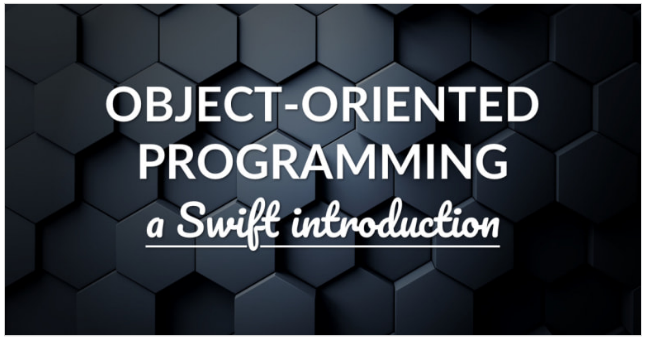
# Introduction Of Object-Oriented Programming In Swift

Written by Reinder de Vries (https://learnappmaking.com/about/) on June 3 2018 in App Development (https://learnappmaking.com/category/app-development/)



(https://learnappmaking.com/wp-content/uploads/2018/06/object-oriented-programming-oop-swift-introduction-770x400.jpg)

Object-Oriented Programming (OOP (Object-Oriented Programming)) helps you structure your Swift code with so-called *classes*. These classes have *properties* and *functions*, and classes can *inherit* attributes from each other.

This article is an introduction of Object-Oriented Programming with Swift. We'll dive into classes, objects, properties, functions, and a concept called *inheritance*. I'll show you what the purpose of Object-Oriented

**Hi, I'm Reinder.**
I help developers play with code.

About me (https://learnappmaking.com/about/)

Check out the app I'm building (https://learnappmaking.com/category/crest-app-series/)

Get started for free (https://learnappmaking.com/basics/)

## Most Popular Content

How To Learn iOS App Development (https://learnappmaking.com/learn-ios-development-how-to/)

27 App Marketing Strategies That Just Work (https://learnappmaking.com/app-marketing-strategy/)

How To Make An App (In 9 Steps) (https://learnappmaking.com/how-to-make-an-app/)

Programming is, with live examples, and why it's important in practical iOS development (https://learnappmaking.com/ios-development-course/).

When you don't use a structure like Object-Oriented Programming in your code, it's like a **waterfall**: the code flows from top to bottom. Object-Oriented Programming, on the other hand, is more like a **house**: you define your code's foundation, and build on top of that, room by room, one building block at a time.

Let's dive in!

## Classes And Instances

Let's start with the most important aspect of Object-Oriented Programming: *classes*. They are the building blocks of your code.

Here's an example of a class in Swift:

For Loops In Swift (How To) (https://learnappmaking.com/loops-swift-how-to/)

Working With Codable And JSON In Swift (https://learnappmaking.com/codable-json-swift-how-to/)

How To: Random Numbers in Swift (https://learnappmaking.com/random-numbers-swift/)

The Scene Delegate In Xcode 11 And iOS 13 (https://learnappmaking.com/scene-delegate-app-delegate-xcode-11-ios-13/)

» Show latest (https://learnappmaking.com/blog/)

### Browse Topics

App Development (https://learnappmaking.com/category/app-development/)

App Marketing (https://learnappmaking.com/category/app-marketing/)

App Business (https://learnappmaking.com/category/app-business/)

Crest App Series (https://learnappmaking.com/category/crest-app-series/)

Careers (https://learnappmaking.com/category/careers/)

### Swift Sandbox

```
class Car
{

}
```

Inside the squiggly brackets you can define variables, called *properties*, and functions, called *methods*, that belong to this class `Car`. You can see a *class* as a structure for code, like a toolbox that contains a screwdriver, wrench, hammer and drill.

In Swift, you can create *instances* of a *class*. Like this:

```
class Car
{

}

let ferrari:Car = Car()
```

The constant `ferrari` now contains an instance of the class `Car`. You've created an instance with the *initializer* `Car()`, and with the *assignment operator* `=` you've assigned that instance to the constant `ferrari`.

You can now pass that instance of `Car` around in your code, just like any other value. The class `Car` is the type (https://learnappmaking.com/swift-variables-constants-how-to/) of `ferrari`, much like a variable `score` can be of type `Int`.

You can create as much instances of one class as you want. That's the power of Object-Oriented Programming! You create the `Car` class once, and then use it to create an entire garage of cars. This is called *code reuse*, and it's a staple in practical iOS development.

> *Instances* of a class are often simply called *objects*.

