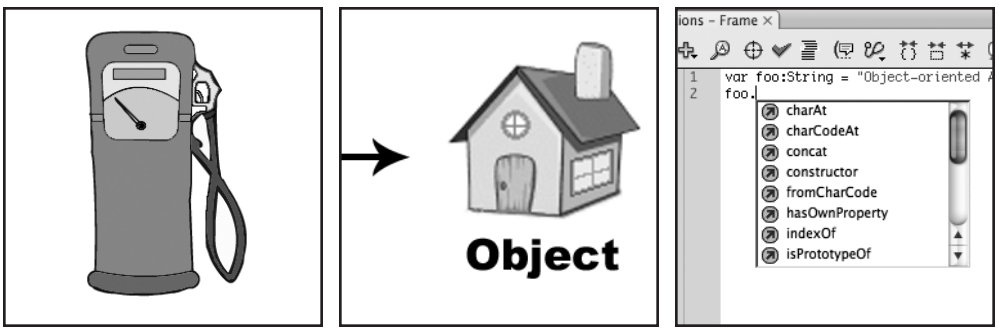
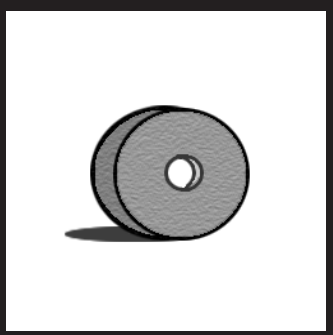
PART ONE **OOP AND ACTIONSCRIPT**



1 INTRODUCTION TO OOP



Object-oriented programming (OOP) sounds much scarier than it actually is. Essentially OOP is nothing more than a way of looking at a particular problem and breaking it down into smaller pieces called *objects*. These objects form the building blocks of object-oriented applications, and when designed properly they help form a solid framework on which to build your project.

The scoop with OOP

Before OOP became commonplace, we had something called *procedural programming*, which often required developers to write very complex and highly interdependent code. A minor change to any part of the code could spell disaster for the entire application. Debugging that type of application was a terribly painful and time-consuming task that often resulted in the need to completely rebuild large pieces of code.

When more and more user interaction got introduced in applications, it became apparent that procedural programming wouldn't cut it. Object-oriented programming was born as an attempt to solve these very problems. Although it certainly isn't the be-all and end-all of successful programming, OOP does give developers a great tool for handling any kind of application development.

The wonderful thing about object-oriented thinking is that you can look at practically any item in terms of a collection of objects. Let's look at a car for example. To the average Joe, a car is simply a vehicle (or object) that gets you places. If you ask a mechanic about a car, he'll most likely tell you about the engine, the exhaust, and all sorts of other parts. All these car parts can also be thought of as individual objects that work together to form a larger object, "the car." None of these parts actually know the inner workings of the other parts, and yet they work (or should work) together seamlessly.

Understanding the object-oriented approach

> *"'See that bird?' he says. 'It's a Spencer's warbler. (I knew he didn't know the real name.) Well, in Italian, it's a Chutto Lapittida. In Portuguese, it's a Bom da Peida. In Chinese, it's a Chung-long-tah, and in Japanese, it's a Katano Tekeda. You can know the name of that bird in all the languages of the world, but when you're finished, you'll know absolutely nothing whatever about the bird. You'll only know about humans in different places, and what they call the bird. So let's look at the bird and see what it's doing, that's what counts.'"*
>
> *—Richard Feynman*

When studying OOP, you'll come across a *plethora* of big words like *encapsulation*, *polymorphism*, and *inheritance*. Truth be told the ideas behind them are often quite simple, and there's no real need to memorize those terms unless you'd like to use them for showing off at your next family get-together.

Knowing the theory behind this terminology is, however, essential, and that's just what we'll be discussing next.

