Dojo from the ground up, Part 2: Mastering object-oriented development with Dojo

Joe Lennon
February 01, 2011

The Dojo toolkit enables web application developers to create Rich Internet Applications by offering a wide variety of features that save development time and effort. From DOM helpers and Asynchronous JavaScript and XML (Ajax) to a full-blown widget library and object-orientation features, Dojo includes virtually everything you need to build large-scale Ajax-powered web applications. If the functions you are looking for are not included in Dojo itself, it's likely that you can find them in DojoX, a repository of extensions and experimental features that are not included in the Base or Core modules of the toolkit. In Part 2 of this three-part series on developing rich web-based applications using the Dojo toolkit, you will learn about JavaScript's object-orientation features, and how they differ from a traditional class-based object-oriented programming language. You will then see how Dojo bridges this gap by offering a class-based system of its own.

View more content in this series

What is object-oriented development?

Develop skills on this topic
This content is part of a progressive knowledge path for advancing your skills. See Get started with Dojo development

Object-oriented programming (OOP) is a software development paradigm that is based on the definition of data structures called objects, which consist of data properties and functions. These properties (member variables) and functions (or methods) define the potential interaction that a piece of software can perform with that object. The primary benefit of OOP is that it helps with code re-use and maintenance by making it easier to organize your code.

Basics of object-orientation
The basic premise of object-orientated programming is that you create objects in your software that define a series of properties that apply to that object and a series of methods or functions that can retrieve or modify the object's properties. A simple example of an object might be a car. The data properties associated with a car might include its manufacturer, model number, registration number, color, cubic capacity, and so on. The methods you might find a car object providing include accelerate, brake, change gear, turn, stop, and so on. In OOP, the idea is that you define
the basic properties and methods that are common to all cars, then each individual car will take the form of that definition, albeit with different values to one another. As you will see later in this article, there are different types of approaches to object-orientation in software development.

Common OOP terminology

To get the most out of this article, you should be at least familiar with object-oriented programming and its concepts. Following are basic descriptions of some common terminology used when discussing object-oriented development. It is worth pointing out that not all types of OOP incorporate each of these aspects; for example, there are no classes in prototype-based object languages such as JavaScript.

Class

In class-based object-oriented development, a class defines the different properties and functions that make up an object. The class defines the template from which objects are generated, so they should define the common attributes and actions that these objects can adhere to. A class is typically made up of member variables and methods.

Member variables

A member variable of an object is an attribute or property of the object. In the car example I offered earlier, these attributes may include things like the car's manufacturer, model, color, cubic capacity, and so on.

Methods

A method is an action that an object can perform. For example, a car can accelerate, brake, turn, and so on. Often, a method will modify the value of a member variable. For example, when a car object accelerates using the accelerate method its current speed property will increase. Many objects will have a special method called a constructor, which is always called instantly whenever an object is created.

Instance or object

An instance or object is the actual object itself, rather than a template for defining an object. For example, you might have an object called myCar, which has the properties and methods of a car template. In the instance of an object, the properties would actually have values. So, for example, myCar might have a color property with a value of silver and a cubic capacity property of 2500. The current values of an object's properties are referred to as its state, and its state can change throughout the life cycle of the object.

Inheritance

In class-based OOP, inheritance is the process in which a subclass or child class inherits the member variables and methods of its superclass or parent class. In addition to inheriting these attributes and actions, a subclass can define member variables and methods of its own and provide default values for properties of the parent class. For example, you might have a FourByFour class, which is a subclass of the Car class. This subclass may set the drivetrain attribute of its superclass to default to 4WD (four wheel drive). In addition, it may define another
attribute with the name transfer case that only applies to 4x4 vehicles as well as a method that lets you change the low range gear that is not available on regular vehicles.

**Encapsulation**

In class-based OOP, member variables are often declared as private so they cannot be accessed or modified outside the confines of the class itself. There are special methods known as modifiers that let you define methods that will either retrieve or modify the value of private member functions in the class. These methods (often referred to as getters and setters) allow a programmer to use information hiding to only make certain properties accessible to applications and other classes. This technique is often referred to as encapsulation.

