Structures and Classes

Structures and classes 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 structures and classes using the same syntax you use to define constants, variables, and functions.

An instance of a class is traditionally known as an *object*.

Comparing Structures and Classes

Structures and classes 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

Classes have additional capabilities that structures don't have:

- 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.

As a general guideline, prefer structures because they're easier to reason about, and use classes when they're appropriate or necessary.

Definition Syntax:

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

Whenever you define a new structure or class, you define a new Swift type. Give types **UpperCamelCase** names (such as **SomeStructure** and **SomeClass** here) to match the capitalization of standard Swift types (such as **String**, **Int**, and **Bool**).

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

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

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"
```

You can drill down into subproperties, 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"
```

Classes Are Reference Types

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

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's set to be interlaced, its name is set to "1080i", and its frame rate is set to 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're just two different names for the same single instance, as shown in the figure below: