Learn by doing: less theory, more results

JavaScript Testing

Test and debug JavaScript the easy way

Beginner's Guide

Liang Yuxian Eugene

[PACKT]
PUBLISHING

JavaScript Testing
Beginner's Guide

Test and debug JavaScript the easy way

Liang Yuxian Eugene

JavaScript Testing
Beginner's Guide

Copyright © 2010 Packt Publishing

Credits

Author
Liang Yuxian Eugene

Reviewers
Chetan Akarte

Kenneth Geisshirt

Stefano Provenzano

Aaron Saray

Mihai Vilcu

Acquisition Editor
Steven Wilding

Development Editor
Tarun Singh

Technical Editors
Paramanand N. Bhat

Pooja Pande

Copy Editors
Lakshmi Menon

Janki Mathuria

Editorial Team Leader
Akshara Aware

Project Team Leader
Priya Mukherji

Project Coordinator
Vincila Colaco

Indexer
Hemangini Bari

Proofreader
Dirk Manuel

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Author

Liang Yuxian Eugene enjoys solving difficult problems creatively in the form of building web applications by using Python/Django and JavaScript/JQuery. He also enjoys doing research related to the areas of recommendation algorithms, link analysis, data visualization, data mining, information retrieval, business intelligence, and intelligent user interfaces. He is currently pursuing two degrees, Business Administration and Computer Science at National Cheng Chi University (NCCU) at Taipei, Taiwan. Eugene has recently started a personal blog at `http://www.liangeugene.com`.

About the Reviewers

Chetankumar D. Akarte has been working in PHP, JavaScript and .Net for the last five years. He has worked extensively on both small scale and large scale PHP and .Net ecommerce, social networking, Wordpress and Joomla based web projects. Over the years, Chetan has been actively involved in the "Xfunda Developers Community". He has regularly blogged on Microsoft .NET technology at `http://www.tipsntracks.com`.

Chetan completed a Bachelor of Engineering degree in Electronics from the Nagpur University, India in 2006. He likes contributing to newsgroups, and forums. He has also written some articles for Electronics For You, DeveloperIQ, and Flash & Flex Developer's magazines.

Chetan lives in Navi Mumbai, India. You can visit his websites at `http://www.xfunda.com` and `http://www.tipsntracks.com`, or get in touch with him at `chetan.akarte@gmail.com`.

> I would like to thank my sister Poonam and brother-in-law Vinay for their consistent support and encouragement. I would also like to thank Packt Publishing for providing me with the opportunity to do something useful, and especially my Project Coordinator Vincila Colaco for all of the valuable support.

Kenneth Geisshirt is a chemist by education and a geek by nature. He has been programing for more than 25 years–the last six years as a subcontractor. In 1990 Kenneth first used free software, and in 1992 turned to Linux as a primary operating system (officially Linux user no. 573 at the Linux Counter). He has written books about Linux, PAM, and Javascript–and many articles on open source software for computer magazines. Moreover, Kenneth has been a technical reviewer of books on Linux network administration and the Vim editor.

Stefano Provenzano is an Italian senior consultant and professional software engineer. Stefano has worked on several projects in different fields of computer science—3D realtime engines for PC and Playstation games, visual simulation and virtual prototyping, web applications, and system integration. In 2006, Stefano started his own software development and consulting company, Shin Software. Currently, Stefano is developing CRM and INTRANET applications by using PHP and Javascript.

I want to thank my wife Irene and my little son Davide.

Aaron Saray found love when he was eight. It was in the shapely form of a Commodore 64. From then on, he continued to devote his time to various programing languages from BASIC to Pascal, PHP to Javascript, HTML to CSS. Aaron is both an author of a PHP Design Patterns book and a technical editor of other PHP and Javascript books. He has also worked as a professional in the Web Development field for almost a decade, and comes with a solid history to provide his vast experience to the review of this book. You can find more about his work at his technical blog by visiting `http://aaronsaray.com/blog`.

As each book project becomes complete, I learn more about my industry and myself. I want to specifically thank my best friend for consistently reminding me that life is always better with balance.

Mihai Vilcu has had exposure to top technologies in testing for both automated and manual testing. "Software testing excellence" is the motto that drives Mihai's career". This includes functional and non-functional testing. Mihai was also involved over several years in large scale testing projects.

Some of the applications covered by Mihai in his career include CRMs, ERPs, billing platforms, rating, collection and business process management applications.

As software platforms are used intensely in many industries, Mihai has performed testing in fields like telecom, banking, healthcare, software development, and others.

Feel free to contact Mihai for questions regarding testing on his email: `mvilcu@mvfirst.ro`, or directly on his website at `www.mvfirst.ro`.

Table of Contents

Preface

JavaScript is an important part of web development in today's Web 2.0 world. Although there are many JavaScript frameworks in the market, learning to write, test, and debug JavaScript without the help of any framework will make you a better JavaScript developer. However, testing and debugging can be time-consuming, tedious and painful. This book will ease your woes by providing various testing strategies, advice, and tool guides that will make testing smooth and easy.

This book is organized in an easy-to-follow, step-by-step tutorial style, in order to maximize your learning. You will first learn about the different types of errors that you will most often encounter as a JavaScript developer. You will also learn the most essential features of JavaScript through our easy-to-follow examples.

As you go along, you will learn how to write better JavaScript code through validation; learning how to write validated code alone will help you improve tremendously as a JavaScript developer and, most importantly, help you to write JavaScript code that runs better, faster, and with less bugs.

As our JavaScript program gets larger, we need better ways of testing our JavaScript code. You will learn about various testing concepts and how to use them in your test plan. After which, you will learn how to implement the test plan for your code. To accommodate more complex JavaScript code, you will learn more about the built-in features of JavaScript, in order to identify and catch different types of JavaScript error; such information helps to spot the root of the problem so that you can act on it.

Finally, you will learn how to make use of the built-in browser tools and other external tools to automate your testing process.

What this book covers

Chapter 1, *What is JavaScript Testing?*, covers JavaScript's role and the basic building blocks in web development, such as HTML and CSS. It also covers the types of errors that you will most commonly face.