**Abstraction**

Abstraction is the process of reducing the complexity of your objects by only defining those properties and methods that are significant to an object in its current context. For example, when defining a `Car` class, you could abstract the class further to a `Vehicle` class by defining all the properties that a car has that are also common to other types of vehicles such as vans, trucks, motorcycles, and so forth. The `Car` class would then inherit these properties from the `Vehicle` class, as would a `Motorcycle` class or `Van` class.

**Polymorphism**

In the context of OOP, polymorphism means that a subclass can inherit methods from its superclass without having to provide the same implementation for everything. Take an example where you may have two subclasses of the `Car` class, one for cars with automatic transmission (let's call it `ATCar`) and another for cars with manual transmission (say, `MTCar`). All `Car` objects can accelerate, so both `ATCar` and `MTCar` would inherit the `accelerate` method from the parent class. However, in an `ATCar` your `accelerate` method would automatically call the change gear method when the engine reaches a certain level of RPM. As a result, you override the parent class’ definition of the `accelerate` method in the `ATCar` subclass, while in `MTCar` the method would remain the same as the one that it inherits from the `Car` class.

This article is not meant as an all-encompassing guide to object-oriented programming. If you are uncomfortable with the concepts outlined above, it would be well worth your while jumping to Related topics and reading more about OOP before continuing this article.

**Object-oriented JavaScript**

Many of the concepts presented in the previous section are exclusive to a specific OOP paradigm referred to as class-based object-oriented programming. However, not all programming languages adhere to this paradigm. Another common type of OOP is prototype-based object-orientation, and this is the paradigm used in the JavaScript language. In this section, you learn a little more about JavaScript's implementation of object orientation, an example of how to use it, and some stumbling blocks that you may find, particularly if you come from a class-based OOP background.

**JavaScript is not just a basic scripting language**

When JavaScript first gained popularity, it was primarily used as a means of performing simple tricks on basic web pages. The majority of the people who used JavaScript were not software
developers, they were graphic or web designers who may have had a lot of experience with HTML, but had little or no knowledge of programming languages. HTML on its own is quite limited in terms of allowing someone to produce dynamic effects, and this is where JavaScript came into play. However, rather than actually learning how to program in JavaScript, the majority of designers would just find the snippet of code they needed, learn enough to know how to tweak it to their own specific needs, and then use that small piece of code. When the designer goes through this process several times, they can feel that they have an acceptable level of knowledge of JavaScript.

In the early days of the web, JavaScript was quite limited. However, it has grown into a mature and full-featured programming language, which is no longer just used to write simple tricks for websites, but to power entire rich Internet applications. In fact, JavaScript as a language is also used today in various other ways. The CouchDB document-oriented database management system, for example, uses JavaScript functions to query data in the database.

Many Java developers (and developers in other more traditional programming languages) may look down on JavaScript and see it as a basic scripting language. Although JavaScript has evolved to become very powerful, the mentality that it is just used to do tricks on websites remains. This is because most web developers who use JavaScript will use libraries and frameworks such as jQuery, Prototype, or Dojo to take the pain out of writing JavaScript for them. In fact, many web developers could be considered experts in writing jQuery applications, but actually have little expertise in JavaScript itself. The thing that many of these developers are missing is that vanilla JavaScript is actually very powerful and includes object-orientation features out of the box. In this section, you learn about these features.

Prototype-based OOP

JavaScript implements a type of object-oriented programming different to that used in Java™ code. Whereas Java programming is based on a class-based OOP model, JavaScript is based on a class-less-based OOP model commonly known as prototype-based object-orientation. Rather than defining a set of class templates from which objects are created, objects are simply declared when needed. When an object needs to inherit features from another object, it can simply clone the features of a prototype object. One of the key advantages of prototypal OOP is that an object prototype can be modified at run time, meaning that the definition of the structure of objects is not strict. Most class-based OOP programming languages do not allow the class to be dynamically altered at run time (although there are a few exceptions such as Perl, Python, and Ruby).

