Introduction to Object-Oriented JavaScript

Object-oriented to the core, JavaScript features powerful, flexible OOP capabilities. This article starts with an introduction to object-oriented programming, then reviews the JavaScript object model, and finally demonstrates concepts of object-oriented programming in JavaScript. This article does not describe the newer syntax for object-oriented programming in ECMAScript 6.

JavaScript review

If you don't feel confident about JavaScript concepts such as variables, types, functions, and scope, you can read about those topics in A re-introduction to JavaScript. You can also consult the JavaScript Guide.

Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm that uses abstraction to create models based on the real world. OOP uses several techniques from previously established paradigms, including modularity, polymorphism, and encapsulation. Today, many popular programming languages (such as Java, JavaScript, C#, C++, Python, PHP, Ruby and Objective-C) support OOP.

OOP envisions software as a collection of cooperating objects rather than a collection of functions or simply a list of commands (as is the traditional view). In OOP, each object can receive messages, process data, and send messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility.

OOP promotes greater flexibility and maintainability in programming, and is widely popular in large-scale software engineering. Because OOP strongly emphasizes modularity, object-oriented code is simpler to develop and easier to understand later on. Object-oriented code promotes more direct analysis, coding, and understanding of complex situations and procedures than less modular programming techniques.[1]

Terminology

Namespace

A container which lets developers bundle all functionality under a unique, application-specific name.

Class
: Defines the object's characteristics. A class is a template definition of an object's properties and methods.

Object
: An instance of a class.

Property
: An object characteristic, such as color.

Method
: An object capability, such as walk. It is a subroutine or function associated with a class.

Constructor
: A method called at the moment an object is instantiated. It usually has the same name as the class containing it.

Inheritance
: A class can inherit characteristics from another class.

Encapsulation
: A technique which involves bundling the data and the methods that use the data together.

Abstraction
: The conjunction of an object's complex inheritance, methods, and properties must adequately reflect a reality model.

Polymorphism
: Poly means "*many*" and morphism means "*forms*". Different classes might define the same method or property.

For a more extensive description of object-oriented programming, see Object-oriented programming at Wikipedia.

Prototype-based programming

Prototype-based programming is an OOP model that doesn't use classes, but rather it first accomplishes the behavior of any class and then reuses it (equivalent to inheritance in class-based languages) by decorating (or expanding upon) existing *prototype* objects. (Also called classless, prototype-oriented, or instance-based programming.)

The original (and most canonical) example of a prototype-based language is Self developed by David Ungar and Randall Smith. However, the class-less programming style grows increasingly popular lately, and has been adopted for programming languages such as JavaScript, Cecil, NewtonScript, Io, MOO, REBOL, Kevo, Squeak (when using the Viewer framework to manipulate Morphic components), and several others.[1]

JavaScript object oriented programming

Namespace

A namespace is a container which allows developers to bundle up functionality under a unique, application-specific name. **In JavaScript a namespace is just another object containing methods, properties, and objects.**

It's important to note that in JavaScript, there's no language-level difference between regular objects and namespaces. This differs from many other object-oriented languages, and can be a point of confusion for new JavaScript programmers.

The idea behind creating a namespace in JavaScript is simple: create one global object, and all variables, methods, and functions become properties of that object. Use of namespaces also reduces the chance of name conflicts in an application, since each application's objects are properties of an application-defined global object.

Let's create a global object called MYAPP:

```
// global namespace
var MYAPP = MYAPP || {};
```

In the above code sample, we first checked whether `MYAPP` is already defined (either in same file or in another file). If yes, then use the existing MYAPP global object, otherwise create an empty object called `MYAPP` which will encapsulate methods, functions, variables, and objects.

We can also create sub-namespaces (keep in mind that the global object needs to be defined first):

```
// sub namespace
MYAPP.event = {};
```

The following is code syntax for creating a namespace and adding variables, functions, and a method:

```javascript
// Create container called MYAPP.commonMethod for common method and properties
MYAPP.commonMethod = {
  regExForName: "", // define regex for name validation
  regExForPhone: "", // define regex for phone no validation
  validateName: function(name){
    // Do something with name, you can access regExForName variable
    // using "this.regExForName"
  },

  validatePhoneNo: function(phoneNo){
    // do something with phone number
  }
}

// Object together with the method declarations
MYAPP.event = {
    addListener: function(el, type, fn) {
    // code stuff
    },
    removeListener: function(el, type, fn) {
    // code stuff
    },
    getEvent: function(e) {
    // code stuff
    }

    // Can add another method and properties
}

// Syntax for Using addListener method:
MYAPP.event.addListener("yourel", "type", callback);
```

Standard built-in objects

JavaScript has several objects included in its core, for example, there are objects like `Math`, `Object`, `Array`, and `String`. The example below shows how to use the `Math` object to get a random number by using its `random()` method.

```javascript
console.log(Math.random());
```

Note: This and all further examples presume a function named `console.log()` is defined globally. The `console.log()` function is not actually a part of JavaScript itself, but many browsers implement it to aid in debugging.