It can be challenging to visualize abstract concepts like classes and instances. Let's make an analogy:

*Code Swift right in your browser!*
Go to the Swift Sandbox (https://learnappmaking.com/sandbox/)

What are you lookir

SEARCH

/

- A class is like the blueprint for a building. An architect draws the blueprint, and then passes it on to a constructor.
- An instance is like the building itself. The construction worker uses the blueprint to create the building.

A building isn't an exact copy of its blueprint! It's just a live representation of the blueprint, with windows, doors, a roof, etcetera.

When coding your app, you're building an entire city by using a great number of blueprints. Some of these blueprints are provided by the government (i.e., Cocoa Touch SDKs), such as the blueprint for a standard fire station and train station. Other blueprints are designed by you, the architect.

Your city is not just a pile of blueprints. You create instances of these blueprints: the actual buildings! And you create the blueprint for an apartment complex once, and then use it to build apartments all over the city.

Makes sense? Let's move on to the next aspect of Object-Oriented Programming: properties and functions.

**Learn More (https://learnappmaking.com/ios-development-course/)**

**Start for Free (https://learnappmaking.com/basics/)**

## Properties And Functions

A class can define properties and methods. A property is like a variable that belongs to a class, and a method is like a function that belongs to a class.

Properties store information that belongs to an instance of a class, and methods can execute tasks that belong to an instance of a class. This is very similar to variables (https://learnappmaking.com/swift-variables-constants-how-to/) and functions (https://learnappmaking.com/swift-functions-how-to/), except that the variables and functions are now part of a structure – the *class*.

Do you see how a class structures information and tasks in your code? The class "wraps" variables and functions that belong together.

> A *method* is a function that belongs to a class, but for the sake of simplicitly, we're just going to call them *functions*.

Let's look at an example:

```
class Car
{
    var wheels:Int = 0
    var maxSpeed:Int = 0
}
```

Here's what happens:

- You declare a class called `Car` with `class Car`. The *class body* goes between the squiggly brackets `{` and `}`.
- You declare two properties: a property called `wheels` of type `Int` and another property called `maxSpeed` of type `Int`, both with a default value of `0`.

Anything you can do with a variable, you can do with a property. A property has a type (https://learnappmaking.com/swift-variables-constants-how-to/#types) and you declare it with `var`, just like a variable. You can use type inference (https://learnappmaking.com/swift-variables-constants-how-to/#type-inference) and declare constants with `let`.

The big difference, of course, is that these properties belong to a class. Every instance of the class `Car` will now have the properties `wheels` and `maxSpeed`.

We can create a fast Ferrari, like this:

```
let ferrari = Car()
ferrari.wheels = 4
ferrari.maxSpeed = 300
```

And we can create a slower All Terrain Vehicle (ATV), like this:

```
let atv = Car()
atv.wheels = 8
atv.maxSpeed = 80
```

That's pretty cool, right? You use *dot-syntax* to access properties on an object, such as `ferrari.wheels`. Most properties can be read from (called *getting*) and written to (called *setting*), unless otherwise specified.

> Want to learn more about variables? Check this article: Variables And Constants In Swift Explained (https://learnappmaking.com/swift-variables-

> constants-how-to/).

OK, now let's continue with functions (https://learnappmaking.com/swift-functions-how-to/). Everything we've done with properties so far, applies to functions too. When a class declares a function, and that function is available on every instance of that class.

Like this:

```
class Car
{
    func drive()
    {
        print("VROOOOOOM!!!")
    }
}
```

In the above example we're declaring a function `drive()` on the class `Car`. It prints a simple line of text to the *Console* when the function is executed.

You can now call that function `drive()` on the `Car` class with the same *dot-syntax*. Like this:

```
let ferrari = Car()
ferrari.drive()
// Output: VROOOOOOM!!!
```

> Want to learn more about functions? Check this article: Functions In Swift Explained (https://learnappmaking.com/swift-functions-how-to/).

## Subclassing And Inheritance

You now know that Object-Oriented Programming is used to structure your code in classes. A class can have properties, to store information, and functions, to execute tasks.

Object-Oriented Programming also has several advanced concepts, such as *inheritance*, *polymorphism*, *delegation* and *composition*. We'll focus on inheritance, for now.

Let's say we've defined a simple class called `Vehicle`. We can then create *subclasses* of `Vehicle`, such as:

- `Bicycle` – a vehicle with no engine and 2 wheels
- `RaceCar` – a vehicle with a powerful engine and 4 wheels
- `Bus` – a large vehicle with 8 wheels and space for 50 people

See how each of these subclasses is-a-kind-of vehicle, and also defines some attributes of its own? That's what subclassing and inheritance is about.