Chapter 2, *Ad Hoc Testing and Debugging in JavaScript,* covers why we perform ad hoc testing for our JavaScript programs, and JavaScript's most commonly-used features, by writing a simple program, This program will be used as an example to perform ad hoc testing.

Chapter 3, *Syntax Validation,* covers how to write validated JavaScript. After completing this chapter, you will have improved your skills as a JavaScript developer and, at the same time, understood more about the role of validation in testing JavaScript code.

Chapter 4, *Planning to Test,* covers the importance of having a plan to test, and the strategies and concepts we can use when we are performing testing. This chapter also covers the various strategies and concepts for testing, and we will perform a simple test plan to see what it means to plan to test.

Chapter 5, *Putting the Test Plan Into Action,* follows Chapter 4, as we apply the simple test plan that we have developed. Most importantly, we will get our hands dirty by uncovering bugs, taking note of them and fixing the bugs by applying the theories that we learnt in Chapter 4.

Chapter 6, *Testing More Complex Code,* covers sophisticated ways to test our code. One way of testing the code is to use the built-in error objects provided by JavaScript. This chapter also covers how to use the console log, how to write your own messages, and how to trap your errors.

Chapter 7, *Debugging Tools,* addresses the point where our code gets too large and complex to be tested by using manual methods. We now engage the help of debugging tools provided by popular browsers in the market, including Internet Explorer 8, FireFox 3.6, Chrome 5.0, Safari 4.0 and Opera 10.

Chapter 8, *Testing Tools*, moves into how you can automate your testing by using testing tools that are free, cross-browser and cross-platform. It also covers how to test your interface, automate tests, and perform assertion and benchmarking tests.

What you need for this book

A basic text editor such as Notepad++.

Browsers like Internet Explorer 8, Google Chrome 4.0, Safari 4.0 and newer, FireFox 3.6.

JavaScript version 1.7 or later.

Other software covered includes Sahi, JSLitmus, QUnit.

Who this book is for

This book is for beginner JavaScript programmers or beginner programmers who may have little experience in using JavaScript, with HTML and CSS.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1

2. Action 2

3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short multiple choice questions intended to help you test your own understanding.

Have a go hero – heading

These sections set practical challenges and give you ideas for experimenting with what you have learned.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
<input type="submit" value="Submit"
        onclick="amountOfMoneySaved(moneyForm.money.value)" />
</form>
</body>
</html>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
function changeElementUsingName(a){
     var n = document.getElementsByName(a);
    for(var i = 0; i< n.length; i++){
    n[i].setAttribute("style","color:#ffffff");
    }
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking the **Next** button moves you to the next screen".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note via the **SUGGEST A TITLE** form on `www.packtpub.com`, or send an e-mail to `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

> **Downloading the example code for this book**
>
> You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

What is JavaScript Testing?

First of all, let me welcome you to this book. If you've picked up this book, I would assume that you are interested in JavaScript testing. You most probably have experienced JavaScript, and want to enhance your skills by learning how to test your JavaScript programs.

JavaScript is most often associated with the web browser and is one of the key tools for creating interactive elements on web pages. However, unlike server-side languages like PHP, Python and so on, JavaScript fails silently in general (although browsers like IE provides warning messages at times); there are no error messages to inform you that an error has occurred. This makes debugging difficult.

In general, we will be learning about the basic building blocks for JavaScript testing. This will include the basics of **HTML (Hyper-text Markup Language), CSS (Cascading Style Sheets)** *and JavaScript. After this, you will learn about various techniques to make HTML, CSS, and JavaScript work together; these techniques are the building blocks of what you are going to learn in other chapters.*

To be more specific, this is what we will learn about in this chapter:

- ◆ The basics of HTML, CSS, and JavaScript
- ◆ The syntax of HTML, CSS, and JavaScript
- ◆ How to select HTML elements by using CSS and JavaScript
- ◆ Why do web pages need to work without JavaScript?
- ◆ What is testing and why do you need to test?
- ◆ What is an error?
- ◆ Types of JavaScript errors

Examples shown in this chapter are simplistic—they are designed to allow you to see the major syntax and built-in methods or functions that are being used. In this chapter, there will be minimal coding; you will be asked to enter the code. After that, we'll briefly run through the code examples and see what is happening.

With that in mind, we'll get started right now.

Where does JavaScript fit into the web page?

Every web page consists of the following properties—content, appearance, and behavior. Each of these properties is controlled by Hyper Text Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript, respectively.

HTML Content

HTML stands for Hyper Text Markup Language. It is the dominant markup language for web pages. In general, it controls the content of a web page. HTML defines web pages (or HTML documents) through semantic markups such as `<head>`, `<body>`, `<form>`, and `<p>` to control headings, the body of a document, forms, paragraphs, and so on. You can see HTML as a way to describe how a webpage should look like.

HTML makes use of markup tags, and these tags usually come in pairs. The syntax of HTML is as follows:

`<name-of-html-tag>`some of your content enclosed here`</name-of-html-tag>`

Notice that the HTML tags are enclosed by angular brackets; the HTML tag pair starts off with `<name-of-html-tag>` and ends with `</name-of-html-tag>`. This second HTML tags are known as the closing tags and they have a forward slash before the HTML tag.

Some of the common HTML elements include the following:

- `<head> </head>`
- `<body> </body>`
- `<title> </title>`
- `<p> </p>`
- `<h1> </h1>`
- `<a> </a>`

For a complete list of html elements, please visit `http://www.w3schools.com/tags/default.asp`.