See JavaScript Reference: Standard built-in objects for a list of the core objects in JavaScript.

Every object in JavaScript is an instance of the object `Object` and therefore inherits all its properties and methods.

Custom objects

The class

JavaScript is a prototype-based language and contains no `class` statement, such as is found in C++ or Java. This is sometimes confusing for programmers accustomed to languages with a `class` statement. Instead, JavaScript uses functions as constructors for classes. Defining a class is as easy as defining a function. In the example below we define a new class called Person with an empty constructor.

```
var Person = function () {};
```

The object (class instance)

To create a new instance of an object `obj` we use the statement `new obj`, assigning the result (which is of type `obj`) to a variable to access it later.

In the example above we define a class named `Person`. In the example below we create two instances (`person1` and `person2`).

```
var person1 = new Person();
var person2 = new Person();
```

Please see `Object.create()` for a new, additional, instantiation method that creates an uninitialized instance.

The constructor

The constructor is called at the moment of instantiation (the moment when the object instance is created). The constructor is a method of the class. In JavaScript the function serves as the constructor of the object, therefore there is no need to explicitly define a constructor method. Every action declared in the class gets executed at the time of instantiation.

The constructor is used to set the object's properties or to call methods to prepare the object for use. Adding class methods and their definitions occurs using a different syntax described later in this article.

In the example below, the constructor of the class `Person` logs a message when a `Person` is instantiated.

```
var Person = function () {
  console.log('instance created');
};

var person1 = new Person();
var person2 = new Person();
```

The property (object attribute)

Properties are variables contained in the class; every instance of the object has those properties. Properties are set in the constructor (function) of the class so that they are created on each instance.

The keyword `this`, which refers to the current object, lets you work with properties from within the class. Accessing (reading or writing) a property outside of the class is done with the syntax: `InstanceName.Property`, just like in C++, Java, and several other languages. (Inside the class the syntax `this.Property` is used to get or set the property's value.)

In the example below, we define the `firstName` property for the `Person` class at instantiation:

```
var Person = function (firstName) {
  this.firstName = firstName;
  console.log('Person instantiated');
};

var person1 = new Person('Alice');
var person2 = new Person('Bob');

// Show the firstName properties of the objects
console.log('person1 is ' + person1.firstName); // logs "person1 is Alice"
console.log('person2 is ' + person2.firstName); // logs "person2 is Bob"
```

The methods

Methods are functions (and defined like functions), but otherwise follow the same logic as properties. Calling a method is similar to accessing a property, but you add `()` at the end of the method name, possibly with arguments. To define a method, assign a function to a named property of the class's `prototype` property. Later, you can call the method on the object by the same name as you assigned the function to.

In the example below, we define and use the method `sayHello()` for the `Person` class.

```
var Person = function (firstName) {
  this.firstName = firstName;
};

Person.prototype.sayHello = function() {
  console.log("Hello, I'm " + this.firstName);
};

var person1 = new Person("Alice");
var person2 = new Person("Bob");

// call the Person sayHello method.
person1.sayHello(); // logs "Hello, I'm Alice"
person2.sayHello(); // logs "Hello, I'm Bob"
```

In JavaScript methods are regular function objects bound to an object as a property, which means you can invoke methods "out of the context". Consider the following example code:

```
var Person = function (firstName) {
  this.firstName = firstName;
};

Person.prototype.sayHello = function() {
  console.log("Hello, I'm " + this.firstName);
};

var person1 = new Person("Alice");
var person2 = new Person("Bob");
var helloFunction = person1.sayHello;

// logs "Hello, I'm Alice"
person1.sayHello();

// logs "Hello, I'm Bob"
person2.sayHello();
```

```
// logs "Hello, I'm undefined" (or fails
// with a TypeError in strict mode)
helloFunction();

// logs true
console.log(helloFunction === person1.sayHello);

// logs true
console.log(helloFunction === Person.prototype.sayHello);

// logs "Hello, I'm Alice"
helloFunction.call(person1);

```

As that example shows, all of the references we have to the `sayHello` function— the one on `person1`, on `Person.prototype`, in the `helloFunction` variable, etc.— refer to the *same function*. The value of `this` during a call to the function depends on how we call it. Most commonly, when we call `this` in an expression where we got the function from an object property— `person1.sayHello()`— `this` is set to the object we got the function from (`person1`), which is why `person1.sayHello()` uses the name "Alice" and `person2.sayHello()` uses the name "Bob". But if we call it other ways, `this` is set differently: calling `this` from a variable— `helloFunction()`— sets `this` to the global object (`window`, on browsers). Since that object (probably) doesn't have a `firstName` property, we end up with "Hello, I'm undefined". (That's in loose mode code; it would be different [an error] in strict mode, but to avoid confusion we won't go into detail here.) Or we can set `this` explicitly using `Function#call` (or `Function#apply`), as shown at the end of the example.