In the next section, you learn how to work with OOP in vanilla JavaScript using the prototype model.

Example of basic object orientation in vanilla JavaScript

In Part 1 of this article series, you learned about the Firebug plug-in for Firefox and how you can use the console to run JavaScript code without having to edit, save, and run files and perform events on a web page. Open Firefox and launch Firebug. It doesn't matter what web page you are currently on, as this example does not require any particular JavaScript libraries.
Rather than declaring classes in JavaScript, you create a function that will serve as an object prototype. You can then create an instance of this object using the new keyword. For an example of this, enter the code in Listing 1 into the Firebug console:

**Listing 1. A basic object prototype in JavaScript**

```javascript
function Car() { }
var myCar = new Car();
console.log(myCar);
```

This should produce the following output in the console log: `object {}`.

You can actually click on this output and it will take you to another part of Firebug that lets you inspect the various properties of an object. If you click on this particular object, you'll see a message "There are no properties to show for this object."

What this code is actually doing is defining an object function `Car()`. It then instantiates an object using the new operator, and, finally, it outputs the object to the Firebug console where you can inspect it further. This object is hardly going to set the world right, however, so let's make it a bit more interesting (see Listing 2).

**Listing 2. Defining a member variable in your object**

```javascript
function Car() { }
Car.prototype.current_speed = 0;
var myCar = new Car();
console.log(myCar);
```

This time, the console log will output something a bit more interesting: `object { current_speed=0 }`.

Clicking on the output will bring you back to the DOM inspector window, where you can see the property `current_speed` and a value of 0. Any objects created using the `car` prototype function will, by default, have a `current_speed` property with a default value of zero.

A car is pretty much useless if it can't accelerate, so let's add a method to the prototype (see Listing 3).

**Listing 3. Adding a method to the prototype**

```javascript
function Car() { }
Car.prototype.current_speed = 0;
Car.prototype.accelerate = function(increment) {
    this.current_speed += increment;
}
var myCar = new Car();
myCar.accelerate(30);
myCar.accelerate(20);
console.log(myCar);
```
The console output should produce the following: `Object { current_speed=50 }`.

Clicking the output will show you not only the `current_speed` property, but also the new `accelerate` method that is available to the object. In the example in Listing 3, you can see how to call an object's method: you use dot notation like `object.method(arg[0], arg[1], ..., arg[N]);` in this case, it's `myCar.accelerate(20)`. This same notation can be used to access any particular property of an object. Change the line `console.log(myCar)` to `console.log(myCar.current_speed)` and you should see the console output show the value `50` instead of a representation of the actual object itself.

Next, let's investigate how to implement a constructor function in JavaScript. A constructor is a function that is called immediately after an object is instantiated. When you create the object function, this doubles as a constructor function. Listing 4 is an example of this in action.

### Listing 4. Adding a constructor function body

```javascript
function Car(reg_no) {
  this.reg_no = reg_no;
  console.log('Car with registration no. ' + this.reg_no + ' created.);
}

Car.prototype.reg_no = '';
Car.prototype.current_speed = 0;
Car.prototype.accelerate = function(increment) {
  this.current_speed += increment;
}

var myCar = new Car('10C500');
myCar.accelerate(30);
myCar.accelerate(20);
console.log(myCar.current_speed);
```

In Listing 4, the `Car`'s constructor function accepts a single argument for its registration number. It then sets the object instance's registration number to the argument value and outputs a message to the Firebug console to confirm that the instance has been created. You should see the output in Listing 5 in the Firebug console output window.

### Listing 5. Output

```
Car with registration no. undefined created.
50
```

To demonstrate how you can implement inheritance on an object prototype, let's create a more advanced `car` prototype, complete with methods to accelerate, decelerate, and change gears (see Listing 6).
Listing 6. A more complete Car prototype

```javascript
function Car(reg_no) {
    this.reg_no = reg_no;
}

Car.prototype.reg_no = '';
Car.prototype.current_speed = 0;
Car.prototype.current_gear = 0;

Car.prototype.accelerate = function(increment) {
    this.current_speed += increment;
}

Car.prototype.decelerate = function(decrement) {
    this.current_speed -= decrement;
}

Car.prototype.increaseGear = function() {
    this.current_gear++;
}

Car.prototype.decreaseGear = function() {
    this.current_gear--;
}
```

Now let's create an inherited object prototype from this called ATCar, which describes a car with automatic transmission. This particular example is hardly the perfect car, as it implements gear changes based on speed rather than RPM, but it should help get the idea of inheritance and polymorphism in prototypal OOP across (see Listing 7).

