1
Introduction

What do Yahoo! Maps, Google Maps, Yahoo! Mail, My Yahoo!, Gmail, Digg, YouTube and a plethora of other popular "Web 2.0" applications have in common? They all offer rich and responsive user interfaces, heavily employing code written in the JavaScript language. JavaScript started with simple one-liners embedded in HTML, but is now used in much more sophisticated ways. Developers leverage the object-oriented nature of the language to build scalable code architectures made up of reusable pieces. JavaScript provides behavior, the third pillar in today's paradigm that sees web pages as consisting of three clearly distinguishable parts: content (HTML), presentation (CSS), and behavior (JavaScript).

JavaScript programs run inside a host environment. The web browser is the most common environment, but it is not the only one. Using JavaScript, you can create all kinds of widgets, application extensions, and other pieces of software. Learning JavaScript is a pretty good deal: you learn one language and can then code all kinds of different applications.

This book is about JavaScript and focuses on its object-oriented nature. The book starts from zero, and does not assume any prior programming knowledge. Although there is one chapter dedicated to the web browser environment, the rest of the book is about JavaScript in general, so is applicable to all environments.

Let's start with the first chapter, which gives you an overview of the story behind JavaScript. It also introduces the basic concepts you'll encounter in discussions on object-oriented programming.

A Bit of History

Initially, the Web was conceived as a collection of static HTML documents, tied together with hyperlinks. Soon, as the Web grew in popularity and size, the webmasters who were creating static HTML web pages felt they needed something more. They wanted the opportunity for richer user interaction, mainly driven by desire to save server round-trips for simple tasks such as form validation. Two options came up: Java applets (they failed) and LiveScript, which was conceived by Netscape in 1995 and later included in the Netscape 2.0 browser under the name of JavaScript.

The ability to alter otherwise static elements of a web page was very well received and other browsers followed suit. Microsoft's Internet Explorer (IE) 3.0 shipped with JScript, which was a copy of the same language plus some IE-specific features. Eventually there was an effort to standardize the various implementations of the language and this is how ECMAScript (European Computer Manufacturers Association) was born. Today we have the standard, called ECMA-262, and JavaScript is just one implementation of this standard, albeit the most popular one.

For better or for worse, JavaScript's instant popularity happened during the period of the Browser Wars I (approximately 1996-2001). Those were the times of the initial Internet boom when the two major browser vendors—Netscape and Microsoft, were competing for market share. These two vendors were constantly adding more bells and whistles to their browsers and their versions of JavaScript. This situation, together with the lack of agreed-upon standards brought a lot of bad opinions of JavaScript. More often than not, development was a pain: you write a script while working in one browser. Once you're done with development, you test in the other browser and it simply doesn't work. At the same time, the browser vendors were busy adding features, but falling behind on providing proper development tools.

Browser vendors were introducing incompatibilities that annoyed the web developers, but this was only one part of the problem. The other part of the problem were the web developers themselves, who were adding too many features to their web pages. Developers were eager to make use of every new possibility that the browsers provided and ended up "enhancing" their web pages with things like animations in the status bar, flashing colors, blinking texts, shaking browser windows, snowflakes, objects stalking your mouse cursor, and so on, often at the expense of usable pages. These various ways to abuse JavaScript was the other reason why the language got its bad reputation. This caused "the real programmers" (developers with background in more established languages such as Java or C/C++) to dismiss JavaScript as nothing but a toy for front-end designers to play around with.

The JavaScript backlash caused some web projects to completely ban any client-side programming and trust only their predictable and reliable server. And really, why would you double the time to deliver a project and spend this additional time debugging problems with the different browsers?

The Winds of Change

Everything changed in the years following the end of the Browser Wars I. A number of processes reshaped the web development landscape in a very positive way.

- Microsoft won the war, and for about five years (which is more or less forever in Internet time), they stopped adding features to Internet Explorer and JScript. This allowed time for other browsers as well as developers to catch up and even surpass IE's capabilities.
- The movement for web standards was embraced by developers and browser vendors alike. Naturally, developers didn't like having to code everything two (or more) times to account for browsers' differences; therefore they liked the idea of having agreed-upon standards that everyone would follow. We're still far from being able to develop in a fully standards-compliant environment, but ideally, this will happen in the future.
- Developers and technologies matured and more people started caring about things like usability, progressive enhancement techniques, and accessibility.