Here's the gist of it:

- A subclass inherits properties and functions from its superclass
- A subclass can *override* properties and functions from its superclass, effectively replacing them with its own implementation
- A subclass can *extend* its superclass with new properties and functions
- A subclass and superclass can be casted (https://learnappmaking.com/type-casting-swift-how-to/) within their own hierarchy

The principle of inheritance enables you to create hierarchy in your code, and helps you reuse parts of your code. It doesn't make sense to copy the attributes that different kinds of vehicles share, when you can reuse them.

Let's take a look at some code. Here's the *base class* `Vehicle`:

```
class Vehicle
{
    var wheels:Int = 0

    func drive()
    {
        print("Driving this vehicle!")
    }
}
```

Then, let's create a subclass:

```
class Bus: Vehicle
{
    var seats:Int = 0
    var gears:Int = 0

    func openDoors()
    {
        print("Opening bus doors...")
    }

    override func drive()
    {
        print("Out of the waaaayyy! This bus is uns
toppable!")
    }
}
```

Here's what happens in the above code:

- You declare a class named `Bus` and it *subclasses* `Vehicle`, as per the `Bus: Vehicle` syntax.
- You declare two properties on the `Bus` class, `seats` and `gears`, both of type `Int`.
- You declare one function `openDoors()` on the `Bus` class, which opens the doors of the bus.
- You declare another function `drive()` on the `Bus` class. This function is *overridden*. The superclass implementation of `Vehicle` is replaced with a new function.

To see how inheritance works, let's create an instance of `Bus`.

```
let greyhound = Bus()
greyhound.wheels = 8
greyhound.seats = 200
greyhound.gears = 7
```

See what happens in the above code?

- The `greyhound` bus has 8 wheels. This `wheels` property is inherited from `Vehicle`, because all vehicles have wheels. See how the property `wheels` isn't declared on the `Bus` class? Instead, its declared on `Vehicle`, and inherited, because `Bus` subclasses `Vehicle`.
- The `seats` and `gears` properties are declared on the `Bus` class. It's important to note that the `Vehicle` class does **not** have these properties!

When you call the function `drive()` on the `greyhound` object, the overridden implementation is called. Like this:

```
greyhound.drive()
// Output: Out of the waaaayyy! This bus is unstopp
able!
```

The output is **not** *Driving this vehicle!*, because the function `drive()` is overridden by the subclass implementation.

It's important to note here that a bicycle, bus and racecar all have wheels, albeit a different number of wheels. Subclassing isn't so much about the value of properties, such as a different number of wheels, but about the properties themselves.

A bicycle has pedals and a frame, whereas a bus has doors that can swing open. It's not about the number of wheels, but instead about the attributes that classes in the hierarchy share with each other.

## Try Object-Oriented Programming Yourself!

Alright, let's put what you've learned into practice! You can use the *Swift Sandbox* below to try out Swift code. Play around with classes, properties, functions and subclassing.

```
class Vehicle
{
    var wheels:Int = 0
    var maxSpeed:Int = 0

    func drive()
    {
        print("This vehicle is driving!")
    }
}

class RaceCar: Vehicle
{
    var hasSpoiler = true

    override func drive()
    {
        print("VROOOOM!!!")
    }
}

class Bus: Vehicle
{
    var seats:Int = 0
    var gear:Int = 1

    func shiftGears()
    {
        gear += 1
    }
```

PLAY

☐ Hide warnings

## Further Reading

Object-Oriented Programming helps you structure your code. It enables you to reuse and extend code, without unnecessarily repeating yourself. Ultimately, that helps you to avoid bugs and code more productively.

Think about what the words *"object oriented"* mean. Object-Oriented Programming is quite literally oriented around objects. Instead of letting your code flow like a waterfall, you organize it in classes and objects, with their properties and functions. Much clearer!

Object-Oriented Programming is a fundamental topic in iOS development. OOP is literally everywhere, which makes it all the more important to master properly.

Want to learn more? Check out these resources:

- Functions In Swift Explained (https://learnappmaking.com/swift-functions-how-to/)
- Variables And Constants In Swift Explained (https://learnappmaking.com/swift-variables-constants-how-to/)
- Type Casting In Swift Explained (https://learnappmaking.com/type-casting-swift-how-to/)
- Understanding Model-View-Controller (MVC) In Swift (https://learnappmaking.com/model-view-controller-mvc-swift/)

**Enjoyed this article? Please share it!**

---

Reinder de Vries

## Reinder de Vries

Reinder de Vries is a professional iOS developer. He teaches app developers how to build their own apps at LearnAppMaking.com. Since 2009 he has developed a few dozen apps for iOS, worked

for global brands and lead development at several startups. When he’s not coding, he enjoys strong espresso and traveling.

---