and class instances are always passed by *reference*. This means that they are suited to different kinds of tasks. As you consider the data constructs and functionality that you need for a project, decide whether each data construct should be defined as a class or as a structure.

As a general guideline, consider creating a structure when one or more of these conditions apply:

- The structure’s primary purpose is to encapsulate a few relatively simple data values.
- It is reasonable to expect that the encapsulated values will be copied rather than referenced when you assign or pass around an instance of that structure.
- Any properties stored by the structure are themselves value types, which would also be expected to be copied rather than referenced.
- The structure does not need to inherit properties or behavior from another existing type.

Examples of good candidates for structures include:

- The size of a geometric shape, perhaps encapsulating a `width` property and a `height` property, both of type `Double`.
- A way to refer to ranges within a series, perhaps encapsulating a `start` property and a `length` property, both of type `Int`.
- A point in a 3D coordinate system, perhaps encapsulating `x`, `y` and `z` properties, each of type `Double`.

In all other cases, define a class, and create instances of that class to be managed and passed by reference. In practice, this means that most custom data constructs should be classes, not structures.

Assignment and Copy Behavior for Strings, Arrays, and Dictionaries

In Swift, many basic data types such as `String`, `Array`, and `Dictionary` are implemented as structures. This means that data such as strings, arrays, and dictionaries are copied when they are assigned to a new constant or variable, or when they are passed to a function or method.

This behavior is different from Foundation: `NSString`, `NSArray`, and `NSDictionary` are implemented as classes, not structures. Strings, arrays, and dictionaries in Foundation are always assigned and passed around as a reference to an existing instance, rather than as a copy.

> NOTE
>
> The description above refers to the “copying” of strings, arrays, and dictionaries. The behavior you see in your code will always be as if a copy took place. However, Swift only performs an *actual* copy behind the scenes when it is absolutely necessary to do so. Swift manages all value copying to ensure optimal performance, and you should not avoid assignment to try to preempt this optimization.