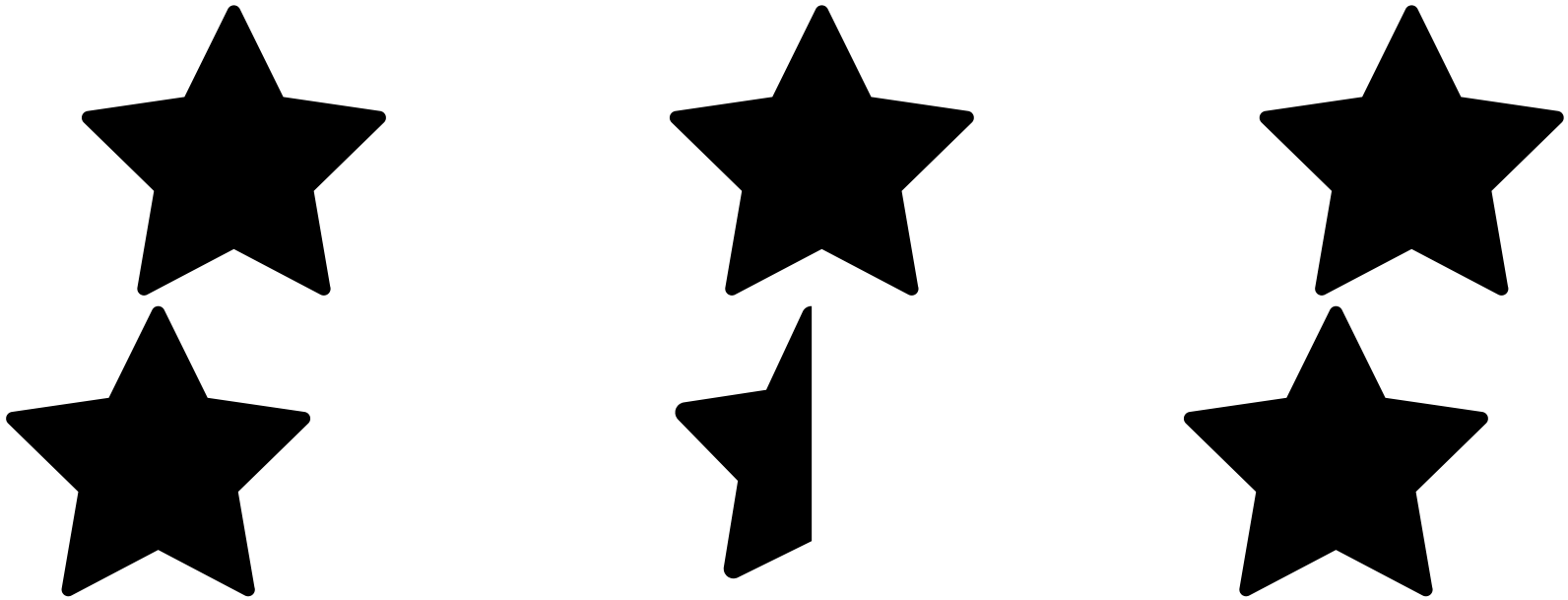
Home · iOS & Swift Tutorials

Object Oriented Programming in Swift

Learn how object oriented programming works in Swift by breaking things down into objects that can be inherited and composed from.

By Cosmin Pupăză May 24 2017 · Article (25 mins) · Beginner

4.4/5



16

Ratings

Object oriented programming is a fundamental programming paradigm that you must master if you are serious about learning Swift. That's because object oriented programming is at the heart of most frameworks you'll be working with. Breaking a problem down into objects that send *messages* to one another might seem strange at first, but it's a proven approach for simplifying complex systems, which dates back to the 1950s.

Objects can be used to model almost anything — coordinates on a map, touches on a screen, even fluctuating interest rates in a bank account. When you're just starting out, it's useful to practice modeling physical things in the real world before you extend this to more abstract concepts.