In this healthier environment, developers started finding out new and better ways to use the instruments that were already available. After the public release of applications such as Gmail and Google Maps, which were rich on client-side programming, it became clear that JavaScript is a mature, unique in certain ways, and powerful prototypal object-oriented language. The best example of it's rediscovery is the wide adoption of the functionality provided by the XMLHttpRequest object, which was **once an IE-only innovation, but was then** implemented by most other browsers. XMLHttpRequest allows JavaScript to make HTTP requests and get fresh content from the server in order to update some parts of a page, without a full page reload. Due to the wide use of XMLHttpRequest, a new breed of desktop-like web applications, dubbed AJAX applications, was born.

The Present

An interesting thing about JavaScript is that it always runs inside a *host environment*. The browser is the most popular host environment, but it is not the only one. JavaScript can run on the server, on the desktop, and in rich media. Today, you can use JavaScript to do all of this:

- Create rich and powerful web applications (the kind of applications that run inside the web browser, such as Gmail)
- Write server-side code such as ASP scripts or, for example, code that is run using Rhino (a JavaScript engine written in Java)
- Create rich media applications (Flash, Flex) using ActionScript, which is based on ECMAScript

- Write scripts that automate administrative tasks on your Windows desktop, using Windows Scripting Host
- Write extensions/plugins for a plethora of desktop application such as Firefox, Dreamweaver, and Fiddler
- Create web applications that store information in an off-line database on the user's desktop, using Google Gears
- Create Yahoo! Widgets, Mac Dashboard widgets, or Adobe Air applications that run on your desktop

This is by no means an extensive list. JavaScript started inside web pages, but today it's safe to say it is practically everywhere.

The Future

We can only speculate what the future will be, but it's quite certain that it will include JavaScript. For quite some time JavaScript may have been underestimated and underused (or rather overused in the wrong ways), but every day we witness new uses of JavaScript in much more interesting and creative ways. Where they once wrote simple one-liners, often embedded in-line in HTML tag attributes (such as `onclick`), developers nowadays ship sophisticated, well-designed and architected, extensible applications, and libraries. JavaScript is indeed taken seriously and developers are starting to rediscover and enjoy its unique object-oriented features more and more.

Once listed in the "nice to have" sections of job postings, these days the knowledge of JavaScript is a yes/no factor when it comes to hiring web developers. Common job interview questions you can hear today include: "Is JavaScript an object-oriented language? Good. Now how do you implement inheritance in JavaScript?" After reading this book, you'll be prepared to ace your JavaScript job interview and even impress your interviewers with some bits that maybe they didn't know.

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, property
- Class
- Encapsulation

- Aggregation
- Reusability/inheritance
- Polymorphism

Let's take a closer look into each one of these concepts.

Objects

As the name *object-oriented* suggests, objects are quite 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 some more convoluted concept. Taking a common object like a cat for example, you can see that it has certain characteristics (color, name, weight) and can perform some actions (meow, sleep, hide, escape). The characteristics of the object are called *properties* in OOP and the actions are called *methods*.

There is also an analogy with the spoken language:

- Objects are most often named using nouns (book, person)
- Methods are verbs (read, run)
- Values of the properties are adjectives

Let's take, for example, the sentence "The black cat sleeps on my head". "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 my head" 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 the Birds class. In OOP, a class is a blueprint, or recipe for an object. Another name for "object" is "instance", so we say that the eagle is an instance of the 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 like C++ and Java. You should understand 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 me a new object called Bob which is of class Person". In a prototypal OO language, you'd say, "I'm going to take this object Person that I have lying around 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) both:

- Data (stored in properties) and
- 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 a user of the object, are given some interface to work with, such as buttons, the display, and so on. You use the interface in order to get the object to do something useful for you, like playing a song. Exactly how it 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 internal implementation of the object has 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*. The aggregation concept illustrates the ability to combine several objects into a new one. Aggregation is 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*. A personal computer is a very 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 the objects it consists of: the Monitor object, the Mouse object, the Keyboard object, and so on. Then you can dive deeper into each of the sub-objects. This way you are composing complex objects by assembling reusable parts.