Listing 7. An ATCar object prototype, inheriting from the Car

```javascript
function ATCar(reg_no) {
    Car.call(this, reg_no);
}

ATCar.prototype = new Car();
ATCar.prototype.constructor = ATCar;

ATCar.prototype.accelerate = function(increment) {
    Car.prototype.accelerate.call(this, increment);
    if(increment >= 10) this.increaseGear();
}

ATCar.prototype.decelerate = function(decrement) {
    Car.prototype.decelerate.call(this, decrement);
    if(this.current_speed === 0) this.current_gear = 0;
    else if(decrement >= 10) this.decreaseGear();
}
```

The first thing you'll notice is that the constructor uses the call function on the Car object prototype. This basically will call the constructor on the parent prototype definition whenever an ATCar is instantiated. This lets you ensure that you implement inheritance, where the ATCar prototype does not need to know the inner workings of the Car object prototype. Next, you'll see that you are setting the prototype property of the ATCar function to a new instance of the Car function. This basically tells JavaScript that you want to inherit the properties and methods of the Car prototype in your ATCar prototype.

In Listing 7, you are also overriding the accelerate and decelerate methods so that on automatic cars, acceleration and deceleration will also cause the gears to change automatically. In both of these methods, you are first calling the parent prototype's method so you don't have to re-
implement the actual acceleration, saving on repetitive code. This is not so important in this particular example as it is only one line of code, but imagine this was a complex function; repeating it would be painful!

Finally let’s see this example in practice (see Listing 8).

**Listing 8. Using the automatic car prototype**

```javascript
var myCar = new ATCar('10C500');
myCar.accelerate(30);
myCar.accelerate(20);
myCar.decelerate(5);

console.log(myCar);
```

This will output the new object you have created. Click on the object to view its detail. The result should look something like Figure 1.

**Figure 1. Firebug view of the myCar object properties**

![Firebug view of the myCar object properties](image)

This article’s primary objective is to show you how to use Dojo’s class-based OOP simulation features, and as such, this is only an introduction to vanilla JavaScript’s prototypal approach to object-orientation. For further information on prototype based OOP, see [Related topics](#).

**Problems with prototypal OOP**

Different people have different views on whether prototype-based object-oriented programming is better or worse than class-based OOP. Each argument for or against can be fairly easily countered based on your preferences as a programmer. One of the most common problems with prototype-based object orientation, particularly in JavaScript, is the lack of basic understanding that many developers have of it. This is gradually becoming less of an issue, however, as the number of developers who have a deep understanding of JavaScript is steadily increasing. With that said, many programmers who are familiar with class-based OOP languages such as the Java language will still prefer to stick to a class-based system, and Dojo provides some excellent features that simulate this type of system to let you code JavaScript in a similar fashion. The next section explains how to use these features of Dojo to create class-based applications in JavaScript.
Simulating class-based OOP with Dojo

Before discussing Dojo's class-based simulation further, it's important to note that at the end of the day, Dojo is still a JavaScript library. Java code and JavaScript are not the same; in fact, they are quite different. Dojo does not try to force JavaScript to act like Java code, but rather it allows you to work with JavaScript OOP in a way that is more familiar to Java (and other class-based OOP languages) developers, with the underlying structure still working in a prototypal fashion.

Creating classes with dojo.declare

To create a class using Dojo, you use the `dojo.declare` function. Let's create a `Car` class using this function now (see Listing 9).

