Class & Structure

-Devendranath

For Offline / Online Training, reach me@ iPhoneDev1990@gmail.com

- **Classes** and Structures are basic building blocks to create custom datatypes
- Swift's structures have **properties** and **methods** as classes
- Unlike other programming languages, Swift does not require you to create separate interface and implementation files for custom classes

Classes vs 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

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. Structures doesn't have Deinitializers.
- **Reference counting** allows more than one reference to a class instance.
- Structures are value types whereas classes are reference types.

Syntaxes

```
class SomeClass
{
   // Properties (instance variables)
  // Methods
}

struct SomeStructure {
   // Properties (instance variables)
  // Methods
}
```

```
struct Resolution {
    var width = 0
    var height = 0
    func description()
    {
        print("width :\(width) and height: \(height)")
    }
}
```

```
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
// CREATING VARIABLES FOR STRUCTURES AND CLASSES
let someResolution = Resolution(width:100, height:200)
let someVideoMode = VideoMode()
```

Accessing structure variables:

Syntax:

structureVariable.structureProperty = value

someResolution.width = 1024

someResolution.height = 768

Example

```
struct MyCustomDatatype {
    var aIntVar = 0;
    var aCharVar: Character?;
    var aStringVar: String?;

    func aStrMethod()
    {
        print("This is structure method");
    }
}
```

```
let aStrVar = MyCustomDatatype(aIntVar: 10, aCharVar:
"C", aStringVar: "This is Structure");

print(aStrVar);
aStrVar.aStrMethod();
```

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 behaviour from another existing type.**

Structures with Custom Initilizers

```
struct Books
{
    var pages: Int
    var name: String = ""

    init(p: Int, n: String)
    {
        pages = p;
        name = n
    }

    init()
    {
        pages = 200;
        name = "Obj-c"
    }
}

let bBook = Books(p: 100, n: "Obj-C")
let cBook = Books()

print(bBook.pages) // 100
print(cBook.pages) // 200
```

NOTE: Structures doesn’t contain Deinitilizers

Mutating Methods

By default Swift's structure doesn't allow modifying the properties in instance methods. To modify the properties in the methods, we need to use the mutating keyword in-front of that method.

```
struct TV {
    var brand: String
    let model: Int

    mutating func changeBrand(newBrand: String) {
        brand = newBrand
    }

    func displayTvInfo() {
        print(brand)
        print(model)
    }
}
```

Thank You

For Offline / Online Training, reach me@ iPhoneDev1990@gmail.com