Classes and objects

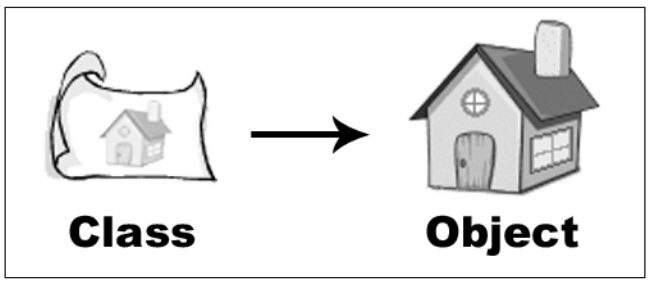
When studying OOP, you cannot ignore classes and objects, as those are the fundamental building blocks of any project. A good understanding of what classes and objects are and the roles they play will help you get on track to understanding OOP.

There's a subtle difference between a class and an object. A *class* is a self-contained description for a set of related services and data. Classes list the services they provide without revealing how they work internally. Classes aren't generally able to work on their own; they need to instantiate at least one object that is then able to act on the services and data described in the class.

Suppose you want to build a house. Unless you build it yourself, you need an architect and a builder. The architect drafts a blueprint, and the builder uses it to construct your house. Software developers are architects, and classes are their blueprints. You cannot use a class directly, any more than you could move your family into a blueprint. Classes only describe the final product. To actually do something you need an *object*.

If a class is a blueprint, then an object is a house. Builders create houses from blueprints; OOP creates objects from classes. OOP is efficient. You write the class once and create as many objects as needed.

Because classes can be used to *create* multiple objects, objects are often referred to as *class instances*.

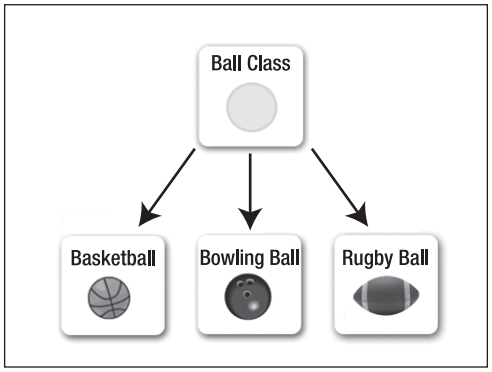


Properties

Properties give individual objects unique qualities. Without properties, each house (from the previous example) would remain identical to its neighbors (all constructed from the same blueprint). With properties, each house is unique, from its exterior color to the style of its windows.

Let's look at a Ball class for example. From that one class you can create multiple ball instances; however, not all balls look identical to one another. By providing your Ball class

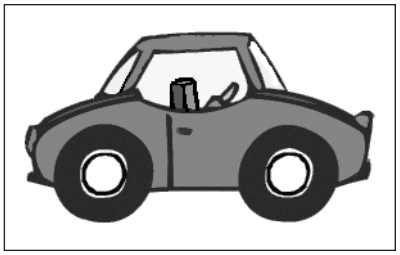
with properties such as color, weight, and shape, you can create instances that describe balls as diverse as a basketball, bowling ball, or rugby ball just by assigning different values to properties in each instance of the class.



In OOP, you write classes to offer predefined behaviors and maybe hold some data. Next, you create one or more objects from a class. Finally, you endow objects with their own individual property values. The progression from classes to objects to objects with unique properties is the essence of OOP.

Encapsulation: Hiding the details

When you get into your car, you turn the key, the car starts, and off you go. You don't need to understand how the car parts work to find yourself in rush-hour traffic. The car starts when you turn the key. Car designers hide the messy internal details so you can concentrate on important things like finding another radio station. OOP calls this concept *encapsulation*.



Analogies like the preceding car example are very useful to explain concepts such as encapsulation, but it is no doubt more appealing to take an in-depth look at potential real-world scenarios like, for example, an accounting office.

Accountants love details (all the numbers, receipts, and invoices). The accountant's boss, however, is interested in the bottom line. If the bottom line is zero, the company is debt-free. If the bottom line is positive, the company is profitable. She is happy to ignore all the messy details and focus on other things. Encapsulation is about ignoring or hiding internal details. In business, this is delegation. Without it, the boss may need to deal with accounting, tax law, and international trading at a level beyond her ability.