Time for action – building a HTML document

We are going to create an HTML document by making use of some of the HTML tags and syntax that we have seen above. (The example you see here can be found in the source code folder of `Chapter 1`, with the document titled `chapter1-common-html.html`)

1. Let's start by opening your favorite text editor or tool such as Microsoft Notepad, and creating a new document.

2. Enter the following code into your new document and save it.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>This is a sample title</title>
</head>
<body>
<h1>This is header 1</h1>
<h2>This is header 2</h2>
<h3>This is header 3</h3>
<p>This is a paragraph. It can be styled by CSS</p>
<hr>
<div style="position:absolute; background-color:black;
color:#ffffff;top:10px;right:10px;border:solid 3px yellow;
height:200px; width:200px;">Your content here</div>
<div>
    <div>I am enclosed within a <i>div</i> tag. And it can be
styled on a document level.
        <ol>
            <li>This is an ordered list and it is centered</li>
            <li>apple</li>
            <li>orange</li>
            <li>banana</li>
        </ol>
        <ul>
            <li>This is an unordered list. And it can be styled by
CSS.</li>
            <li>apple</li>
            <li>orange</li>
            <li>banana</li>
        </ul>
    </div>
    <div>I am enclosed within a <i>div</i> tag. And it can be
styled by CSS.
        <ol>
            <li>This is an ordered list and it is centered</li>
            <li>apple</li>
```

```
                <li>orange</li>
                <li>banana</li>
            </ol>
            <ul>
                <li>This is an unordered list. And it can be styled by
    CSS</li>
                <li>apple</li>
                <li>orange</li>
                <li>banana</li>
            </ul>
            <a  href="#">This is a link. And it can be styled by CSS
    </a>
        </div>
    </div>
    </body>
    </html>
```

3. Finally, open the document in your browser and you will see an example similar to the following screenshot:
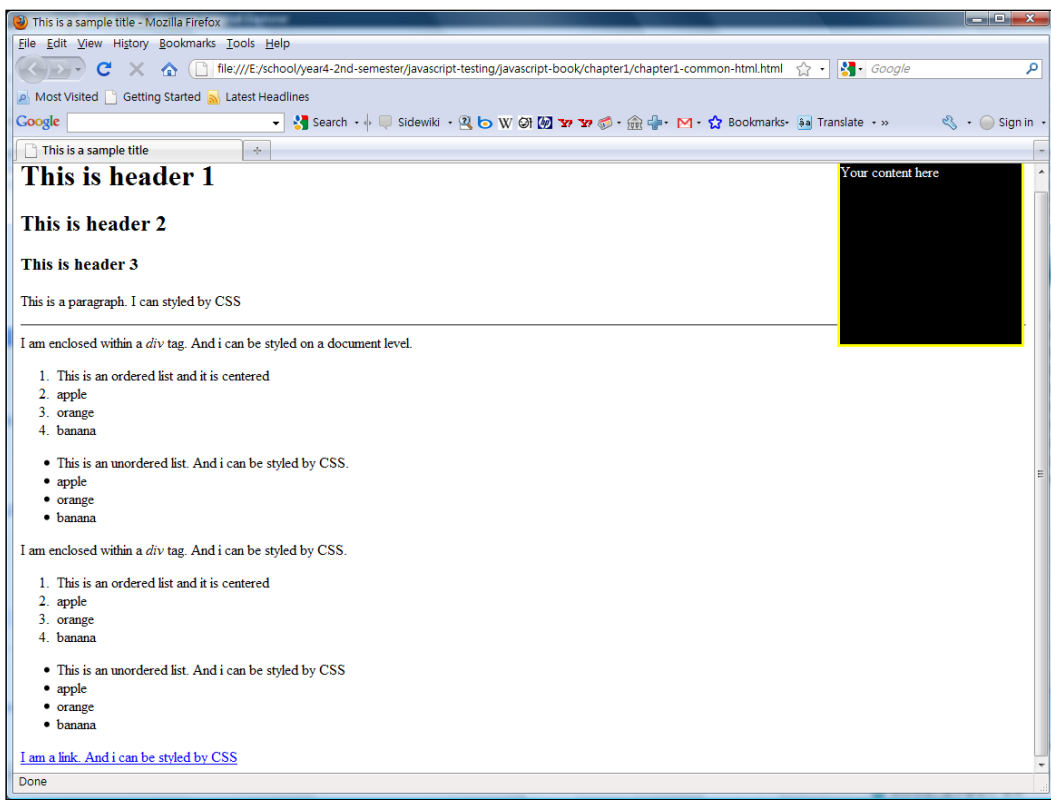
Take note of the black box on the upper-right corner. It is a simple example of CSS at work. This will be explained shortly.

What just happened?

You have just created an HTML document by using the more common HTML elements and HTML syntax.

Each HTML tag has a specific purpose, as you can see from the result in the browser. For example, you must have noticed that `<h1>This is header 1</h1>` produced the largest text in terms of font-size, `<h2>This is header 2</h2>` produced the second largest text in terms of font size, and so forth.

`<ol> </ol>` represents an ordered list, while `<ul> </ul>` stands for an unordered list (list with bullet points).

You should have noticed the use of `<div> </div>`. This is used to define a section within an HTML document. However, the effects and power of the `<div> </div>` can only be seen in the next part of this chapter.

But wait, it seems that I have not done a complete introduction of HTML. That's right. I have not introduced the various attributes of HTML elements. So let's have a quick overview.

Styling HTML elements using its attributes

In general, the core attributes of HTML elements are the `class`, `id`, `style`, and `title` attribute. You can use these attributes in the following manner:

```
<div id="menu" class="shaded" style="…" title="Nice menu"> Your
content here </div>
```

Notice that all four attributes could be used at the same time. Also, the sequence of the attributes does not matter.

But we have not done any styling yet. The styling only takes place in the `style` attribute. To see an example, enter the following code between the `<body>` and `</body>` tag in the previous code.