To use another analogy, a Book object could can contain (aggregate) one or more author objects, a publisher object, several chapter objects, a table of contents, and so on.

Inheritance

Inheritance is a very elegant way to reuse code that has already been written. For example, you can have a generic object Person, which has properties such as name and date of birth, and that implements the functionality walk, talk, sleep, eat. Then you figure out that you need an object Programmer. You could re-implement all the methods and properties that Person has, but it would be smarter to just say that Programmer *inherits* Person, and save yourself some work. The Programmer object only needs to implement more-specific functionality, such as the method "write code", while reusing all of the Person's functionality.

In classical OOP, classes inherit from other classes, but in JavaScript, because 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 inherited a number of methods, can pick one or more methods and redefine them, customizing them for its own needs. This way the interface stays the same, 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 example above, we had a Programmer object that 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 our code, there's a variable called Bob and it so happens that we don't know if Bob is a Person, or a Programmer object. We 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

If you are new to the OO programming lingo and you're not sure you've fully grasped the concepts above, don't worry. We'll look at some code and you'll see that, although they may seem complicated when just talking about high-level concepts, things are much simpler in practice.

Thus said, let's rehash the concepts once more.

Feature	**Illustrates concept**
Bob is a man (an object).	objects
Bob's date of birth is June 1st, 1980, gender: male, hair: black.	properties
Bob can eat, sleep, drink, dream, talk and calculate his age.	methods
Bob is an instance of class Programmer.	class (in classical OOP)
Bob is based on another object, called Programmer.	prototype (in prototypal OOP)
Bob holds data (such as *birth date*) and methods that work with the data (such as *calculate age*).	encapsulation
We 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, we don't know, nor do we want to know.	information hiding
Bob is part of a Web Dev Team object, together with Jill, a Designer object and Jack, a Project Manager object.	aggregation, composition

Feature	Illustrates concept
Designer, Project Manager 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, method overriding

Setting up Your Training Environment

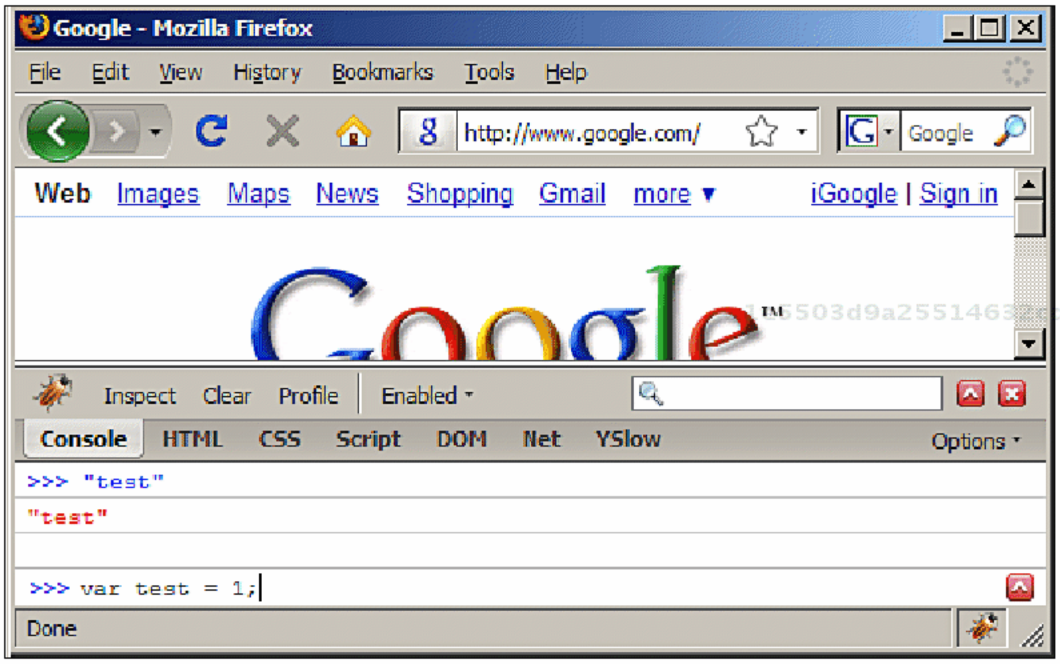
This book takes a "do it yourself" approach when it comes to writing code, because the author firmly believes that the best way to really learn a programming language is by writing code. So there's no cut-and-paste-ready code downloads, which you can simply put in your pages. On the contrary, you're expected to type in code, see how it works and then tweak it and play around with it. When trying out the code examples, you're encouraged to enter the code into Firebug's console. Let's see how you go about doing this.