In this tutorial, you'll use object oriented programming to create your own band of musical instruments. You'll also learn many important concepts along the way including:

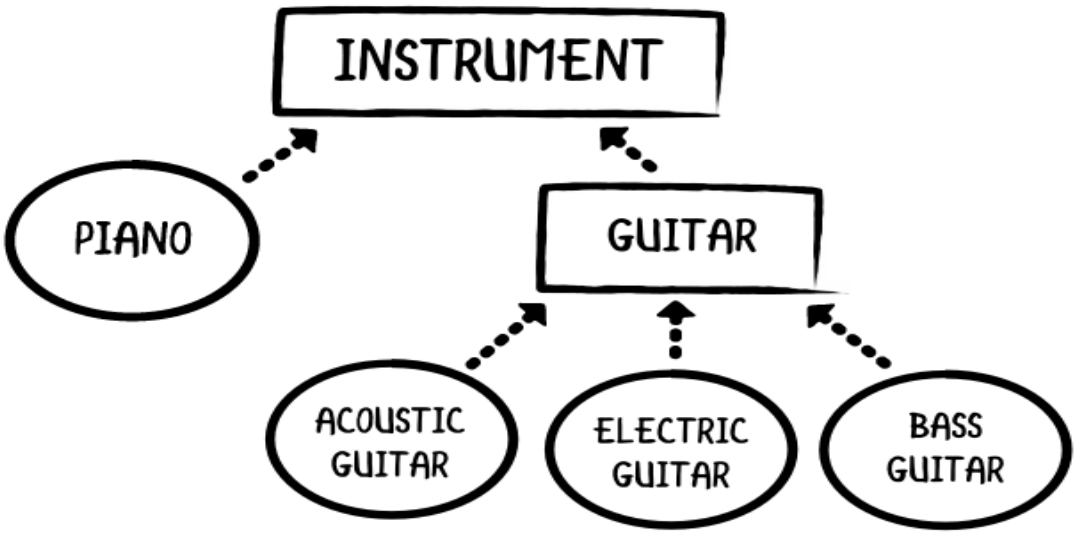
Encapsulation
Inheritance
Overriding versus *Overloading*
Types versus *Instances*
Composition
Polymorphism
Access Control

That's a lot, so let's get started! :]

Getting Started

Fire up Xcode and go to *File\New\Playground...*. Type *Instruments* for *Name*, select *iOS* for *Platform* and click *Next*. Choose where to save your playground and click *Create*. Delete everything from it in order to start from scratch.

Designing things in an object-oriented manner usually begins with a general concept extending to more specific types. You want to create musical instruments, so it makes perfect sense to begin with an instrument type and then define concrete (not literally!) instruments such as pianos and guitars from it. Think of the whole thing as a *family tree* of instruments where everything flows from general to specific and top to bottom like this:



The relationship between a child type and its parent type is an *is-a* relationship. For example, "Guitar is-a Instrument." Now that you have a visual understanding of the objects you are dealing with, it's time to start implementing.

Properties

Add the following block of code at the top of the playground:

```
// 1
class Instrument {
```

```
  // 2
  let brand: String
  // 3
  init(brand: String) {
    //4
    self.brand = brand
  }
}
```

There's quite a lot going on here, so let's break it down:

. You create the `Instrument` *base class* with the `class` keyword. This is the *root class* of the instruments hierarchy. It defines a blueprint which forms the basis of any kind of instrument. Because it's a type, the name `Instrument` is capitalized. It doesn't *have* to be capitalized, however this is the convention in Swift.

. You declare the instrument's *stored properties* (data) that *all* instruments have. In this case, it's just the brand, which you represent as a `String`.

. You create an *initializer* for the class with the `init` keyword. Its purpose is to construct new instruments by initializing all stored properties.

. You set the instrument's `brand` stored property to what was passed in as a parameter. Since the property and the parameter have the same name, you use the `self` keyword to distinguish between them.

You've implemented a `class` for instruments containing a `brand` property, but you haven't given it any behavior yet. Time to add some behavior in the form of methods to the mix.

Methods

You can tune and play an instrument regardless of its particular type. Add the following code inside the `Instrument` class right after the initializer:

```
func tune() -> String {
  fatalError("Implement this method for \(brand)")
}
```

The `tune()` *method* is a placeholder function that crashes at runtime if you call it. Classes with methods like this are said to be *abstract* because they are not intended for direct use. Instead, you must define a *subclass* that *overrides* the method to do something sensible instead of only calling `fatalError()`. More on overriding later.