OOP loves encapsulation. With encapsulation, classes hide their own internal details. Users of a class (yourself, other developers, or other applications) are not required to know or care why it works. Class users just need the available service names and what to provide to

use them. Building classes is an abstraction process; you start with a complex problem, and then reduce it down (abstracting it) to a list of related services. Encapsulation simplifies software development and increases the potential for code reuse.

To demonstrate, I'll present some pseudo-code (false code). You can't enter pseudo-code into a computer, but it's great for previewing ideas. First, you need an Accounting class:

```
Start Of Accounting Class
End Of Accounting Class
```

Everything between the start and end line is the Accounting class. A useless class so far, because it's empty. Let's give the Accounting class something to do:

```
Start Of Accounting Class
    Start Of Bottom Line Service
        (Internal Details Of Bottom Line Service)
    End Of Bottom Line Service
End Of Accounting Class
```

Now the Accounting class has a Bottom Line service. How does that service work? Well, I know (because I wrote the code), but you (as a user of my class) have no idea. That's exactly how it should be. You don't know or care how my class works. You just use the Bottom Line service to see if the company is profitable. As long as my class is accurate and dependable, you can go about your business. You want to see the details anyway? OK, here they are:

```
Start Of Accounting Class
    Start Of Bottom Line Service
        Do Invoice Service
        Do Display Answer Service
    End Of Bottom Line Service
End Of Accounting Class
```

Where did the Invoice and Display Answer services come from? They're part of the class too, but encapsulation is hiding them. Here they are:

```
Start Of Accounting Class
    Start Of Bottom Line Service
        Do Invoice Service
        Do Display Answer Service
    End Of Bottom Line Service

    Start Of Invoice Service
        (Internal Details Of Invoice Service)
    End Of Invoice Service

    Start Of Display Answer Service
        (Internal Details Of Display Answer Service)
    End Of Display Answer Service
End Of Accounting Class
```

The Bottom Line service has no idea how the Invoice service works, nor does it care. You don't know the details, and neither does the Bottom Line service. This type of simplification is the primary benefit of encapsulation. Finally, how do you request an answer from the Bottom Line service? Easy, just do this:

```
Do Bottom Line Service
```

That's all. You're happy, because you only need to deal with a single line of code, which is essentially the interface that the class exposes. The Bottom Line service (and encapsulation) handles the details for you.

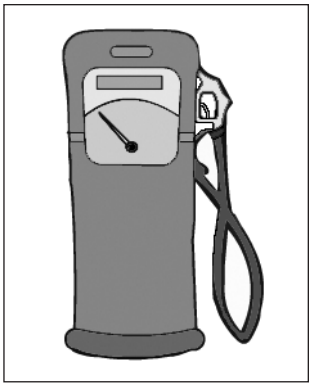
When I speak of hiding code details, I'm speaking conceptually. I don't mean to mislead you. This is just a mental tool to help you understand the importance of abstracting the details. With encapsulation, you're not actually hiding code (physically). If you were to view the full Accounting class, you'd see the same code that I see.

```
Start Of Accounting Class
    Start Of Bottom Line Service
        Do Invoice Service
        Do Display Answer Service
    End Of Bottom Line Service

    Start Of Invoice Service
        Gather Invoices
        Return Sum
    End Of Invoice Service

    Start Of Display Answer Service
        Display Sum
    End Of Display Answer Service
End Of Accounting Class
```

If you're wondering why some of the lines are indented, this is standard practice (that is not followed often enough). It shows, at a glance, the natural hierarchy of the code (of what belongs to what). Please adopt this practice when you write computer code.

Polymorphism: Exhibiting similar features

Are you old enough to remember fuel stations before the self-service era? You could drive into these places and somebody else would fill up your tank. The station attendant knew about OOP long before you did. He put the fuel nozzle into the tank (any tank) and pumped the fuel! It didn't matter if you drove a Ford, a Chrysler, or a Datsun. All cars have fuel tanks, so this behavior is easy to repeat for any car. OOP calls this concept *polymorphism*.



