mdn web docs

Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm fundamental to many programming languages, including Java and C++. In this article, we'll provide an overview of the basic concepts of OOP. We'll describe three main concepts: **classes and instances**, **inheritance**, and **encapsulation**. For now, we'll describe these concepts without reference to JavaScript in particular, so all the examples are given in pseudocode.

Note: To be precise, the features described here are of a particular style of OOP called **class-based** or "classical" OOP. When people talk about OOP, this is generally the type that they mean.

After that, in JavaScript, we'll look at how constructors and the prototype chain relate to these OOP concepts, and how they differ. In the next article, we'll look at some additional features of JavaScript that make it easier to implement object-oriented programs.

Prerequisites:	Understanding JavaScript functions, familiarity with JavaScript basics (see First steps and Building

	blocks), and OOJS basics (see Introduction to objects and Object prototypes).
Objective:	To understand the basic concepts of class-based object-oriented programming.

Object-oriented programming is about modeling a system as a collection of objects, where each object represents some particular aspect of the system. Objects contain both functions (or methods) and data. An object provides a public interface to other code that wants to use it but maintains its own private, internal state; other parts of the system don't have to care about what is going on inside the object.

Classes and instances

When we model a problem in terms of objects in OOP, we create abstract definitions representing the types of objects we want to have in our system. For example, if we were modeling a school, we might want to have objects representing professors. Every professor has some properties in common: they all have a name and a subject that they teach. Additionally, every professor can do certain things: they can all grade a paper and they can introduce themselves to their students at the start of the year, for example.

So `Professor` could be a **class** in our system. The definition of the class lists the data and methods that every professor has.

In pseudocode, a `Professor` class could be written like this:

```
class Professor
    properties
        name
        teaches
    methods
        grade(paper)
        introduceSelf()
```

This defines a `Professor` class with:

- two data properties: `name` and `teaches`
- two methods: `grade()` to grade a paper and `introduceSelf()` to introduce themselves.

On its own, a class doesn't do anything: it's a kind of template for creating concrete objects of that type. Each concrete professor we create is called an **instance** of the `Professor` class. The process of creating an instance is performed by a special function called a **constructor**. We pass values to the constructor for any internal state that we want to initialize in the new instance.

Generally, the constructor is written out as part of the class definition, and it usually has the same name as the class itself:

```
class Professor
    properties
        name
        teaches
    constructor
        Professor(name, teaches)
    methods
        grade(paper)
        introduceSelf()
```

This constructor takes two parameters, so we can initialize the `name` and `teaches` properties when we create a new concrete professor.

Now that we have a constructor, we can create some professors. Programming languages often use the keyword `new` to signal that a constructor is being called.

JS

```js
walsh = new Professor("Walsh", "Psychology");
lillian = new Professor("Lillian", "Poetry");

walsh.teaches; // 'Psychology'
walsh.introduceSelf(); // 'My name is Professor Walsh and I will be your Psychology professor.'

lillian.teaches; // 'Poetry'
lillian.introduceSelf(); // 'My name is Professor Lillian and I will be your Poetry professor.'
```

This creates two objects, both instances of the `Professor` class.

Inheritance

Suppose in our school we also want to represent students. Unlike professors, students can't grade papers, don't teach a particular subject, and belong to a particular year.

However, students do have a name and may also want to introduce themselves, so we might write out the definition of a student class like this:

```
class Student
    properties
        name
        year
    constructor
        Student(name, year)
    methods
        introduceSelf()
```

It would be helpful if we could represent the fact that students and professors share some properties, or more accurately, the fact that on some level, they are the *same kind of thing*. **Inheritance** lets us do this.

We start by observing that students and professors are both people, and people have names and want to introduce themselves. We can model this by defining a new class `Person`, where we define all the common properties of people. Then, `Professor` and `Student` can both **derive** from `Person`, adding their extra properties:

```
class Person
    properties
        name
    constructor
        Person(name)
    methods
        introduceSelf()

class Professor : extends Person
    properties
        teaches
    constructor
        Professor(name, teaches)
    methods
        grade(paper)
        introduceSelf()

class Student : extends Person
    properties
        year
    constructor
        Student(name, year)
    methods
        introduceSelf()
```

In this case, we would say that `Person` is the **superclass** or **parent class** of both `Professor` and `Student`. Conversely, `Professor` and `Student` are **subclasses** or **child classes** of `Person`.