Functions defined inside a `class` are called methods because they have access to properties, such as `brand` in the case of `Instrument`. Organizing properties and related operations in a `class` is a powerful tool for taming complexity. It even has a fancy name: *encapsulation*. Class types are said to encapsulate data (e.g. stored properties) and behavior (e.g. methods).
Next, add the following code before your `Instrument` class:

```
class Music {
  let notes: [String]

  init(notes: [String]) {
    self.notes = notes
  }

  func prepared() -> String {
    return notes.joined(separator: " ")
  }
}
```

This is a `Music` class that encapsulates an array of notes and allows you to flatten it into a string with the `prepared()` method.
Add the following method to the `Instrument` class right after the `tune()` method:

```
func play(_ music: Music) -> String {
  return music.prepared()
}
```

The `play(_:)` method returns a `String` to be played. You might wonder why you would bother creating a special `Music` type, instead of just passing along a `String` array of notes. This provides several advantages: Creating `Music` helps build a vocabulary, enables the compiler to check your work, and creates a place for future expansion.
Next, add the following method to the `Instrument` class right after `play(_:)`:

```
func perform(_ music: Music) {
  print(tune())
  print(play(music))
}
```

The `perform(_:)` method first tunes the instrument and then plays the music given in one go. You've composed two of your methods together to work in perfect symphony. (Puns very much intended! :])
That's it as far as the `Instrument` class implementation goes. Time to add some specific instruments now.

Inheritance

Add the following class declaration at the bottom of the playground, right after the `Instrument` class implementation:

```
// 1
class Piano: Instrument {
  let hasPedals: Bool
  // 2
  static let whiteKeys = 52
  static let blackKeys = 36

  // 3
  init(brand: String, hasPedals: Bool = false) {
    self.hasPedals = hasPedals
    // 4
    super.init(brand: brand)
  }

  // 5
  override func tune() -> String {
    return "Piano standard tuning for \(brand)."
  }

  override func play(_ music: Music) -> String {
    // 6
    let preparedNotes = super.play(music)
    return "Piano playing \(preparedNotes)"
  }
}
```

Here's what's going on, step by step:

. You create the `Piano` class as a *subclass* of the `Instrument` *parent class*. All the stored properties and methods are automatically *inherited* by the `Piano` *child class* and available for use.

. All pianos have *exactly* the same number of white and black keys regardless of their brand. The associated values of their corresponding properties *don't* change dynamically, so you mark the properties as `static` in order to reflect this.

. The initializer provides a default value for its `hasPedals` parameter which allows you to leave it off if you want.

. You use the `super` keyword to call the parent class initializer after setting the child class stored property `hasPedals`. The super class initializer takes care of initializing inherited properties — in this case, `brand`.

. You *override* the inherited `tune()` method's implementation with the `override` keyword. This provides an implementation of `tune()` that doesn't call `fatalError()`, but rather does something specific to `Piano`.

. You override the inherited `play(_:)` method. And inside this method, you use the `super` keyword this time to call the `Instrument` parent method in order to get the music's prepared notes and then play on the piano.

Because `Piano` derives from `Instrument`, users of your code already know a lot about it: It has a brand, it can be tuned, played, and can even be performed.

Note: Swift classes use an initialization process called two-phase-initialization to guarantee that all properties are initialized before you use them. If you want to learn more about initialization, check out our tutorial series on Swift initialization.

The piano tunes and plays accordingly, but you can play it in different ways. Therefore, it's time to add pedals to the mix.

