Sun*

Structures & Classes

Outline

1. General
2. Comparing Structures and Classes
3. Structures and Enumerations Are Value Types
4. Classes Are Reference Types

1. General

- *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.

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

2. Comparing Structures and Classes

1. Definition Syntax
2. Creating Instances
3. Accessing Properties
4. Memberwise Initializers for Structure Types

2.1 Definition Syntax

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

```
struct Resolution {
    var width = 0
    var height = 0
}

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

2.2 Creating Instances

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

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

2.3 Accessing Properties

- You can access the properties of an instance using *dot syntax*:

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

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"
```

2.4 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: 1920, height: 1080)
```

- Unlike structures, class instances don’t receive a default memberwise initializer.

3. Structures and Enumerations Are Value Types

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

- All basic types in Swift - integers, floating-point numbers, Booleans, strings, arrays, dictionaries - are value types, and are implemented as structures behind the scenes.

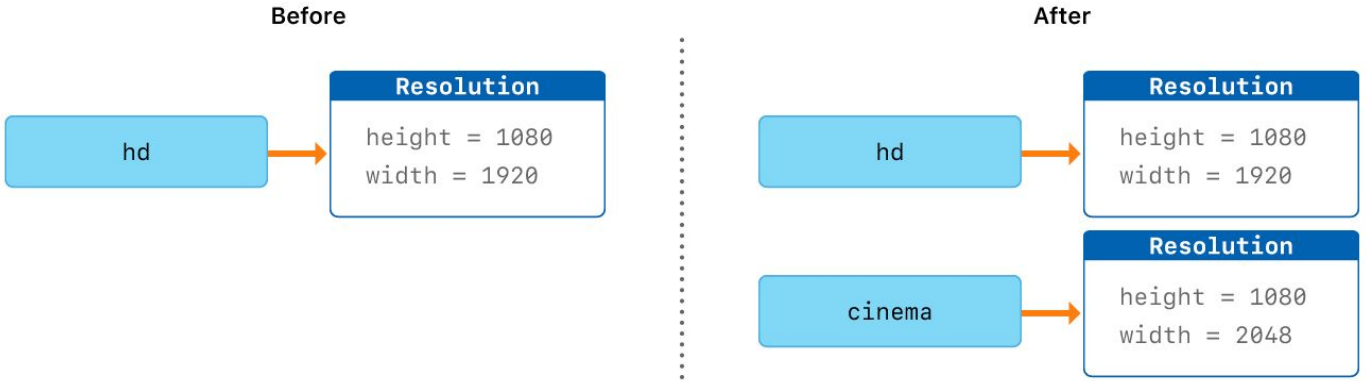
- All structures and enumerations are value types in Swift.

3. Structures and Enumerations Are Value Types (cont)

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

cinema.width = 2048

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

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

3. Structures and Enumerations Are Value Types (cont)

- When `cinema` was given the current value of `hd`, the values stored in `hd` were copied into the new `cinema` instance. The end result was two completely separate instances that contained the same numeric values. However, because they are separate instances, setting the width of `cinema` to 2048 doesn't affect the width stored in `hd`, as shown in the figure below:



4. 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.

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

let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0

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

4. Classes Are Reference Types (cont)

- 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, as shown in the figure below:

4. Classes Are Reference Types (cont)

- Because classes are reference types, it's possible for multiple constants and variables to refer to the same single instance of a class behind the scenes.

- It can sometimes be useful to find out whether 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 (!==)

4. Classes Are Reference Types (cont)

➔ 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."
```

Question & Answer?