```
<div style= "position:absolute; background-color:black;color:#ffffff;
 top:10px;right:10px;border:solid 3px yellow; height:200px;
width:200px;">Your content here
</div>
```

You should be able to see a 200px by 200px black box with yellow border in the upper-right corner of your browser window (as shown in the previous screenshot). Here's a screenshot that shows only the black box:
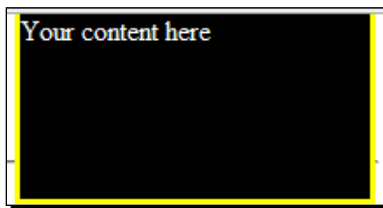
In general, the inline style that you have specified manipulates the stylistic properties of the `style` attribute, to make it look the way you want it to.

Only the `style` attribute allows you to style the HTML element. But this method is only used for specifying inline style for an element.

In case you are wondering what the `<title>` tag does, it is essentially an attribute that specifies extra information about an element. This is most often used within the `<head>` tag. If you open up any HTML document that contains a `<title>` tag, you will find the contents of this tag in the tab of your browser or title of your browser window.

What about `id` attribute and `class` attribute? We'll cover these briefly in the next section.

Specifying id and class name for an HTML element

In general, the `id` attribute and `class` attribute allows the HTML element to be styled by giving the CSS (Cascading Style Sheets, which we will be covering later in this chapter) a way to refer to these elements. You can think of the `id` attribute and `class` attribute as a 'name', or a way to identify the corresponding HTML element such that if this 'name' is referred by the CSS, the element will be styled according to the CSS defined for this particular element. Also, the `id` attribute and `class` attribute are often referred to by JavaScript in order to manipulate some of the DOM (Document Object Model) attributes, and so on.

There is one important idea that you must understand at this point of the chapter: the `id` attribute of each HTML element has to be unique within an HTML file, whereas the `class` attribute doesn't.

Cascading Style Sheets

CSS stands for Cascading Style Sheet. A CSS is used to control the layout, appearance, and formatting of the web page. CSS is a way for you to specify the stylistic appearance of the HTML elements. Via CSS, you can define the fonts, colors, size, and even layout of the HTML elements.

If you noticed, we have not added any form of CSS styles to our HTML document yet; in the previous screenshots, what you see is the default CSS of our browser (apart from the black box on the upper-right), and most browsers have the same default CSS if no specific CSS is defined.

CSS can be internal or external; an internal CSS is embedded in a HTML document using the `<style>` tag, whereas an external CSS is linked to by using the `<link>` tag, for example:

```
<link rel="stylesheet" type="text/css" href="style.css">.
```

In general, using internal CSS is considered to be a bad practice and should be avoided. External CSS is widely favored over internal CSS because it allows us to save more time and effort as we can change the design of the website by just making changes to a `.css` file instead of making individual changes to each HTML document. It also helps in improving performance, as the browser will only need to download one CSS and cache it in memory.

The most important point for this section is the use of CSS selectors and the syntax of the CSS.

The CSS selectors work as follows: for selecting IDs, the name of the ID is preceded by a hash character. For a class selector, it is preceded by a dot. In the code that you will be seeing later, you will see that both ID and class selectors are used (they are also commented in the source code). Here's a quick preview of the selectors:

```
/* this is a id selector */
#nameOfID {
    /* properties here*/
}

/* this is a class selector */
.nameOfClass {
        /* properties here*/
}
```

The syntax of the CSS is as follows: selector { declaration } . The declaration consists of a semicolon-separated list of name or value attribute pairs, in which colons separate the name from the value.

Remember that we've mentioned the `id` attribute and `class` attribute in the preceding section? Now you will see how `id` attributes and `class` attribute are being used by CSS.

Time for action – styling your HTML document using CSS

Now we are going to style the HTML document that we created in the preceding section, by using CSS. **For simplicity, we'll use an internal CSS. What will happen in this section is that you will see the CSS syntax in action, and how it styles each HTML element by making use** of the `id` attribute and `class` attribute of the respective HTML element. Note that both `id` and `class` selectors are used in this example.

> The completed version of this example can be found in the source code folder of `Chapter 1`, with the file name: `chapter1-css-appearance.html`

1. Continuing from the previous example, open up your text editor and insert the following code after the `</title>` tag:

```
<style type="text/css">
body{
    background-color:#cccccc;
}
/* Here we create a CSS selector for IDs by a name preceded by a
hash character */
#container{
    width:750px; /* this makes the width of the div element with
the id 'container' to have a width of 750px */
    height:430px;
    border:1px solid black;solid 1px black;

}
/* #[nameOfElement] */
#boxed1{
    background-color:#ff6600;
    border:2px solid black;
    height:360px;
    width:300px;
    padding:20px;
    float:left;
    margin:10px;
}

#boxed2{
    background-color:#ff6600;
    border:2px solid black;
    height:360px;
    width:300px;
```

```
        padding:20px;
        float:left;
        margin:10px;
    }
    #ordered1{
        font-size:20px;
        color:#ce0000;
        text-align:center;
    }
    #unordered1{
        font-size:12px;
        color:#000f00;
    }
    #ordered2{
        font-size:20px;
        color:#ce0000;
        text-align:center;
    }
    #unordered2{
        font-size:12px;
        color:#000f00;
    }
    #unordered2.nice{
        font-size:16px;
    }
    .intro{
        color:black;
        font-weight:bold;
    }
    a:link {color:#FF0000;}      /* unvisited link */
    a:visited {color:#00FF00;}  /* visited link */
    a:hover {color:#FF00FF;}  /* mouse over link */
    a:active {color:#0000FF;}  /* selected link */
    </style>
```

2. After adding the CSS code above, you will need to add `class` and `id` attributes to your HTML elements. Here's the stuff you'll need to add:

```
<!—- Some code omitted above -- >
<body>

    <!—- Some code omitted -- >
<p class="intro">This is a paragraph. I am styled by a class
called "intro"</p>
<hr>
```

```
<div id="container">
    <div id="boxed1">I am enclosed within a <i>div</i> tag. And I
can  be styled on a document level.
        <ol id="ordered1">
            <li>This is an ordered list and it is centered</li>
            <li>apple</li>
            <li>orange</li>
            <li>banana</li>
        </ol>

        <ul id="unordered1">
            <li>This is an unordered list.</li>
            <li>apple</li>
            <li>orange</li>
            <li>banana</li>
        </ul>

        <a class="link" href="#">I am a link that is styled by a
class</a>
    </div>

    <div id="boxed2">I am enclosed within a <i>div</i> tag. And I
am styled on a local level.
        <ol id="ordered2">
            <li>This is an ordered list and it is centered</li>
            <li>apple</li>
            <li>orange</li>
            <li>banana</li>
        </ol>

        <ul class="nice" id="unordered2">
            <li>This is an unordered list  and I have a class
defined</li>
            <li>apple</li>
            <li>orange</li>
            <li>banana</li>
        </ul>