Method Overloading

Add the following method to the `Piano` class right after the overridden `play(_:)` method:

```
func play(_ music: Music, usingPedals: Bool) -> String {
  let preparedNotes = super.play(music)
  if hasPedals && usingPedals {
    return "Play piano notes \(preparedNotes) with pedals."
  }
  else {
    return "Play piano notes \(preparedNotes) without pedals."
  }
}
```

This *overloads* the `play(_:)` method to use pedals if `usingPedals` is true and the piano actually has pedals to use. It does not use the `override` keyword because it has a different *parameter list*. Swift uses the parameter list (aka *signature*) to determine which to use. You need to be careful with overloaded methods though because they have the potential to cause confusion. For example, the `perform(_:)` method always calls the `play(_:)` one, and will never call your specialized `play(_:usingPedals:)` one.
Replace `play(_:)`, in `Piano`, with an implementation that calls your new pedal using version:

```
override func play(_ music: Music) -> String {
  return play(music, usingPedals: hasPedals)
}
```

That's it for the `Piano` class implementation. Time to create an actual piano instance, tune it and play some really cool music on it. :]

Instances

Add the following block of code at the end of the playground right after the `Piano` class declaration:

```
// 1
let piano = Piano(brand: "Yamaha", hasPedals: true)
piano.tune()
// 2
let music = Music(notes: ["C", "G", "F"])
piano.play(music, usingPedals: false)
// 3
piano.play(music)
// 4
Piano.whiteKeys
Piano.blackKeys
```

This is what's going on here, step by step:

. You create a `piano` as an *instance* of the `Piano` class and `tune` it. Note that while types (classes) are always capitalized, instances are always all lowercase. Again, that's convention.

. You declare a `music` instance of the `Music` class any play it on the piano with your special overload that lets you play the song without using the pedals.

. You call the `Piano` class version of `play(_:)` that always uses the pedals if it can.

. The key counts are `static` constant values inside the `Piano` class, so you don't need a specific *instance* to call them — you just use the class name prefix instead.

Now that you've got a taste of piano music, you can add some guitar solos to the mix.

Intermediate Abstract Base Classes

Add the `Guitar` class implementation at the end of the playground:

```
class Guitar: Instrument {
  let stringGauge: String

  init(brand: String, stringGauge: String = "medium") {
    self.stringGauge = stringGauge
    super.init(brand: brand)
  }
}
```

This creates a new class `Guitar` that adds the idea of string gauge as a text `String` to the `Instrument` base class. Like `Instrument`, `Guitar` is considered an abstract type whose `tune()` and `play(_:)` methods need to be overridden in a subclass. This is why it is sometimes called a *intermediate abstract base class*.

Note: You will notice that there's nothing stopping you creating an instance of a class that is abstract. This is true, and is a limitation to Swift. Some languages allow you to specifically state that a class is abstract and that you can't create an instance of it.

That's it for the `Guitar` class – you can add some really cool guitars now! Let's do it! :]

Concrete Guitars

The first type of guitar you are going to create is an acoustic. Add the `AcousticGuitar` class to the end of the playground right after your `Guitar` class:

```
class AcousticGuitar: Guitar {
  static let numberOfStrings = 6
  static let fretCount = 20

  override func tune() -> String {
    return "Tune \(brand) acoustic with E A D G B E"
  }
```

```
  override func play(_ music: Music) -> String {
    let preparedNotes = super.play(music)
    return "Play folk tune on frets \(preparedNotes)."
  }
}
```

All acoustic guitars have 6 strings and 20 frets, so you model the corresponding properties as *static* because they relate to all acoustic guitars. And they are *constants* since their values *never* change over time. The class doesn't add any new stored properties of its own, so you don't need to create an initializer, as it automatically inherits the initializer from its parent class, `Guitar`. Time to test out the guitar with a challenge!

Challenge: Define a Roland-brand acoustic guitar. Tune, and play it.

Add the following test code at the bottom of the playground right after the `AcousticGuitar` class declaration:

```
let acousticGuitar = AcousticGuitar(brand: "Roland", stringGauge: "light")
acousticGuitar.tune()
acousticGuitar.play(music)
```

Reveal

It's time to make some noise and play some loud music. You will need an amplifier! :]

Private

Acoustic guitars are great, but amplified ones are even cooler. Add the `Amplifier` class at the bottom of the playground to get the party started:

```
// 1
class Amplifier {
  // 2
  private var _volume: Int
  // 3
  private(set) var isOn: Bool

  init() {
    isOn = false
    _volume = 0
```

```
    }

    // 4
    func plugIn() {
      isOn = true
    }

    func unplug() {
      isOn = false
    }

    // 5
    var volume: Int {
      // 6
      get {
        return isOn ? _volume : 0
      }
      // 7
      set {
        _volume = min(max(newValue, 0), 10)
      }
    }
  }
```

There's quite a bit going on here, so lets break it down:

. You define the `Amplifier` class. This is also a root class, just like `Instrument`.