Listing 9. Using dojo.declare to create a Car class

```javascript
dojo.declare("Car", null, {

});

var myCar = new Car();
console.log(myCar);
```

This is the basic shell for creating a class and instantiating an object of that class. The `dojo.declare` function accepts three arguments:

1. The class name
2. The superclass the class should inherit from
3. An object comprising any properties and methods for the class

In the example in Listing 9, you declared a class named `Car`, which does not inherit from any superclass and does not prescribe any member variables or methods. If you look at the object properties for the `myCar` object by clicking on the output in the Firebug console, you should see something similar to Figure 2.
As you can see, creating a class in Dojo will give any objects generated from the class some properties and methods by default. Your current class is not very interesting, so let's add some properties and methods and a constructor, like you did when demonstrating the prototype-oriented way of doing so in the previous section (see Listing 10).

**Listing 10. A more complete Car class**

```javascript
dojo.declare("Car", null, {
  reg_no: ",",
  current_speed: 0,
  current_gear: 0,
  constructor: function(reg_no) {
    this.reg_no = reg_no;
  },
  accelerate: function(increment) {
    this.current_speed += increment;
  },
  decelerate: function(decrement) {
    this.current_speed -= decrement;
  },
  increaseGear: function() {
    this.current_gear++;
  },
  decreaseGear: function() {
    this.current_gear--;
  }
});
```

As you can see, this style of class declaration is far easier to read and is more together than the vanilla JavaScript way of declaring an object prototype. Before moving on to inheritance, let's check that an object can be instantiated and the methods work (see Listing 11).
Listing 11. Using the Car class

```javascript
var myCar = new Car("10C500");
myCar.accelerate(30);
myCar.accelerate(20);
myCar.decelerate(5);
console.log(myCar.reg_no + " travelling at "+myCar.current_speed + " mph");
```

This produces the following output in the Firebug console: 10C500 travelling at 45 mph.

Inheritance and multiple inheritance

Next, create your automatic transmission subclass, like you did previously using vanilla JavaScript, but this time using the dojo.declare function (see Listing 12).

Listing 12. Creating the ATCar subclass using Dojo

```javascript
dojo.declare("ATCar", Car, {
    accelerate: function(increment) {
        this.inherited(arguments);
        if (increment >= 10) this.increaseGear();
    },
    decelerate: function(decrement) {
        this.inherited(arguments);
        if (decrement >= 10) this.decreaseGear();
    }
});
```

As you can see from Listing 12, the only thing that is provided in the subclass is any overridden or new properties and methods (in this case, you are just overriding the accelerate and decelerate methods to automatically change gears). Dojo takes care of calling the constructor in the superclass automatically. If you needed to add a constructor function, you can add a constructor function to the subclass, but you don't need to worry about calling the superclass constructor; this is done automatically. You'll notice that in both the overridden methods the line `this.inherited(arguments)` is called. This will basically call the same method in the superclass. This allows you to avoid rewriting the code to do the actual acceleration, and only facilitate the changing of gears as is provided by the automatic car.

Let's check out the new subclass in action (see Listing 13).

Listing 13. The new subclass in action

```javascript
var myCar = new ATCar("10C500");
myCar.accelerate(30);
myCar.accelerate(20);
myCar.decelerate(5);
console.log(myCar.reg_no + " travelling at "+myCar.current_speed + " mph in gear "+myCar.current_gear);
```

This produces the following output: 10C500 travelling at 45 mph in gear 2.

Dojo also includes support for multiple inheritance. Multiple inheritance allows a subclass to be derived from multiple parent classes, inheriting properties and methods from each of the parent
classes. Strictly speaking, only one of these parent classes is considered to be the superclass (the first one in the array), but the constructors of each derived class will be called in the sequence in which they reside in the array of parent classes.

To illustrate multiple inheritance, take the example of a Smartphone, which performs many functions other than just making and receiving phone calls and text messages (see Listing 14). Typically, it will also perform functions such as playing music, watching videos, and much more. Let's keep it simple for the purpose of the example and say that a Phone can make calls, a MediaPlayer can play videos, and a Smartphone can do both.