Note: See more about `this` on Function#call and Function#apply

Inheritance

Inheritance is a way to create a class as a specialized version of one or more classes (*JavaScript only supports single inheritance*). The specialized class is commonly called the *child*, and the other class is commonly called the *parent*. In JavaScript you do this by assigning an instance of the parent class to the child class, and then specializing it. In modern browsers you can also use Object.create to implement inheritance.

Note: JavaScript does not detect the child class `prototype.constructor` (see Object.prototype), so we must state that manually. See the question "Why is it necessary to set the prototype constructor?" on Stackoverflow.

In the example below, we define the class `Student` as a child class of `Person`. Then we redefine the `sayHello()` method and add the `sayGoodBye()` method.

```javascript
// Define the Person constructor
var Person = function(firstName) {
  this.firstName = firstName;
};

// Add a couple of methods to Person.prototype
Person.prototype.walk = function(){
  console.log("I am walking!");
};

Person.prototype.sayHello = function(){
  console.log("Hello, I'm " + this.firstName);
};

// Define the Student constructor
function Student(firstName, subject) {
  // Call the parent constructor, making sure (using Function#call)
  // that "this" is set correctly during the call
  Person.call(this, firstName);

  // Initialize our Student-specific properties
  this.subject = subject;
}

// Create a Student.prototype object that inherits from Person.prototype.
// Note: A common error here is to use "new Person()" to create the
// Student.prototype. That's incorrect for several reasons, not least
// that we don't have anything to give Person for the "firstName"
// argument. The correct place to call Person is above, where we call
// it from Student.
Student.prototype = Object.create(Person.prototype); // See note below

// Set the "constructor" property to refer to Student
Student.prototype.constructor = Student;

// Replace the "sayHello" method
Student.prototype.sayHello = function(){
  console.log("Hello, I'm " + this.firstName + ". I'm studying "
              + this.subject + ".");
};

// Add a "sayGoodBye" method
Student.prototype.sayGoodBye = function(){
  console.log("Goodbye!");
};

// Example usage:
var student1 = new Student("Janet", "Applied Physics");
student1.sayHello();   // "Hello, I'm Janet. I'm studying Applied Physics."
student1.walk();       // "I am walking!"
student1.sayGoodBye(); // "Goodbye!"

// Check that instanceof works correctly
console.log(student1 instanceof Person);  // true
console.log(student1 instanceof Student); // true

```

Regarding the `Student.prototype = Object.create(Person.prototype);` line: On older JavaScript engines without `Object.create`, one can either use a "polyfill" (aka "shim", see the linked article), or one can use a function that achieves the same result, such as:

```
function createObject(proto) {
    function ctor() { }
    ctor.prototype = proto;
    return new ctor();
}

// Usage:
Student.prototype = createObject(Person.prototype);
```

Note: See Object.create for more on what it does, and a shim for older engines.

Making sure that `this` points to the right thing regardless of how the object is instantiated can be difficult. However, there is a simple idiom to make this easier.

```
var Person = function(firstName) {
  if (this instanceof Person) {
    this.firstName = firstName;
  } else {
    return new Person(firstName);
  }
}
```

Encapsulation

In the previous example, `Student` does not need to know how the `Person` class's `walk()` method is implemented, but still can use that method; the `Student` class doesn't need to explicitly define that method unless we want to change it. This is called **encapsulation**, by which every class packages data and methods into a single unit.

Information hiding is a common feature in other languages often as private and protected methods/properties. Even though you could simulate something like this on JavaScript, this is not a requirement to do Object Oriented programming.[2]

Abstraction

Abstraction is a mechanism that allows you to model the current part of the working problem, either by inheritance (specialization) or composition. JavaScript achieves specialization by inheritance, and composition by letting class instances be the values of other objects' attributes.

The JavaScript Function class inherits from the Object class (this demonstrates specialization of the model) and the `Function.prototype` property is an instance of `Object` (this demonstrates composition).

```
var foo = function () {};

// logs "foo is a Function: true"
console.log('foo is a Function: ' + (foo instanceof Function));

// logs "foo.prototype is an Object: true"
console.log('foo.prototype is an Object: ' + (foo.prototype instanceof Object));
```

Polymorphism

Just as all methods and properties are defined inside the prototype property, different classes can define methods with the same name; methods are scoped to the class in which they're defined, unless the two classes hold a parent-child relation (i.e. one inherits from the other in a chain of inheritance).

Notes

These are not the only ways you can implement object-oriented programming in JavaScript, which is very flexible in this regard. Likewise, the techniques shown here do not use any language hacks, nor do they mimic other languages' object theory implementations.

There are other techniques that make even more advanced object-oriented programming in JavaScript, but those are beyond the scope of this introductory article.

References

1. Wikipedia. "Object-oriented programming"
2. Wikipedia. "Encapsulation (object-oriented programming)"

See also

- `Function.prototype.call()`
- `Function.prototype.apply()`
- `Object.create()`
- Strict mode