. The stored property `_volume` is marked `private` so that it can only be accessed inside of the `Amplifier` class and is hidden away from outside users. The underscore at the beginning of the name emphasizes that it is a private implementation detail. Once again, this is merely a convention. But it's good to follow conventions. :]

. The stored property `isOn` can be read by outside users but not written to. This is done with `private(set)`.

. `plugIn()` and `unplug()` affect the state of `isOn`.

. The *computed property* named `volume` wraps the private stored property `_volume`.

. The *getter* drops the volume to 0 if it's not plugged in.

. The volume will always be clamped to a certain value between 0 and 10 inside the *setter*. No setting the amp to 11.
The access control keyword `private` is extremely useful for hiding away complexity and protecting your class from invalid modifications. The fancy name for this is "protecting the invariant". Invariance refers to truth that should always be preserved by an operation.

Composition

Now that you have a handy amplifier component, it's time to use it in an electric guitar. Add the `ElectricGuitar` class implementation at the end of the playground right after the `Amplifier` class declaration:

```
// 1
class ElectricGuitar: Guitar {
  // 2
  let amplifier: Amplifier

  // 3
  init(brand: String, stringGauge: String = "light", amplifier: Amplifier) {
    self.amplifier = amplifier
    super.init(brand: brand, stringGauge: stringGauge)
  }

  // 4
  override func tune() -> String {
    amplifier.plugIn()
    amplifier.volume = 5
    return "Tune \(brand) electric with E A D G B E"
  }

  // 5
  override func play(_ music: Music) -> String {
    let preparedNotes = super.play(music)
    return "Play solo \(preparedNotes) at volume \(amplifier.volume)."
  }
```

```
}
```

Taking this step by step:

. `ElectricGuitar` is a concrete type that derives from the abstract, intermediate base class `Guitar`.

. An electric guitar contains an amplifier. This is a *has-a* relationship and not an *is-a* relationship as with inheritance.

. A custom initializer that initializes all of the stored properties and then calls the super class.

. A reasonable `tune()` method.

. A reasonable `play()` method.

In a similar vain, add the `BassGuitar` class declaration at the bottom of the playground right after the `ElectricGuitar` class implementation:

```
class BassGuitar: Guitar {
  let amplifier: Amplifier

  init(brand: String, stringGauge: String = "heavy", amplifier: Amplifier) {
    self.amplifier = amplifier
    super.init(brand: brand, stringGauge: stringGauge)
  }

  override func tune() -> String {
    amplifier.plugIn()
    return "Tune \(brand) bass with E A D G"
  }

  override func play(_ music: Music) -> String {
    let preparedNotes = super.play(music)
    return "Play bass line \(preparedNotes) at volume \(amplifier.volume)."
  }
}
```

This creates a bass guitar which also utilizes a (has-a) amplifier. Class containment in action. Time for another challenge!

Challenge: You may have heard that classes follow *reference semantics*. This means that variables holding a class instance actually hold a reference to that instance. If you have two variables with the same reference, changing data in one will change data in the other, and it's actually the same thing. Show reference semantics in action by instantiating an amplifier and sharing it between a Gibson electric guitar and a Fender bass guitar.
Add the following test code at the bottom of the playground right after the `BassGuitar` class declaration:

```
let amplifier = Amplifier()
let electricGuitar = ElectricGuitar(brand: "Gibson", stringGauge: "medium", amplifier: amplifier)
electricGuitar.tune()

let bassGuitar = BassGuitar(brand: "Fender", stringGauge: "heavy", amplifier: amplifier)
bassGuitar.tune()

// Notice that because of class reference semantics, the amplifier is a shared
// resource between these two guitars.

bassGuitar.amplifier.volume
electricGuitar.amplifier.volume

bassGuitar.amplifier.unplug()
bassGuitar.amplifier.volume
electricGuitar.amplifier.volume

bassGuitar.amplifier.plugIn()
bassGuitar.amplifier.volume
electricGuitar.amplifier.volume
```