**Listing 14. Multiple inheritance in Dojo**

```javascript
dojo.declare("Phone", null, {
    phone_number: ",",
    minutes_remaining: 0,
    constructor: function(properties) {
        this.phone_number = properties.phone_number;
        this.minutes_remaining = properties.minutes_remaining;
        console.log("Phone "+this.phone_number+" powered on. You have "+this.minutes_remaining+" minute(s) remaining.");
    }
});

dojo.declare("MediaPlayer", null, {
    disk_space: 0,
    constructor: function(properties) {
        this.disk_space = properties.disk_space;
        this.songs = properties.songs;
        console.log("Media Player powered on. You have "+this.songs.length+" songs,
        with "+this.disk_space+" GB free space left.");
    }
});

dojo.declare("Smartphone", [Phone, MediaPlayer], {
    phone_id: ",",
    constructor: function(properties) {
        this.phone_id = properties.phone_id;
        console.log("Smartphone ID "+this.phone_id+" boot up complete.");
    }
});

var songs = [
    {artist:"U2",title:"Vertigo"},
    {artist:"Coldplay",title:"Yellow"}
];

var myPhone = new Smartphone({
    phone_number: "(555) 123-4567",
    minutes_remaining: 60,
    disk_space: 2.5,
    songs: songs,
    phone_id: "4345FD7JAP076"
});

console.log(myPhone);
```

The first thing worth pointing out here is how `dojo.declare` implements multiple inheritance. As you can see, rather than just passing the parent class as the second argument, an array of classes is passed. The constructors of these parent classes will be called automatically in the order they appear in the array. An important thing to note is that if each parent class constructor
accepts different arguments, Dojo will not be able to differentiate the arguments that should be passed to each constructor function. As a result, if you do require to pass different arguments to the different constructors, you should add the arguments as key/value pairs in a simple JavaScript and consume them in the constructors that way.

Listing 15 is the output that will appear in the Firebug console.

**Listing 15. Multiple Inheritance example output**

```
Phone (555) 123-4567 powered on. You have 60 minute(s) remaining.
Media Player powered on. You have 2 songs, with 2.5 GB free space left.
Smartphone ID 4345FDFD73AP076 boot up complete.
Object { phone_number="(555) 123-4567", more...}
```

Clicking on the link in the final line will show the properties of the `myPhone` object, which should look something like Figure 3.

**Figure 3. The myPhone object, inheriting from multiple parent classes**

In Figure 3, the different properties for the different classes are all present in the instantiated object of the Smartphone class. The `phone_number` and `mintues_remaining` properties of the Phone class are there, the `disk_space` and `songs` from the MediaPlayer class are there, and the `phone_id` member variable from the Smartphone subclass is also there. If any of these classes had methods, these methods would also show up here.

**Using dojo.mixin to improve the multiple inheritance example**

Dojo provides a nice utility function, `dojo.mixin`, which lets you mix objects by combining their properties from left to right (see Listing 16).

**Listing 16. A basic example of dojo.mixin**

```
var objA = { a: 1, b: 2 }; 
var objB = { b: 3, c: 4 }; 
dojo.mixin(objA, objB); 
console.log(objA);
```
After mixing `objB` into `objA`, the properties in `objA` will look like Figure 4.

**Figure 4. Mixing objects with `dojo.mixin`**

![Figure 4](image)

The `b` property that was initially set to 2 in `objA`, has been overwritten with the value 3 from `objB`. Also, the `c` property has been added. With the basic example finished, let's have a look at how you can use `dojo.mixin` in your multiple inheritance example.

When creating the `Phone` class in the previous example, you might recall the two lines in Listing 17 in the constructor for that class.

**Listing 17. Lines in the Phone class constructor**

```javascript
this.phone_number = properties.phone_number;
this.minutes_remaining = properties.minutes_remaining;
```

This wasn't too tedious as there were only 2 properties, but what if there were many more? Wouldn't it be a real pain to have to assign the properties in this way? That's where the `dojo.mixin` function really proves useful. Replace these two lines (and the similar lines in the `MediaPlayer` and `Smartphone` classes) with the following line: `dojo.mixin(this, properties);`

The result is the exact same as before, but there's no fumbling with the various properties that have been passed to the constructor. Neat, huh?