        <a class="link" href="#">I am a link that is styled by a
class</a>
    </div>
</div>
</body>
</html>
```

The `class` and `id` attributes that need to be added are highlighted in the code snippet above. If you are not sure if you have done it correctly, open up `chapter1-css-appearance.html` and have a look.

3. Now save the file and open it in your browser. You should see that your HTML document now looks different to how it was before it was styled by CSS. Your output should be similar to the example shown in following screenshot:
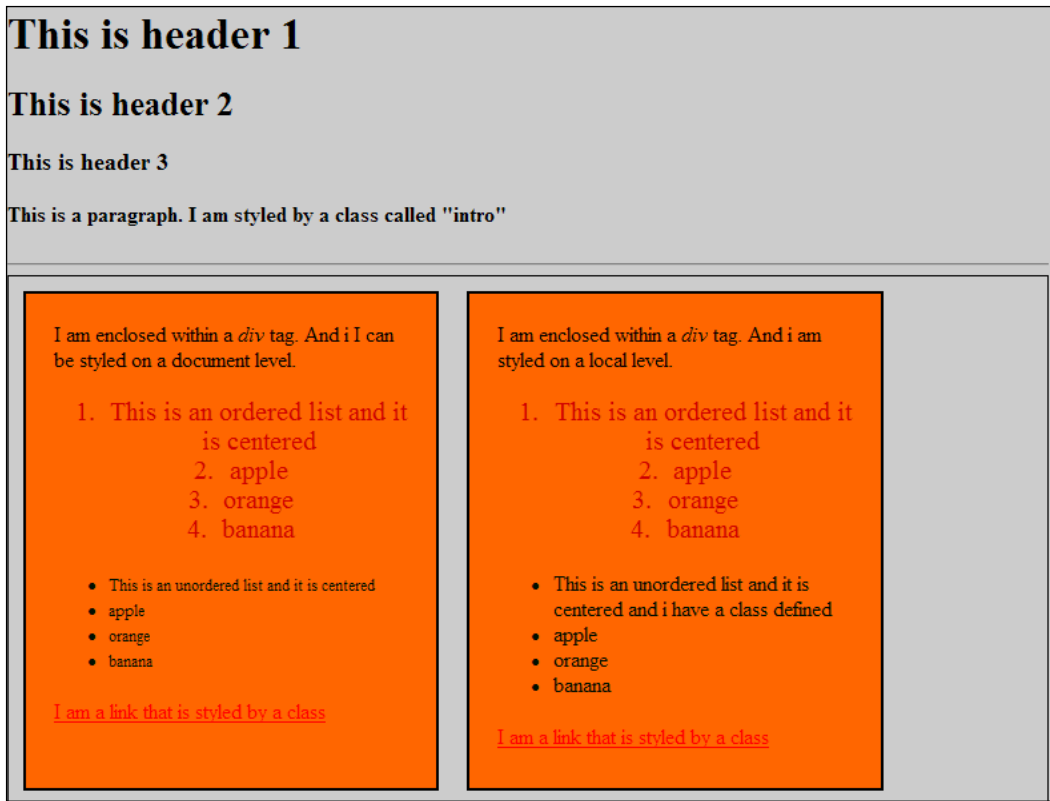
What just happened?

You have just applied CSS to the HTML document that you created in the previous section. Notice that you have used both the `id` selector and `class` selector syntax. Within each selector, you should also see some stylistic attributes.

The HTML elements in this example are similar to the previous example, except that the HTML elements now have `id` and `class` names.

In the following sub-sections, I'll continue to explain the techniques used for referring to the various HTML elements, and how we styled the elements by using their stylistic attributes.

Referring to an HTML element by its id or class name and styling it

We referenced various HTML elements by its `id` or `class` name. Consider the following code snippet in the above example:

```
<!—some code omitted above-->


<p class="intro">This is a paragraph. I am styled by a class called
"intro"</p>
<!—some code omitted -->
<div id="boxed">This is enclosed within a <i>div</i> tag. And it is
styled on a local level.
        <ol id="ordered1">
            <li>This is an ordered list and it is centered</li>
            <li>apple</li>
            <li>orange</li>
            <li>banana</li>
        </ol>

        <ul class="nice" id="unordered1">
            <li>This is an unordered list  and has a class defined</
li>
            <li>apple</li>
            <li>orange</li>
            <li>banana</li>
        </ul>
        <a class="link" href="#">This is a link that is styled by a
class</a>
    </div>
```

The highlighted code refers to the HTML elements where `ids` and `class` name attributes are being used. Notice that some of the HTML elements have both `ids` and `class` name attributes while some do not.

Now consider the CSS snippet which is found in the example:

```
#boxed1{
    background-color:#ff6600;
    border:2px solid black;
    height:360px;
    width:300px;
    padding:20px;
    float:left;
    margin:10px;
}
```

The `#boxed1` selector refers to the `<div>` with the id `#boxed1` in the HTML document. Notice that the `<div>` with the id `#boxed1` is styled according to the name and value attribute pairs within the declaration. If you make some changes to the value attribute and refresh your browser, you will notice changes to the `#boxed1` element as well.

Now, consider the following CSS snippets:

```
.intro{
    color:black;
    font-weight:bold;
}
```

And:

