# Object-oriented programming

Before diving into JavaScript, let's take a moment to review what people mean when they say object-oriented, and what the main features of this programming style are. Here's a list of concepts that are most often used when talking about **object-oriented programming** (**OOP**):

- Object, method, and property
- Class
- Encapsulation
- Aggregation
- Reusability/inheritance
- Polymorphism

Let's take a closer look into each one of these concepts. If you're new to the object-oriented programming lingo, these concepts might sound too theoretical, and you might have trouble grasping or remembering them from one reading. Don't worry, it does take a few tries, and the subject can be a little dry at a conceptual level. However, we'll look at plenty of code examples further on in the book, and you'll see that things are much simpler in practice.

## Objects

As the name object-oriented suggests, objects are important. An object is a representation of a thing (someone or something), and this representation is expressed with the help of a programming language. The thing can be anything, a real-life object, or a more convoluted concept. Taking a common object, a cat, for example, you can see that it has certain characteristics-color, name, weight, and so on and can perform some actions-meow, sleep, hide, escape, and so on. The characteristics of the object are called properties in OOP-speak, and the actions are called methods.

The analogy with the spoken language are as follows:

- Objects are most often named using nouns, such as book, person, and so on
- Methods are verbs, for example, read, run, and so on
- Values of the properties are adjectives

Take the sentence "The black cat sleeps on the mat" as an example. "The cat" (a noun) is the object, "black" (adjective) is the value of the color property, and "sleep" (a verb) is an action or a method in OOP. For the sake of the analogy, we can go a step further and say that "on the mat" specifies something about the action "sleep", so it's acting as a parameter passed to the `sleep` method.

## Classes

In real life, similar objects can be grouped based on some criteria. A hummingbird and an eagle are both birds, so they can be classified as belonging to some made-up `Birds` class. In OOP, a class is a blueprint or a recipe for an object. Another name for object is instance, so we can say that the eagle is one concrete instance of the general `Birds` class. You can create different objects using the same class because a class is just a template, while the objects are concrete instances based on the template.

There's a difference between JavaScript and the classic OO languages such as C++ and Java. You should be aware right from the start that in JavaScript, there are no classes; everything is based on objects. JavaScript has the notion of prototypes, which are also objects (we'll discuss them later in detail). In a classic OO language, you'd say something like-create a new object for me called `Bob`, which is of class `Person`. In a prototypal OO language, you'd say-I'm going to take this object called Bob's dad that I have lying around (on the couch in front of the TV?) and reuse it as a prototype for a new object that I'll call `Bob`.

## Encapsulation

Encapsulation is another OOP related concept, which illustrates the fact that an object contains (encapsulates) the following:

- Data (stored in properties)
- The means to do something with the data (using methods)

One other term that goes together with encapsulation is information hiding. This is a rather broad term and can mean different things, but let's see what people usually mean when they use it in the context of OOP.

Imagine an object, say, an MP3 player. You, as the user of the object, are given some interface to work with, such as buttons, display, and so on. You use the interface in order to get the object to do something useful for you, like play a song. How exactly the device is working on the inside, you don't know, and, most often, don't care. In other words, the implementation of the interface is hidden from you. The same thing happens in OOP when your code uses an object by calling its methods. It doesn't matter if you coded the object yourself or it came from some third-party library; your code doesn't need to know how the methods work internally. In compiled languages, you can't actually read the code that makes an object work. In JavaScript, because it's an interpreted language, you can see the source code, but the concept is still the same-you work with the object's interface without worrying about its implementation.

Another aspect of information hiding is the visibility of methods and properties. In some languages, objects can have `public`, `private`, and `protected` methods and properties. This categorization defines the level of access the users of the object have. For example, only the methods of the same object have access to the `private` methods, while anyone has access to the `public` ones. In JavaScript, all methods and properties are `public`, but we'll see that there are ways to protect the data inside an object and achieve privacy.

## Aggregation

Combining several objects into a new one is known as aggregation or composition. It's a powerful way to separate a problem into smaller and more manageable parts (divide and conquer). When a problem scope is so complex that it's impossible to think about it at a detailed level in its entirety, you can separate the problem into several smaller areas, and possibly then separate each of these into even smaller chunks. This allows you to think about the problem on several levels of abstraction.

Take, for example, a personal computer. It's a complex object. You cannot think about all the things that need to happen when you start your computer. But, you can abstract the problem saying that you need to initialize all the separate objects that your `Computer` object consists of the `Monitor` object, the `Mouse` object, the `Keyboard` object, and so on. Then, you can dive deeper into each of the subobjects. This way, you're composing complex objects by assembling reusable parts.

To use another analogy, a `Book` object can contain (aggregate) one or more `Author` objects, a `Publisher` object, several `Chapter` objects, a `TOC` (table of contents), and so on.

## Inheritance

Inheritance is an elegant way to reuse existing code. For example, you can have a generic object, `Person`, which has properties such as `name` and `date_of_birth`, and which also implements the `walk`, `talk`, `sleep`, and `eat` functionality. Then, you figure out that you need another object called `Programmer`. You can reimplement all the methods and properties that a `Person` object has, but it will be smarter to just say that the `Programmer` object inherits a `Person` object, and save yourself some work. The `Programmer` object only needs to implement more specific functionality, such as the `writeCode` method, while reusing all of the `Person` object's functionality.

In classical OOP, classes inherit from other classes, but in JavaScript, as there are no classes, objects inherit from other objects.

When an object inherits from another object, it usually adds new methods to the inherited ones, thus extending the old object. Often, the following phrases can be used interchangeably-B inherits from A and B extends A. Also, the object that inherits can pick one or more methods and redefine them, customizing them for its own needs. This way, the interface stays the same and the method name is the same, but when called on the new object, the method behaves differently. This way of redefining how an inherited method works is known as **overriding**.

## Polymorphism

In the preceding example, a `Programmer` object inherited all of the methods of the parent `Person` object. This means that both objects provide a `talk` method, among others. Now imagine that somewhere in your code, there's a variable called `Bob`, and it just so happens that you don't know if `Bob` is a `Person` object or a `Programmer` object. You can still call the `talk` method on the `Bob` object and the code will work. This ability to call the same method on different objects, and have each of them respond in their own way, is called polymorphism.

## OOP summary

Here's a quick table summarizing the concepts discussed so far:

| **Feature** | **Illustrates concept** |
|---|---|
| Bob is a man (an object). | Objects |
| Bob's date of birth is June 1, 1980, gender - male, and hair - black. | Properties |
| Bob can eat, sleep, drink, dream, talk, and calculate his own age. | Methods |
| Bob is an instance of the `Programmer` class. | Class (in classical OOP) |
| Bob is based on another object called `Programmer`. | Prototype<br><br>(in prototypal OOP) |
| Bob holds data, such as `birth_date`, and methods that work with the data, such as `calculateAge()`. | Encapsulation |
| You don't need to know how the calculation method works internally. The object might have some private data, such as the number of days in February in a leap year. You don't know, nor do you want to know. | Information hiding |
| Bob is part of a `WebDevTeam` object together with Jill, a `Designer` object, and Jack, a `ProjectManager` object. | Aggregation and composition |
| `Designer`, `ProjectManager`, and `Programmer` are all based on and extend a `Person` object. | Inheritance |
| You can call the methods `Bob.talk()`, `Jill.talk()`, and `Jack.talk()`, and they'll all work fine, albeit producing different results. Bob will probably talk more about performance, Jill about beauty, and Jack about deadlines. Each object inherited the method talk from Person and customized it. | Polymorphism and method overriding |