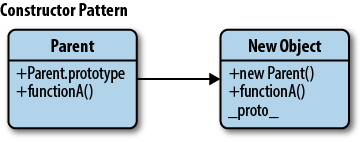
The Constructor Pattern

In classical object-oriented programming languages, a constructor is a special method used to initialize a newly created object once memory has been allocated for it. In JavaScript, as almost everything is an object, we’re most often interested in *object* constructors.

Object constructors are used to create specific types of objects—both preparing the object for use and accepting arguments a constructor can use to set the values of member properties and methods when the object is first created ([Figure 9-1](https://www.safaribooksonline.com/library/view/learning-javascript-design/9781449334840/ch09s01.html#constructor)).



*Figure 9-1. Constructor pattern*

Object Creation

The two common ways to create new objects in JavaScript are as follows:

*// Each of the following options will create a new empty object:*

**var** newObject = {};

*// or which is a shorthand for the object constructor*

**var** newObject = **new** Object();

Where the Object constructor creates an object wrapper for a specific value, or where no value is passed, it will create an empty object and return it.

There are then four ways in which keys and values can be assigned to an object:

*// ECMAScript 3 compatible approaches*

*// 1. Dot syntax*

*// Set properties*

newObject.someKey = "Hello World";

*// Get properties*

**var** key = newObject.someKey;

*// 2. Square bracket syntax*

*// Set properties*

newObject["someKey"] = "Hello World";

*// Get properties*

**var** key = newObject["someKey"];

*// ECMAScript 5 only compatible approaches*

*// For more information see: http://kangax.github.com/es5-compat-table/*

*// 3. Object.defineProperty*

*// Set properties*

Object.defineProperty( newObject, "someKey", {

value: "for more control of the property's behavior",

writable: **true**,

enumerable: **true**,

configurable: **true**

});

*// If the above feels a little difficult to read, a short-hand could*

*// be written as follows:*

**var** defineProp = **function** ( obj, key, value ){

config.value = value;

Object.defineProperty( obj, key, config );

};

*// To use, we then create a new empty "person" object*

**var** person = Object.create( **null** );

*// Populate the object with properties*

defineProp( person, "car", "Delorean" );

defineProp( person, "dateOfBirth", "1981" );

defineProp( person, "hasBeard", **false** );

*// 4. Object.defineProperties*

*// Set properties*

Object.defineProperties( newObject, {

"someKey": {

value: "Hello World",

writable: **true**

},

"anotherKey": {

value: "Foo bar",

writable: **false**

}

});

*// Getting properties for 3. and 4. can be done using any of the*

*// options in 1. and 2.*

As we will see a little later in the book, these methods can even be used for inheritance, as follows:

*// Usage:*

*// Create a race car driver that inherits from the person object*

**var** driver = Object.create( person );

*// Set some properties for the driver*

defineProp(driver, "topSpeed", "100mph");

*// Get an inherited property (1981)*

console.log( driver.dateOfBirth );

*// Get the property we set (100mph)*

console.log( driver.topSpeed );

Basic Constructors

As we saw earlier, JavaScript doesn’t support the concept of classes, but it does support special constructor functions that work with objects. By simply prefixing a call to a constructor function with the keyword new, we can tell JavaScript we would like the function to behave like a constructor and instantiate a new object with the members defined by that function.

Inside a constructor, the keyword this references the new object that’s being created. Revisiting object creation, a basic constructor may look as follows:

**function** Car( model, year, miles ) {

**this**.model = model;

**this**.year = year;

**this**.miles = miles;

**this**.toString = **function** () {

**return** **this**.model + " has done " + **this**.miles + " miles";

};

}

*// Usage:*

*// We can create new instances of the car*

**var** civic = **new** Car( "Honda Civic", 2009, 20000 );

**var** mondeo = **new** Car( "Ford Mondeo", 2010, 5000 );

*// and then open our browser console to view the*

*// output of the toString() method being called on*

*// these objects*

console.log( civic.toString() );

console.log( mondeo.toString() );

The above is a simple version of the constructor pattern, but it does suffer from some problems. One is that it makes inheritance difficult and the other is that functions such as toString() are redefined for each of new object created using the Car constructor. This isn’t optimal, as the function should ideally be shared between all instances of the Car type.

Thankfully, as there are a number of both ES3- and ES5-compatible alternatives to constructing objects, it’s trivial to work around this limitation.

Constructors with Prototypes

Functions in JavaScript have a property called a *prototype*. When we call a JavaScript constructor to create an object, all the properties of the constructor’s prototype are then made available to the new object. In this fashion, multiple Car objects can be created that access the same prototype. We can thus extend the original example as follows:

**function** Car( model, year, miles ) {

**this**.model = model;

**this**.year = year;

**this**.miles = miles;

}

*// Note here that we are using Object.prototype.newMethod rather than*

*// Object.prototype so as to avoid redefining the prototype object*

Car.prototype.toString = **function** () {

**return** **this**.model + " has done " + **this**.miles + " miles";

};

*// Usage:*

**var** civic = **new** Car( "Honda Civic", 2009, 20000 );

**var** mondeo = **new** Car( "Ford Mondeo", 2010, 5000 );

console.log( civic.toString() );

console.log( mondeo.toString() );

A single instance of toString() will now be shared between all Car objects.