Getting the Tools You Need

As a developer, you most likely already have Firefox installed and use it for your daily web browsing pleasure. If not, do yourself a favor and install it right now. It's free and runs on any platform—Windows, Linux, or Mac. Download it from `http://www.mozilla.com/firefox/`.

The Firefox browser is extensible and there are lots of useful extensions out there (all written in JavaScript!). A popular extension is Firebug—an indispensable tool for web development, with lots of useful features. Download Firebug from `http://www.getfirebug.com/`, install it, and try it out by starting Firefox and going to any page and pressing *F12* (on Windows) or clicking on the little bug icon at the bottom right corner of the Firefox screen. This will open the Firebug feature we're most interested in—the console.

Using the Firebug Console


Google - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.google.com/
Google
Web Images Maps News Shopping Gmail more ▼ iGoogle | Sign in
Inspect Clear Profile Enabled
Console HTML CSS Script DOM Net YSlow Options
Done

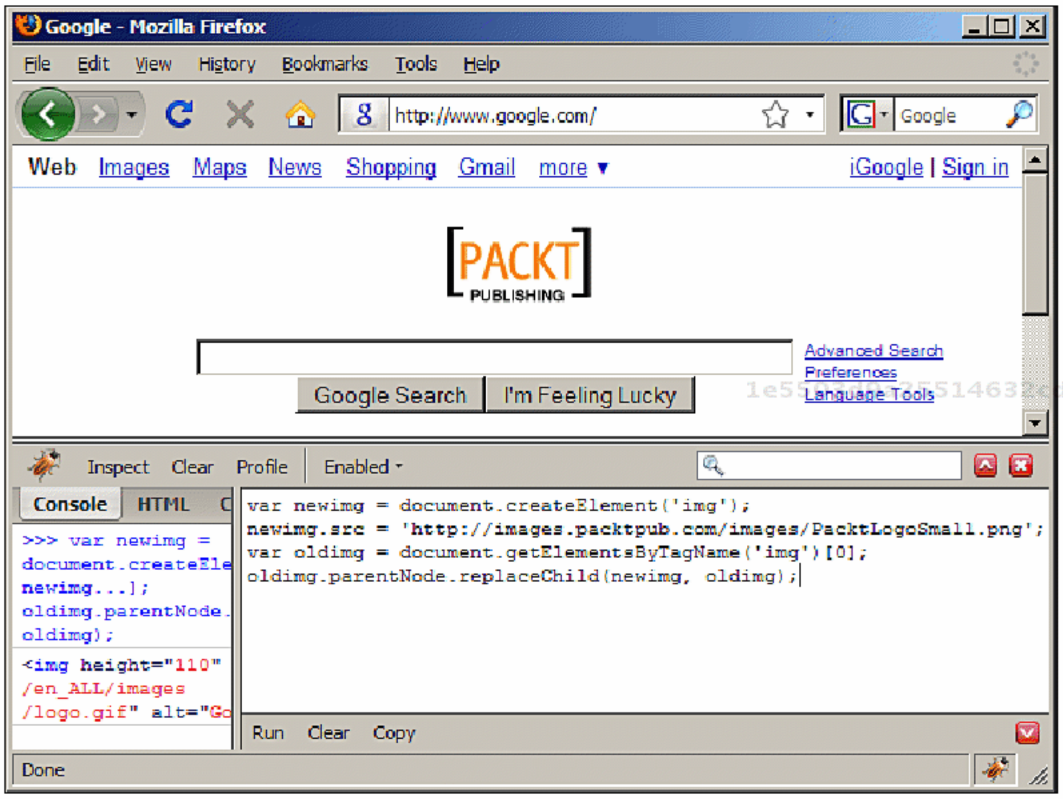
```
>>> "test"
"test"

>>> var test = 1;
```