```
a:link {color:#FF0000;}     /* unvisited link */
a:visited {color:#00FF00;}  /* visited link */
a:hover {color:#FF00FF;}  /* mouse over link */
a:active {color:#0000FF;}  /* selected link */
```

The previous two code snippets are what we call `class` selectors, which have a slightly different syntax than the `id` selectors. For instance the `.intro` class selector selects the `<p>` with class name `"intro"` while the `a:link`, `a:visited`, `a:hover`, and `a:active` selectors refer to the four states of an anchor pseudo class.

Until now, we have covered how CSS selectors work to select HTML elements in an HTML document. But we have not covered the situation where an HTML element has both `id` and `class` attributes; we'll explain it now.

Differences between a class selector and an id selector

Although `id` selectors and `class` selectors appear to be the same, there are subtle differences. For instance, the `id` selector is used to specify a single HTML element, whereas the `class` selector is used to specify several HTML elements.

For example, you may try changing the anchor element `<a class="link" href="#">` to `<a class="intro" href="#">` and you would notice that the link is now bold.

> If an HTML element has a style attribute that is controlled by both the stylistic attributes of an `id` and `class` selector, then the style attributes in the `class` selector will take precedence over those in the `id` selector.

Other uses for class selectors and id selectors

In the following section, you will learn that the `id` and `class` name of an HTML element play an important role in providing interactivity on a web page. This is done by using JavaScript, where JavaScript makes a reference to an HTML element either by its `id` or `class` name, after which various actions such as DOM manipulation are performed on the HTML element that is referenced.

Complete list of CSS attributes

The examples given here are not complete. For a complete reference to CSS, you may visit `http://www.w3schools.com/css/css_reference.asp`.

JavaScript providing behavior to a web page

In this section we'll cover some of the key aspects of JavaScript. In general, if HTML provides content for an HTML document and CSS styles the HTML document, then JavaScript breathes life into an HTML document by providing behavior to the webpage.

The behavior can include changing the background colour of an HTML document dynamically, or changing the font size of the text, and so on. JavaScript can even be used to create effects such as animating slideshows, and fade-in and fade-out effects.

In general, the behaviors are event-based, and are achieved by manipulating the DOM in real-time (at least from the users' point of view).

In case you are fairly new to JavaScript, JavaScript is an interpreted programing language with object-oriented capabilities. It is loosely-typed, which means that you do not need to define a data type when declaring variables or functions.

In my opinion, the best way to understand the language features of JavaScript is through an example. Now, it's time for action.

Time for action – giving behavior to your HTML document

We are going to apply JavaScript to an HTML document (styled with CSS). In general, the HTML elements and CSS are not changing as compared to the previous example, except that you will see HTML buttons added to the HTML document.

The JavaScript applied to the HTML document in this example is known as inline JavaScript because it exists within the HTML document.

What we are trying to accomplish here is to show you the language features such as how to declare variables, functions, manipulating DOM of the HTML elements, and various methods of referencing HTML elements by their `id` or `class`. You will also learn about some of the commonly-used built-in methods of arrays, and elements that are referenced, and how to use them to make your tasks easier.

This example is nothing fancy, but you will learn some of the most important and commonly-used techniques for referencing HTML elements and then manipulating the DOM.

> (The completed code for this example can be found in the source code folder, `Chapter 1`, with the file name of: `chapter1-javascript-behavior.html`):

1. Continuing on from the previous example, enter the following JavaScript code after the `</style>` tag:

```
<script type="text/javascript">
function changeProperties(d){
    var e = document.getElementById(d);
    e.style.position = "absolute";
    e.style.fontFamily = "sans-serif";
    e.style.backgroundColor = "#000000";
    e.style.border = "solid 2px black";
    e.style.left = "200px";
    e.style.color = "#ffffff";
}
function arrangeList(f) {
    // This is the element whose children we are going to sort
    if (typeof f == "string"){ // check to see if the element is
"string"
        f = document.getElementById(f);
    }
    // Transfer the element (but not text node) children of e to
a real array
    var listElements = [];
    for(var x = f.firstChild; x != null; x = x.nextSibling)
        if (x.nodeType == 1){
            listElements.push(x);
        }
    listElements.sort(function(n, m) { // .sort is a built in
method of arrays
                var s = n.firstChild.data;
                var t = m.firstChild.data;
                if (s < t){
```

```
                        return -1;
                }
                else if (s > t){
                        return 1;
                }
                else{
                        return 0;
                }
        });
        for(var i = 0; i < listElements.length; i++){
            f.appendChild(listElements[i]);
        }
}
function insertContent(a){
    var elementToBeInserted = document.getElementById(a);
    elementToBeInserted.innerHTML = "<h1>This is a dynamic
content</h1><br><p>great to be here</p>";
}
function changeElementUsingName(a){
    var n = document.getElementsByName(a);
    for(var i = 0; i< n.length; i++){
    n[i].setAttribute("style","color:#ffffff");
    }
}
function hideElement(a){
    var header = document.getElementById(a);
    header.style.visibility = "hidden";
}
function hideElementUsingTagName(a){
    var n = document.getElementsByTagName(a);
    for(var i = 0; i< n.length; i++){
    n[i].setAttribute("style","visibility:hidden");
    }
}
</script>
```

Now save your document and load it in your browser, and you will see an example similar to the one shown in the next screenshot:

This is header 1

This is header 2

This is header 3

This is a paragraph. I am styled by a class called "intro"

I am enclosed within a *div* tag. And i can be styled on a document level.

1. this is an ordered list and it is centered
2. apple
3. orange
4. banana

- this is an unordered list and it is centered
- apple
- orange
- banana

I am a link that is styled by a class

I am enclosed within a *div* tag.

1. this is an ordered list and it is centered
2. apple
3. orange
4. banana

- this is an unordered list and it is centered and i have a class defined
- apple
- orange
- banana

I am a link that is styled by a class

| change properties | Create dynamic content | Rearrange list | Rearrange unordered list | hide header 1 |
| Change hyperlink colors | Hide header 2 (using tag name) | | | |

What just happened?

You have just created an HTML document styled with CSS, and applied JavaScript to it. There are generally no changes to the HTML elements and CSS as compared to the previous example, but you will see the `<button>` elements.

Now you can see the power of JavaScript in action by clicking on the HTML buttons. You should see that if you click on the **change properties** button, you will see the HTML box on the right shifts to the left by 200pixels, and its background change color. You can also click on other buttons to test their effect on the HTML document.

What happens when you click on each HTML button is that you are invoking a JavaScript function that manipulates the relevant HTML element in the document, via the DOM. You should see effects like hiding content, creating dynamic content, rearranging the list of items, and so on.

In the following sections, I'll first start by briefly introducing the JavaScript syntax, followed by attaching events to HTML elements, and finally using JavaScript's built-in methods to find HTML elements and manipulating them.

JavaScript Syntax

We'll start with learning the basic syntax of JavaScript. Consider the opening `<script>` tag:

