Mobile Application Programming

Swift Classes

Swift Top-Level Entities

- Like C/C++ but unlike Java, Swift allows declarations of functions, variables, and constants at the top-level, outside any class declaration
- Constants are declared using the let keyword
- Variables are declared using the var keyword
- Functions are declared using the func keyword with parameter names interleaved with the name of the function, causing it to read like a sentence

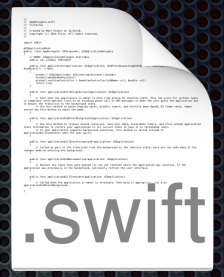
Swift Objects

- Classes, structures, and enums are all object types with different defaults in usage
  - Classes are reference types that share the same object when assignments are made
  - Structs are always copied on assignment
- Single inheritance, but may conform to many protocols
- Add functions and protocols to existing objects using *extension* keyword. Also used to break up large objects

Swift Classes

- Member functions and properties declared using same syntax as top-level declarations
- Function declarations use parameter labels, but the first label is omitted when declared in a class
- Properties declare both getter / setter and a (hidden) backing variable using *var* and *let* keywords
- Use *private*, *fileprivate*, *internal* (default), *public,* and *open* for access control
- Constructors are declared using *init*(), but have different inheritance rules than most languages

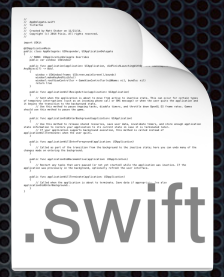
Properties



- Properties for class instances are declared using *var* or *let*
- Access properties using *self* or the name directly when unambiguous
- External access to the properties is defined using *private*, *fileprivate*, *internal* (default), *public*, or *open*

```
import Point
import Vector

class Car {
   private var _vin: String = “FAST”
   private var _year: Int = 1970
   private var _position: Point = Point()
   private var _velocity: Vector = Vector()

   var velocity: Vector  {
      get { return _velocity }
      set { _velocity = newValue }
   }

   var vin: String { return _vin }
   var year: Int { return _year }

   func moveByInterval(interval: Double) {
      _position += _velocity * interval
   }
}
```

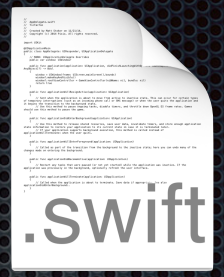
Stored Properties



- Properties that are given a value at declaration or during initialization are called stored properties
- These have a hidden backing store allocated for each instance as well as get and set methods
- Observe property changes using *willSet* and *didSet*

```
import Point
import Vector

class Car {
    private var _vin: String = “FAST”
    private var _year: Int = 1970
    private var _position: Point = Point()
    private var _velocity: Vector = Vector()

    var velocity: Vector  {
        get { return _velocity }
        set { _velocity = newValue }
    }

    var vin: String { return _vin }
    var year: Int { return _year }

    func moveByInterval(interval: Double) {
        _position += _velocity * interval
    }
}
```

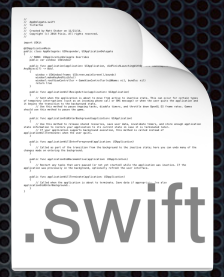
Computed Properties



- Properties with explicit get and set methods define computed properties
- If no set method is provided the property is read only (get can be omitted in this case)
- These have no backing store and act like named methods

```
import Point
import Vector

class Car {
   private var _vin: String = “FAST”
   private var _year: Int = 1970
   private var _position: Point = Point()
   private var _velocity: Vector = Vector()

   var velocity: Vector  {
      get { return _velocity }
      set { _velocity = newValue }
   }

   var vin: String { return _vin }
   var year: Int { return _year }

   func moveByInterval(interval: Double) {
      _position += _velocity * interval
   }
}
```

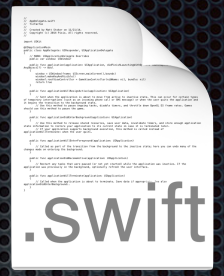
Methods



- Methods are declared using *func* like top-level functions
- Parameters should have labels so the method reads like a sentence
- The first parameter should have no label. Instead name the method with the first part of the sentence

```
import Point
import Vector

class Car {
   private var _vin: String = “FAST”
   private var _year: Int = 1970
   private var _position: Point = Point()
   private var _velocity: Vector = Vector()

   var velocity: Vector  {
      get { return _velocity }
      set { _velocity = newValue }
   }

   var vin: String { return _vin }
   var year: Int { return _year }

   func moveByInterval(interval: Double) {
      _position += _velocity * interval
   }
}
```

Swift Classes




```
class Car
{
   private var _vin: String
   private var _year: Int
   private var _position: Point  // Imported
   private var _velocity: Vector // Imported

   init(vin: String, year: Int)
   {
      _vin = vin
      _year = year
      _position = Point(x: 0.0, y: 0.0)
      _velocity = Vector(x: 0.0, y: 0.0)
   }

   var vin: String
   {
      return _vin
   }

   var year: Int
   {
      return _year
   }
```

```
   var position: Point
   {
      get { return _position }
      set { _position = newValue }
   }

   var velocity: Vector
   {
      get { return _velocity }
      set { _velocity = newValue }
   }

   func moveWithTime(elapsedTime: Double)
   {
      _position += _velocity * elapsedTime
   }
}

// Usage
var viper: Car = Car(“23958060934985”, 2003)
viper.position = Point(x: 40.76, y: -113.93)
viper.velocity = Vector(x: 100.0, y: 200.0)
viper.moveWithTime(1.2) //Note label omitted
```