**Packaging and modular development in Dojo**

When working with large applications, it is likely that you will have large classes with many member variables and methods. If you come from a Java development background, you will more than likely follow the notion that different classes should reside in different files, grouped by packages. Classes are then "imported" when required for inheritance or other purposes, ensuring they are only loaded when necessary. With JavaScript, there is no such packaging and module system available out of the box, but thankfully, Dojo provides a solution.
For example, in the `Car` class example, using Java code you might store this in a package, as shown in Listing 18.

**Listing 18. Packaging classes in Java programming**

```java
package com.ibm.developerworks.dojoseries;
public class Car {
    //Car class code goes here
}
```

You can later import this class in your other Java classes, as shown in Listing 19.

**Listing 19. Importing classes in Java programming**

```java
package com.ibm.developerworks.dojoseries;
import com.ibm.developerworks.dojoseries.Car;
public class ATCar extends Car {
    //ATCar class code goes here
}
```

Dojo provides a similar packaging system by means of the `dojo.provide` and `dojo.require` functions. Let's take a look at how the Java code in Listing 19 might look in Dojo. First, let's look at the `Car` class in Listing 20.

**Listing 20. Packaging classes in Dojo**

```javascript
dojo.provide("com.ibm.developerworks.dojoseries.Car");
dojo.declare("com.ibm.developerworks.dojoseries.Car", null, {
    //Car class code goes here
});
```

As you may notice, this is quite similar to the Java code, albeit the entire package path for the class is given in the `dojo.provide` statement, rather than just the path to the containing package. The package path is important, as it also determines where Dojo will look for this class when it attempts to load it using `dojo.require`. So, for the example in Listing 20, the Car.js file should be stored in a relative path `com/ibm/developerworks/dojoseries/Car.js`. If it is not stored in this location, Dojo will not be able to load the class correctly when it needs to. Next, let's look at how you can import this class and create a subclass from it (see Listing 21).

**Listing 21. Importing classes in Dojo**

```javascript
dojo.provide("com.ibm.developerworks.dojoseries.ATCar");
dojo.require("com.ibm.developerworks.dojoseries.Car");
dojo.declare("com.ibm.developerworks.dojoseries.ATCar",
    com.ibm.developerworks.dojoseries.Car, {
    //ATCar class code goes here
});
```

You'll notice here that you are using the `dojo.provide` statement again to identify the path that this class can be loaded from. This particular class would be stored in the relative path `com/ibm/developerworks/dojoseries/ATCar.js`. Next, you use `dojo.require` to load the `car` class using its full package path. Finally, you declare the subclass, passing the full path to the parent class as the
second argument. Because this class has now been loaded by Dojo, it is available in the DOM and can be loaded directly by its name, without requiring it to be placed in a string.

Although you can technically have a different class name to the path provided in the dojo.provide statement (note that any loading of classes using dojo.require must use the fully qualified path set out in dojo.provide), it is highly recommended that you do not do this, as it will only lead to confusion.

The next section explains how you can use Dojo's build system to package your application, ensuring it runs at optimal performance and speed.

**Packing it all — using Dojo's build system**

When working with Dojo's object-orientation features, it is highly likely that you will start separating your classes out into different files to make code organization and management a much easier task. However, it is important to note that loading many small JavaScript files can have serious performance implications for your application. Each time a web browser needs to download and execute a JavaScript file it must make a separate HTTP request, wait for the server to respond, and then process that response. As a result, it is usually much faster to load a single large file than to load many small files.

The problem with using a single large file is that it can be a version control and code organization nightmare, not to mention the fact that it probably contains a large amount of code that isn't actually needed by parts of your application. The solution to this issue is to get the right balance between small file sizes and minimizing the number of HTTP requests your application makes.

This is where Dojo's build system comes into play. It lets you define layers, each of which combine the source code from several JavaScript source files and minify the resulting single file to keep the file size to a minimum. The minification process shrinks your code substantially by removing all unnecessary whitespace and comments, and by renaming local variables to shorter names and re-factoring their usage in functions where applicable. By using the build system, you can keep your source code organized for the purpose of development, but ensure that it works at optimum performance levels when deployed to a production environment.