```
<script type="text/javascript">
// code omitted
</script>
```

What the above `<script>` tag does is identify where JavaScript starts and ends. Within the `type` attribute, we write `text/javascript` to denote that this is a JavaScript code.

Now, let us consider the following code snippet:

```
function arrangeList(f) {
    if (typeof f == "string"){ // check to see if the element is
"string"
        f = document.getElementById(f);
    }
    var listElements = [];//declaring a variable
    for(var x = f.firstChild; x != null; x = x.nextSibling)
        if (x.nodeType == 1){
            listElements.push(x);
        }
    listElements.sort(function(n, m) { // .sort is a built in method
of arrays
                var s = n.firstChild.data;
                var t = m.firstChild.data;
                if (s < t){
                    return -1;
                }

                else if (s > t){
                    return 1;
                }
                else{
                    return 0;
                }
    });
```

```
        for(var i = 0; i < listElements.length; i++){
            f.appendChild(listElements[i]);
        }
    }
```

The above code snippet shows the function called `arrangeList`. We define a function by using the reserved keyword `function`, followed by the name of the function. Parameters are passed into the function within the `( )` and in this code snippet, `f` is the parameter passed into the function. The function starts with a {and ends with a}.

In short, the function syntax can be defined as follows:

```
function functionname(parameter1, parameter2, … parameterX){
    Body of the function
}
```

The second highlighted line shows decision making in JavaScript through the use of the `if` statement. The syntax is similar to the C programing `if` statement. The syntax of JavaScript's `if` statement is as follows:

```
if (condition){
    code to be executed if condition is true.
}
```

A variation of the if statement is the **if-else**

```
if (condition){
    code to be executed if condition is true.
}
else{
    code to be executed if condition is not true.
}
```

We use the keyword `var` followed by a variable name. In the above example, `var listElements = [];` means that a variable `listElements` is defined, and it is given the value of an empty list denoted by `[]`. In general, variables can be assigned arbitrary values since JavaScript is loosely-typed.

Continuing from above, you should see the `for` loop in action. Its syntax is also similar to the C language's `for` loop.

If you are new to JavaScript, you may be confused by `document.getElementById()` and statements like `listElements.push(x)`. What happens in these two lines is that we are using some of the built-in methods of JavaScript to reference the HTML element with the corresponding IDs. For now, `document.getElementById()` will be more important to you; this will be covered in the section where you learn how to find elements in your HTML document.

JavaScript events

Let's start off by looking at the following code snippet that is found in your JavaScript:

```
<button onclick="changeProperties('boxed1')">change properties</
button>
<button onclick="insertContent('empty')">Create dynamic content</
button>
<button onclick="arrangeList('ordered1')">Rearrange list</button>
<button onclick="arrangeList('unordered1')">Rearrange unordered list</
button>
<button onclick="hideElement('header1')">hide header 1</button>
<button onclick="changeElementUsingName('lost')">Change
hyperlink colors</button>
<button onclick="hideElementUsingTagName('h2')">Hide header 2 (using
tag name)
</button>
```

The above code snippets show HTML buttons with an event attached to them via `onclick`. When the button is clicked, the corresponding JavaScript function is invoked.

For example, `<button onclick="changeProperties('boxed1')">ch ange properties</button>` means that when this button is clicked, the `changeProperties()` function is invoked with the parameter `boxed1`, which happens to be a `div` element with the ID `boxed1`.

Finding elements in a document

Remember that we've seen a few built-in methods of JavaScript. JavaScript can be used to find elements in an HTML document by using some of JavaScript's built-in methods or properties. After finding the HTML element, you can manipulate its properties. JavaScript features three properties of the `Document` object (which is the root of every DOM tree) that allows you to find the HTML elements that you need. The techniques mentioned here form the backbone of JavaScript testing. Understanding this section is vital to understanding the rest of the book. So make sure that you understand this section of the chapter.

Firstly, the `document.getElementById()`. This property allows you to select an HTML element with a specific ID. `document.getElementById()` returns only a single element because the value of every `id` attribute is (supposed to be) unique. Here's a code snippet from the example:

```
function changeProperties(d){
    var e = document.getElementById(d);
    e.style.position = "absolute";
    e.style.fontFamily = "sans-serif";
    e.style.backgroundColor = "#000000";
    e.style.border = "2px solid black";
    e.style.left = "200px";
    e.style.color = "#ffffff";
}
```

Consider the highlighted line in the above code snippet, `var e = document.getElementById(d)`. What happens here is that the HTML element 'd', which happens to be a parameter of the function `changeProperties()`, is being referred. If you look at the source code for this example, you will see an HTML button with the following: `<button onclick="changeProperties('boxed1')">change properties</button>`. Notice that `'boxed1'` is being referenced, and this means that the parameter 'f' takes the value of the HTML element id of `'boxed1'`. Therefore, `var e = document.getElementById(d)` means that the HTML `div` with the ID of `'boxed1'` is being assigned to variable `e` via the `document.getElementById()` method.

Secondly, note the `document.getElementsByName()` statement. This is similar to `document.getElementById()`, but it looks at the `name` attribute instead of the `id` attribute. It returns an array of elements rather than a single element. Consider the following code snippet:

```
function changeElementUsingName(a){
    var n = document.getElementsByName(a);
    for(var i = 0; i< n.length; i++){
    n[i].setAttribute("style","color:#ffffff");
    }
}
```

What happens here is that the HTML element with the name 'a' (which happens to be a parameter of the function) is referenced, and because it returns an array of elements, we use a `for` loop to loop through the elements, and use the method `.setAttribute` to change the color of the text to white. The `name` attribute applies to `<form>` and `<a>` tags only.

Finally, look at `document.getElementsByTagName()`. This method looks for HTML elements by the HTML tag name. For instance, the following code:

```
function hideElementUsingTagName(a){
    var n = document.getElementsByTagName(a);
    for(var i = 0; i< n.length; i++){
    n[i].setAttribute("style","visibility:hidden");
    }
}
```

finds the HTML element by the tag name, and makes it hidden. In our example, a `h2` is used as a parameter and hence when you click on the relevant button, all text that is enclosed within the `<h2>` tags will disappear.

Now, if you change the parameter to `div`, then you will notice that all of the boxes will disappear.

Putting it all together

Now I'll briefly describe how JavaScript works to interact with HTML elements. Here's what you will learn in this subsection: after an HTML button is clicked (an event), it invokes a JavaScript function. Then, the JavaScript function receives a parameter and executes the function. Consider the following code snippets.

The following code is for an HTML button with an event attached to it:

