Inheritance

Outline

1. Overview
2. Defining a Base Class
3. Subclassing
4. Overriding

1. Overview

- A class can *inherit* methods, properties, and other characteristics from another class. When one class inherits from another, the inheriting class is known as a *subclass*, and the class it inherits from is known as its *superclass*.

- Classes in Swift can call and access methods, properties, and subscripts belonging to their superclass and can provide their own overriding versions of those methods, properties, and subscripts to refine or modify their behavior.

- Classes can also add property observers to inherited properties in order to be notified when the value of a property changes. Property observers can be added to any property, regardless of whether it was originally defined as a stored or computed property.

2. Defining a Base Class

- Any class that does not inherit from another class is known as a base class.

- Swift classes do not inherit from a universal base class. Classes you define without specifying a superclass automatically become base classes for you to build upon.

```swift
class Vehicle {
  var currentSpeed = 0.0
  var description: String {
    return "traveling at \(currentSpeed) miles per hour"
  }

  func makeNoise() {}
}

let someVehicle = Vehicle()
print("Vehicle: \(someVehicle.description)")
// Vehicle: traveling at 0.0 miles per hour
```

3. Subclassing

- Subclassing is the act of basing a new class on an existing class. The subclass inherits characteristics from the existing class, which you can then refine.

- You can also add new characteristics to the subclass.

```
class Bicycle: Vehicle {
    var hasBasket = false
}

let bicycle = Bicycle()
bicycle.hasBasket = true

bicycle.currentSpeed = 15.0
print("Bicycle: \(bicycle.description)")
// Bicycle: traveling at 15.0 miles per hour
```

4. Overriding

1. General
2. Accessing Superclass Methods, Properties, and Subscripts
3. Overriding Methods
4. Overriding Property Getters and Setters
5. Overriding Property Observers
6. Preventing Overrides

4.1 General

- A subclass can provide its own custom implementation of an instance method, type method, instance property, type property, or subscript that it would otherwise inherit from a superclass. This is known as *overriding*.

- To override a characteristic that would otherwise be inherited, you prefix your overriding definition with the **`override`** keyword.

- The **`override`** keyword also prompts the Swift compiler to check that your overriding class's superclass (or one of its parents) has a declaration that matches the one you provided for the override. This check ensures that your overriding definition is correct.

4.2 Accessing Superclass Methods, Properties, and Subscripts

- When you provide a method, property, or subscript override for a subclass, it is sometimes useful to use the existing superclass implementation as part of your override. Where this is appropriate, you access them by using the **`super`** prefix:
  - An overridden method named **`someMethod()`** can call the superclass version of **`someMethod()`** by calling **`super`**`.someMethod()` within the overriding method implementation.
  - An overridden property called **`someProperty`** can access the superclass version of **`someProperty`** as **`super`**`.someProperty` within the overriding getter or setter implementation.
  - An overridden subscript for **`someIndex`** can access the superclass version of the same subscript as **`super`**`[someIndex]` from within the overriding subscript implementation.

4.3 Overriding Methods

- You can override an inherited instance or type method to provide a tailored or alternative implementation of the method within your subclass.

```
class Train: Vehicle {
  override func makeNoise() {
  print("Choo Choo")
  }
}

let train = Train()
train.makeNoise()
// Prints "Choo Choo"
```

4.4 Overriding Properties Getters & Setters

- You can provide a custom getter (and setter, if appropriate) to override any inherited property, regardless of whether the inherited property is implemented as a stored or computed property at source.

- You can present an inherited read-only property as a read-write property by providing both a getter and a setter in your subclass property override. You cannot, however, present an inherited read-write property as a read-only property.

4.4 Overriding Properties Getters & Setters

- Example:

```
class Car: Vehicle {
  var gear = 1
  override var description: String {
    return super.description + " in gear \(gear)"
  }
}

let car = Car()
car.currentSpeed = 25.0
car.gear = 3
print("Car: \(car.description)")
// Car: traveling at 25.0 miles per hour in gear 3
```

4.5 Overriding Properties Observers

- You can use property overriding to add property observers to an inherited property. This enables you to be notified when the value of an inherited property changes, regardless of how that property was originally implemented.

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}

let automatic = AutomaticCar()
automatic.currentSpeed = 35.0
print("AutomaticCar: \(automatic.description)")
// AutomaticCar: traveling at 35.0 miles per hour in gear 4
```

4.6 Preventing Overrides

- You can prevent a method, property, or subscript from being overridden by marking it as *final*. Do this by writing the **`final`** modifier before the method, property, or subscript's introducer keyword (such as **`final var`**, **`final func`**, **`final class func`**, and **`final subscript`**).

- Any attempt to override a final method, property, or subscript in a subclass is reported as a compile-time error. Methods, properties, or subscripts that you add to a class in an extension can also be marked as final within the extension's definition.

- You can mark an entire class as final by writing the **`final`** modifier before the **`class`** keyword in its class definition (**`final class`**). Any attempt to subclass a final class is reported as a compile-time error.

Question & Answer?