Much like cars need fuel to run, I take my daily dose of vitamins by drinking a glass of orange juice at breakfast. This incidentally brings me to a great example showing the concept of polymorphism.

Oranges have pulp. Lemons have pulp. Grapefruits have pulp. Cut any of these fruit open, I dare you, and try to scoop out the fruit with a spoon. Chances are you'll get a squirt of citrus juice in your eye. Citrus fruits know exactly where your eye is, but you don't have to spoon them out to know they share this talent (they're all acid-based juice-squirters). Look at the following Citrus class:

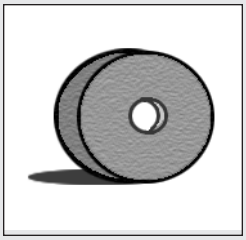
```
Start Of Citrus Class
    Start Of Taste Service
        (Internal Details Of Taste Service)
    End Of Taste Service

    Start Of Squirt Service
        (Internal Details Of Squirt Service)
    End Of Squirt Service
End Of Citrus Class
```

You can use the Citrus class as a base to define other classes:

```
Start Of Orange Class
    Using Citrus Class
    Property Named Juice
End Of Orange Class

Start Of Lemon Class
    Using Citrus Class
    Property Named Juice
End Of Lemon Class

Start Of Grapefruit Class
    Using Citrus Class
    Property Named Juice
End Of Grapefruit Class
```

Besides demonstrating inheritance again, the Orange, Lemon, and Grapefruit classes also exhibit similar behaviors. This is polymorphism. You know that the Orange, Lemon, and Grapefruit classes have the ability to squirt (inherited from the Citrus class), but each class has a Juice property. So the orange can squirt orange juice, the lemon can squirt lemon juice, and the grapefruit can squirt grapefruit juice. You don't have to know in advance which type of fruit, because they all squirt. In fact, you could taste the juice (inherited from the Citrus class) to know which fruit you're dealing with. That's polymorphism: multiple objects exhibiting similar features in different ways.

Inheritance: Avoid rebuilding the wheel



Grog roll wheel. Wheel good. Grog doesn't like rebuilding wheels. They're heavy, made of stone, and tend to crush feet when they fall over. Grog likes the wheel that his stone-age neighbor built last week. Sneaky Grog. Maybe he'll carve some holes into the wheel to store rocks, twigs, or a tasty snack. If Grog does this, he'll have added something new to the existing wheel (demonstrating inheritance long before the existence of computers).

Inheritance in OOP is a real timesaver. You don't need to modify your neighbor's wheel. You only need to tell the computer, "Build a replica of my neighbor's wheel, and then add this, and this, and this." The result is a custom wheel, but you didn't modify the original. Now you have two wheels, each unique. To clarify, here's some more pseudo-code:

```
Start Of Wheel Class
    Start Of Roll Service
        (Internal Details Of Roll Service)
    End Of Roll Service
End Of Wheel Class
```

The Wheel class provides a single service named Roll. That's a good start, but what if you want to make a tire? Do you build a new Tire class from scratch? No, you just use inheritance to build a Tire class, like this:

```
Start Of Tire Class
    Using Wheel Class
End Of Tire Class
```

By using the Wheel class as a starting point, the Tire class already knows how to roll (the tire is a type of wheel). Here's the next logical step:

```
Start Of Tire Class
    Using Wheel Class
    Property Named Size
End Of Tire Class
```

Now the Tire class has a property named size. That means you could create many unique Tire objects. All of the tires can roll (behavior inherited from the Wheel class), but each tire has its own unique size. You could add other properties to the Tire class too. With very little work, you could have small car tires that roll, big truck tires that roll, and bigger bus tires that roll.

What's next?

Now that wasn't too difficult, was it? In this chapter, I covered the basic idea of OOP as well as an introduction to some of its key features, including encapsulation, polymorphism, and inheritance. I'll explain those ideas in much greater detail in Part 3 of this book.

Coming up next, I will focus on the general programming concepts common to modern high-level computer languages.