A guide to using the Dojo build system would probably require an article of its own. Fortunately, the Dojo documentation features extensive coverage of this topic. For more information on how to use the build system, see Related topics.

**Conclusion**

In Part 2 of this three-part series on developing rich web-based applications using the Dojo toolkit, you learned about the basics of object-orientation, how JavaScript uses a prototype-based approach to OOP, and how the long-held perception among many traditional developers about JavaScript not being a powerful OOP-enabled language is wrong. You then learned about JavaScript's vanilla object-oriented features, how to define object prototypes, and how to implement inheritance. You discovered how you could use Firebug to test JavaScript code without
having to save code into a file. You then learned how to write JavaScript classes that have a style that is more familiar to Java developers using Dojo's `dojo.declare` function. You learned how to perform inheritance and multiple inheritance, and how `dojo.mixin` can make combining properties of objects a breeze. You also learned how Dojo's package and module functions let you separate your classes into individual source files, much like you'd expect in a traditional OO language like the Java language or C++. Finally, you discovered that the Dojo build system lets you develop in a manner that facilitates tidy code organization without impacting on performance by minimizing file sizes and HTTP requests in a production environment.

In the third and final part of this series, you will see how Dojo's widget platform, Dijit, can be used to create interactive and rich Internet applications. Dijit is built on top of Dojo's object-oriented features, so what you have learned in this part of the series should prove useful in the next part.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article source code</td>
<td>dojo2.source.zip</td>
<td>2KB</td>
</tr>
</tbody>
</table>
Related topics

- Visit the home page for the Dojo Toolkit.
- Check out some Dojo Toolkit Demos.
- Introducing The Dojo Toolkit: Read an excellent introduction to the Dojo Toolkit from the Opera developer site.
- Introduction to the Dojo toolkit, Part 1: Setup, core, and widgets: Read another fine introduction to the Dojo toolkit from Jawaworld.
- Dojo 1.5: Ready to power your web app: Learn about some of the new features in Dojo 1.5 from this article on Sitepen.
- "Internationalizing web applications using Dojo" (developerWorks, August 2008): Discover a way to perform native language support in the context of web sites and web applications using the i18n feature of the Dojo toolkit.
- "Consuming web services with the Dojo Toolkit" (developerWorks, September 2010): Learn how to consume services using the Dojo Toolkit to enable Ajax on a web page.
- Learn about Prototypal Inheritance in JavaScript from Douglas Crockford.
- Read Nathan Toone’s entry on Understanding dojo.declare, dojo.require, and dojo.provide from DojoCampus.org.
- Read Dojo Confessions (Or: How I gave up my jQuery Security Blanket and Lived to Tell the Tale) by Rebecca Murphey to learn how the author made the switch from jQuery to Dojo.
- Enterprise Dojo by Dan Lee shows you how to write highly cohesive code with JavaScript using Dojo.
- Dojo: Using the Dojo JavaScript Library to Build Ajax Applications by James E. Harmon from Addison-Wesley Professional.
- Getting StartED with Dojo by Kyle Hayes and Peter Higgins from friends of ED.
- "Writing a custom Dojo application"(developerWorks, December 2008): Find out much more about Dojo in this developerWorks article.
- "Develop HTML widgets with Dojo" (developerWorks, October 2006) explores Dojo’s extensibility.
- "Using the Dojo Toolkit with WebSphere Portal"(developerWorks, November 2007) describes how to install, configure, use, and leverage the Dojo Toolkit in WebSphere Portal applications.
- Download the Dojo Toolkit. Version 1.5 was used in this article.
- Access the Dojo Toolkit API documentation.
- Get Firefox.
- Get Firebug.
- IBM - Dojo Extension sample: The Dojo Extension Feature Set can be used to enable a IBM WebSphere Portlet Factory model to leverage functionality provided by the Dojo JavaScript Toolkit.