Reveal

Polymorphism

One of the great strengths of object oriented programming is the ability to use different objects through the same interface while each behaves in its own unique way. This is *polymorphism* meaning "many forms". Add the `Band` class implementation at the end of the playground:

```
class Band {
  let instruments: [Instrument]
```

```
  init(instruments: [Instrument]) {
    self.instruments = instruments
  }

  func perform(_ music: Music) {
    for instrument in instruments {
      instrument.perform(music)
    }
  }
}
```

The `Band` class has an `instruments` array stored property which you set in the initializer. The band performs live on stage by going through the `instruments` array in a `for in` loop and calling the `perform(_:)` method for each instrument in the array.
Now go ahead and prepare your first rock concert. Add the following block of code at the bottom of the playground right after the `Band` class implementation:

```
let instruments = [piano, acousticGuitar, electricGuitar, bassGuitar]
let band = Band(instruments: instruments)
band.perform(music)
```

You first define an `instruments` array from the `Instrument` class instances you've previously created. Then you declare the `band` object and configure its `instruments` property with the `Band` initializer. Finally you use the `band` instance's `perform(_:)` method to make the band perform live music (print results of tuning and playing).
Notice that although the `instruments` array's type is `[Instrument]`, each instrument performs accordingly depending on its *class type*. This is how *polymorphism* works in practice: you now perform in live gigs like a pro! :]
Note: If you want to learn more about classes, check out our tutorial on Swift enums, structs and classes.

Access Control

You have already seen `private` in action as a way to hide complexity and protect your classes from inadvertently getting into invalid states (i.e. breaking the invariant). Swift goes further and provides four levels of access control including:
private: Visible just within the class.
fileprivate: Visible from anywhere in the same file.

internal: Visible from anywhere in the same module or app.
public: Visible anywhere outside the module.
There are additional access control related keywords:
open: Not only can it be used anywhere outside the module but also can be subclassed or overridden from outside.
final: Cannot be overridden or subclassed.
If you don't specify the access of a class, property or method, it defaults to `internal` access. Since you typically only have a single module starting out, this lets you ignore access control concerns at the beginning. You only really need to start worrying about it when your app gets bigger and more complex and you need to think about hiding away some of that complexity.

Making a Framework

Suppose you wanted to make your own music and instrument framework. You can simulate this by adding definitions to the compiled sources of your playground. First, delete the definitions for `Music` and `Instrument` from the playground. This will cause lots of errors that you will now fix.
Make sure the *Project Navigator* is visible in Xcode by going to *View\Navigators\Show Project Navigator*. Then right-click on the *Sources* folder and select *New File* from the menu. Rename the file *MusicKit.swift* and delete everything inside it. Replace the contents with:

```
// 1
final public class Music {
  // 2
  public let notes: [String]

  public init(notes: [String]) {
    self.notes = notes
  }

  public func prepared() -> String {
    return notes.joined(separator: " ")
  }
}

// 3
open class Instrument {
  public let brand: String
```

```
    public init(brand: String) {
      self.brand = brand
    }

    // 4
    open func tune() -> String {
      fatalError("Implement this method for \(brand)")
    }

    open func play(_ music: Music) -> String {
      return music.prepared()
    }

    // 5
    final public func perform(_ music: Music) {
      print(tune())
      print(play(music))
    }
  }
```

Save the file and switch back to the main page of your playground. This will continue to work as before. Here are some notes for what you've done here:

. `final public` means that is going to be visible by all outsiders but you cannot subclass it.

. Each stored property, initializer, method must be marked `public` if you want to see it from an outside source.

. The class `Instrument` is marked `open` because subclassing is allowed.

. Methods can also be marked `open` to allow overriding.

. Methods can be marked `final` so no one can override them. This can be a useful guarantee.

Where to Go From Here?

You can download the final playground for this tutorial which contains the tutorial's sample code.

You can read more about object oriented programming in our Swift Apprentice book or challenge yourself even more with our Design Patterns by Tutorials book.
I hope you enjoyed this tutorial and if you have any questions or comments, please join the forum discussion below!