You might notice that `introduceSelf()` is defined in all three classes. The reason for this is that while all people want to introduce themselves, the way they do so is different:

JS

```js
walsh = new Professor("Walsh", "Psychology");
walsh.introduceSelf(); // 'My name is Professor Walsh and I will be your Psychology professor.'

summers = new Student("Summers", 1);
summers.introduceSelf(); // 'My name is Summers and I'm in the first year.'
```

We might have a default implementation of `introduceSelf()` for people who aren't students *or* professors:

JS

```js
pratt = new Person("Pratt");
pratt.introduceSelf(); // 'My name is Pratt.'
```

This feature - when a method has the same name but a different implementation in different classes - is called **polymorphism**. When a method in a subclass replaces the superclass's implementation, we say that the subclass **overrides** the version in the superclass.

Encapsulation

Objects provide an interface to other code that wants to use them but maintain their own internal state. The object's internal state is kept **private**, meaning that it can only be accessed by the object's own methods, not from other objects. Keeping an object's internal state private, and generally making a clear division between its public interface and its private internal state, is called **encapsulation**.

This is a useful feature because it enables the programmer to change the internal implementation of an object without having to find and update all the code that uses it: it creates a kind of firewall between this object and the rest of the system.

For example, suppose students are allowed to study archery if they are in the second year or above. We could implement this just by exposing the student's `year` property, and other code could examine that to decide whether the student can take the course:

JS

```js
if (student.year > 1) {
  // allow the student into the class
}
```

The problem is, if we decide to change the criteria for allowing students to study archery - for example by also requiring the parent or guardian to give their permission - we'd need to update every place in our system that performs this test. It would be better to have a `canStudyArchery()` method on `Student` objects, that implements the logic in one place:

```
class Student : extends Person
    properties
       year
    constructor
```

```
       Student(name, year)
    methods
       introduceSelf()
       canStudyArchery() { return this.year > 1 }
```

JS

```js
if (student.canStudyArchery()) {
  // allow the student into the class
}
```

That way, if we want to change the rules about studying archery, we only have to update the `Student` class, and all the code using it will still work.

In many OOP languages, we can prevent other code from accessing an object's internal state by marking some properties as `private`. This will generate an error if code outside the object tries to access them:

```
class Student : extends Person
    properties
       private year
    constructor
        Student(name, year)
    methods
       introduceSelf()
       canStudyArchery() { return this.year > 1 }

student = new Student('Weber', 1)
student.year // error: 'year' is a private property of Student
```

In languages that don't enforce access like this, programmers use naming conventions, such as starting the name with an underscore, to indicate that the property should be considered private.

OOP and JavaScript

In this article, we've described some of the basic features of class-based object-oriented programming as implemented in languages like Java and C++.

In the two previous articles, we looked at a couple of core JavaScript features: constructors and prototypes. These features certainly have some relation to some of the OOP concepts described above.

- **constructors** in JavaScript provide us with something like a class definition, enabling us to define the "shape" of an object, including any methods it contains, in a single place. But prototypes can be used here, too. For example, if a method is defined on a constructor's `prototype` property, then all objects created using that constructor get that method via their prototype, and we don't need to define it in the constructor.
- **the prototype chain** seems like a natural way to implement inheritance. For example, if we can have a `Student` object whose prototype is `Person`, then it can inherit `name` and override `introduceSelf()`.

But it's worth understanding the differences between these features and the "classical" OOP concepts described above. We'll highlight a couple of them here.

First, in class-based OOP, classes and objects are two separate constructs, and objects are always created as instances of classes. Also, there is a distinction between the feature used to define a class (the class syntax itself) and the feature used to instantiate an object (a constructor). In JavaScript, we can and often do create objects without any separate class definition, either using a function or an object literal. This can make working with objects much more lightweight than it is in classical OOP.

Second, although a prototype chain looks like an inheritance hierarchy and behaves like it in some ways, it's different in others. When a subclass is instantiated, a single object is created which combines properties defined in the subclass with properties defined further up the hierarchy. With prototyping, each level of the hierarchy is represented by a separate object, and they are linked together via the `__proto__` property. The prototype chain's behavior is less like inheritance and more like **delegation**. Delegation is a programming pattern where an object, when asked to perform a task, can perform the task itself or ask another object (its **delegate**) to perform the task on its behalf. In many ways, delegation is a more flexible way of combining objects than inheritance (for one thing, it's possible to change or completely replace the delegate at run time).

That said, constructors and prototypes can be used to implement class-based OOP patterns in JavaScript. But using them directly to implement features like inheritance is tricky, so JavaScript provides extra features, layered on top of the prototype model, that map more directly to the concepts of class-based OOP. These extra features are the subject of the next article.

Summary

This article has described the basic features of class-based object oriented programming, and briefly looked at how JavaScript constructors and prototypes compare with these concepts.

In the next article, we'll look at the features JavaScript provides to support class-based object-oriented programming.