Initializers

- Use *init* keyword to define a designated initializer
- Must ensure all properties of class have a value
- A default initializer is created if all properties have a default value
- Properties must be set before calling a super class designated initializer

```
// Root Class
class Car
{
   init(vin: String, year: Int)
   {
      _vin = vin
      _year = year
      _position = Point(x: 0.0, y: 0.0)
      _velocity = Vector(x: 0.0, y: 0.0)
   }
   // Rest of class…
}
```

```
// Inheriting Class
class RocketCar: Car
{
   private var _fuel: Double
   init(fuel: Double)
   {
      _fuel = fuel
      super.init(vin: “FAST”, year: 2020)
   }
   var fuel: Double {
      get { return _fuel }
      set { _fuel = newValue }
   }
}
```

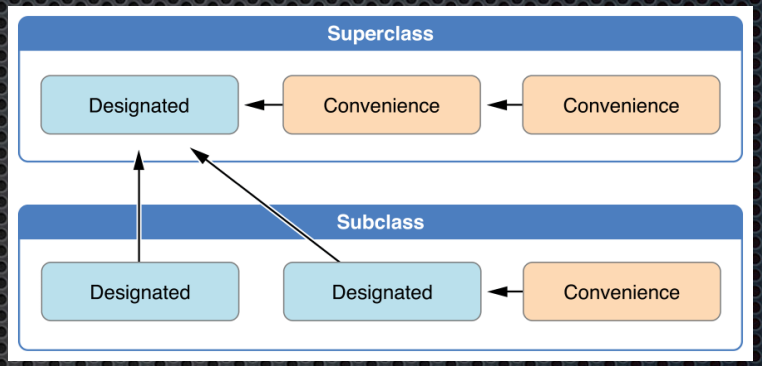
Convenience Constructors

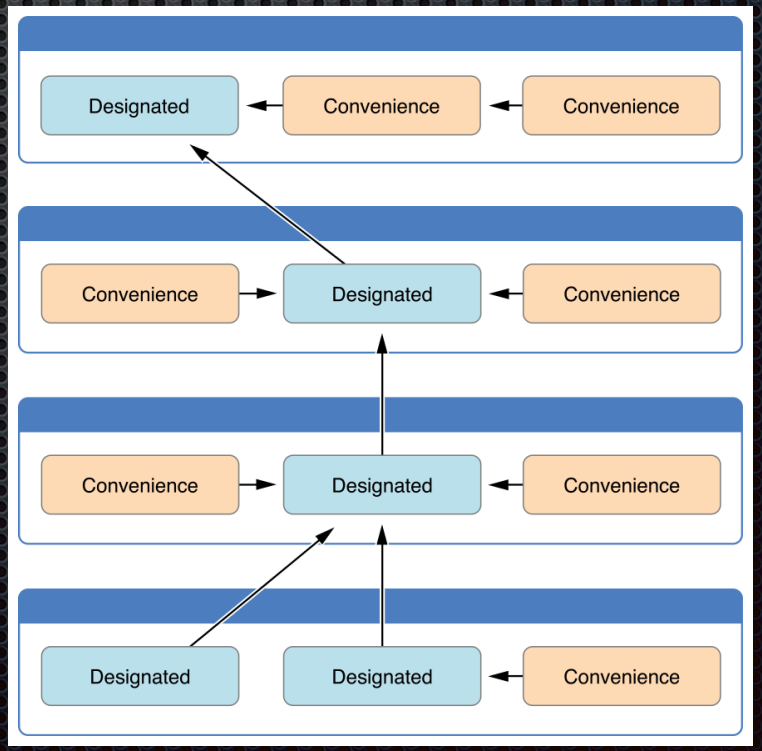
- Convenience constructors prevent duplicating code by calling designated initializers to fill in some or all properties
- Must call another initializer at this class level
- Not inherited by sub-classes

```
class RocketCar: Car
{
   private var _fuel: Double
   init(fuel: Double)
   {
      _fuel = fuel
      super.init(vin: “FAST”, year: 2020)
   }

   convenience init()
   {
      self.init(fuel: 100.0)
   }

   var fuel: Double {
      get { return _fuel }
      set { _fuel = newValue }
   }
}
```

Inheritance & 2-Phase Init

- Only designated initializers are inherited by subclasses
- They can be overridden using the override keyword
- Because sub-classes call super-class designated initializers, there are rules for property initialization order (2-Phase Initialization)
- See the reading for class





Other Features

- Deinitialization using the deinit method
- Class extensions using the extension keyword
- Protocol support by defining protocols using protocol then adding them to the inheritance list for the class
- Automatic Reference Counting for memory management, controlled using weak and unowned
- Operator overloading and subscripts
- Generic object support similar to java