You can type code directly into the Firebug console, and when you press *Enter*, the code is evaluated and executed. The return value of the code is printed in the console. The code is executed in the context of the currently-loaded page, so for example if you type `document.location.href` it will return the URL of the current page.

The console also has an auto-complete feature. It works similarly to the normal command line prompt in your operating system. If, for example, you type **docu** and hit the *Tab* key, **docu** will be auto-completed to **document**. Then if you type **.** (the dot operator), you can press *Tab* several times and it will iterate through all the available properties and methods you can call on the `document` object.

By using the *UP* and *DOWN* arrow keys, you can go through the list of already-executed commands and bring them back in the console.

The console gives you only one line to type in, but you can execute several JavaScript statements by separating them with semi-colons. If you need more space or more lines, you can open the console in a multi-line mode, by clicking the upward-facing arrow on the far right of the input line. An example of multi-line mode is shown in the next screenshot.


Google - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.google.com/
Google
Web Images Maps News Shopping Gmail more ▾ iGoogle | Sign in
PACKT PUBLISHING
Advanced Search
Preferences
Language Tools
Google Search I'm Feeling Lucky
Inspect Clear Profile Enabled
Console HTML
Run Clear Copy
Done

```
>>> var newimg =
document.createEle
newimg...];
oldimg.parentNode.
oldimg);
<img height="110"
/en_ALL/images
/logo.gif" alt="Go
```

```
var newimg = document.createElement('img');
newimg.src = 'http://images.packtpub.com/images/PacktLogoSmall.png';
var oldimg = document.getElementsByTagName('img')[0];
oldimg.parentNode.replaceChild(newimg, oldimg);
```

This example shows how you can use the console to type in some code that swaps the logo on the `google.com` home page with an image of your choice. As you see, you can test your JavaScript code live on any page.

One configuration option you should set in Firefox is the strictness of JavaScript warnings you will see in the console. This will help you make sure that the code you write is of better quality. Although warnings are not errors, you should aim at writing code that doesn't throw any warnings. For example using an undeclared variable is not an error, but it's not a good idea, so Firefox's JavaScript engine will generate a warning, which will be displayed in the console if the strict setting is turned on. To set the "strict" setting, do this:

1. Type **about:config** in Firefox's address bar.
2. Search for **strict** by typing it in the **Filter** field and pressing *Enter*.
3. Double-click the line that says **javascript.options.strict**. This should set its **Value** to **true**.

Summary

In this chapter, you learned about how JavaScript came to be and where it is today. You were also introduced to Object-oriented programming concepts and saw how JavaScript is not a classic OO language, but a prototypal one. Finally, you learned how to set up and use your training environment—the Firebug console. Now you're ready to dive into JavaScript and learn how to use its powerful OO features. For additional information on the topics discussed in this chapter, take a look at the following web pages.

- On the YUI Theater (`http://developer.yahoo.com/yui/theater/`), there are several talks by Douglas Crockford that are highly recommended. Part 1 of the "Theory of the DOM" talks about browser history, and Part 1 of "The JavaScript Programming Language" talks about history of JavaScript (amongst other things).
- For OOP concepts see the Wikipedia article (`http://en.wikipedia.org/wiki/Object-oriented_programming`) and Sun's Java documentation (`http://java.sun.com/docs/books/tutorial/java/concepts/index.html`), although the latter talks about OOP using classes.
- For examples of what's possible today with JavaScript, take a look at the Yahoo! Widgets page (`http://widgets.yahoo.com/`), Google Maps (`http://maps.google.com`), or the JavaScript version of the Processing visualization language (`http://ejohn.org/blog/processingjs/`).