```
<button onclick="insertContent('empty')">Create dynamic content</
button>code
```

Next, the following code is for an HTML `div` element:

```
<div id="empty"></div>
```

Lastly, the following is code which shows the JavaScript function that is to be invoked:

```
function insertContent(a){
    var elementToBeInserted = document.getElementById(a);
    elementToBeInserted.innerHTML = "<h1>This is a dynamic content</
h1><br><p>great to be here</p>";
}
```

Now, let me explain what we are trying to do here; after clicking the HTML button, the JavaScript function `insertContent()` is invoked. The parameter 'empty' is passed into `insertContent()`. 'empty' refers to the `div` element with ID 'empty'.

After `insertContent()` is invoked, the parameter `'empty'` is passed to a variable `var elementToBeInserted`, by using `document.getElementById()`. Then, using the built-in method `innerHTML()` for HTML element nodes (because an HTML element node is passed to the `elementToBeInserted` variable), we dynamically insert the text "`<h1>This is a dynamic content</h1><br><p>great to be here</p>`".

Go ahead and open the file in your web browser, and click on the HTML button. You will notice a new piece of text being inserted into the HTML document, dynamically.

> The built-in method `innerHTML()` for HTML element nodes allows us to manipulate (or in this case, dynamically insert) HTML contents into) the HTML node that is using the `innerHTML()` method. For example, in our example, we will insert "`<h1>This is a dynamic content</h1><br><p>great to be here</p>`" into `<div id="empty"></div>`. Technically speaking, after the insertion, the end result will be:
> : `<div id="empty"><h1>This is a dynamic content</h1><br><p>great to be here</p></div>`.

The difference between JavaScript and server-side languages

Generally speaking, the main difference between JavaScript and server-side languages lies in their usage and where they are executed. In modern usage, JavaScript runs on the client side (the users' web browser), and server-side languages runs on servers, and is therefore often used to read, create, delete, and update databases such as MySQL.

This means that the JavaScript is processed on the web-browser, whereas server-side languages are executed on web servers.

Server-side languages include ASP.NET, PHP, Python, Perl, and so on.

In the context of modern web development techniques, you have probably heard of Web 2.0 applications. An important technique is that JavaScript is often used extensively to provide interactivity and to perform asynchronous data retrieval (and in some cases manipulation), which is also known as AJAX (which is a short-hand for Asynchronous JavaScript and XML).

JavaScript cannot be used to interact with databases, whereas server-side languages such as PHP, Python, and JSP can.

JavaScript is also known as front-end, whereas server-side is back-end technology.

> JavaScript can be used on the server side as well, although it is most frequently associated with client-side technologies. Although JavaScript is typically not associated with interacting with databases, this might change in the future. Consider new browsers such as Google Chrome, which provides a database API for JavaScript to interact with built-in databases in the browser itself.

Why pages need to work without JavaScript

Although there are many arguments as to whether we should make web pages work with or without JavaScript, I personally believe that it depends on how the website or application is used. But anyway, I'll start off with some of the common reasons for why pages need to work without JavaScript.

Firstly, not all users have JavaScript enabled in web browsers. This means that users whose JavaScript is not enabled will not be able to use your application (or features) if it requires JavaScript.

Secondly, if you intend to support your user on their mobile device, then you need to make sure that your website or application works without JavaScript. The main reason is because support for JavaScript on mobile devices is often less than satisfactory; if you use JavaScript, your website or application may not work as well as expected (or worse, fail to work altogether).

Another way to look at this is based on your understanding of your user base. For instance, probably the only time when you can afford to ignore users who have JavaScript disabled is when you can guarantee or know before-hand that your user base has JavaScript enabled. Such situations can occur when you are developing an application for internal use, and you know before-hand that all of your users have JavaScript enabled.

In case you are wondering what you can do to create pages that work without JavaScript, you can check out the idea of graceful degradation. Imagine that you have an application and the core features of this application are AJAX-based. This means that in order to use your application, your user will need to have JavaScript enabled. In this case, you would most probably have to consider making your pages to work without JavaScript in order to ensure that all of your users can use your application.

What is testing?

Generally speaking, programmers write a program with a few objectives in mind. Besides creating a program to solve a certain problem or to fulfil a certain demand, other common objectives would include ensuring that the program is at least correct, efficient, and can be easily extended.

Of the above-mentioned objectives, correctness is the most important objective—at least in this book. By correct, we mean that for any given input, we need to make sure that the input is what we want or need, and that the corresponding output is correct. The implicit meaning of this is that the program logic is correct: it works the way we intended it to work, there are no syntax errors, and the variables, objects, and parameters referenced are correct and what we need.

Take, for instance, a retirement plan calculator written in JavaScript. We could expect the user to enter values such as their current age, retirement age, and savings per month. Imagine if a user were to enter incorrect data, such as a string or character. The JavaScript retirement plan calculator would not work, because the input data is incorrect. Or worse, if the user entered the correct data and our algorithm for calculating the amount of money to be set aside for retirement is incorrect, this results in the output being incorrect.

The above errors could be avoided by testing, which is the main topic of this book. In the remaining portions of this chapter, we will talk about some of the types of errors that you may face as a JavaScript programmer. But before we move into that, I'll briefly discuss why we need to test.

Why do you need to test?

First and the foremost, human beings are prone to mistakes. As a programmer, you have probably made coding mistakes during your programing career. Even the best programmers on Earth have made mistakes. What makes it worse is that we may not have realized the mistake until we tested the program.

Secondly, and perhaps more importantly, JavaScript generally fails silently; there are no error messages to tell you what errors have occurred or where that error has occurred, assuming you are not using any testing unit or tools to test your JavaScript. Therefore, there is little or no way to know what has happened to your JavaScript program, if there is an error.

In Microsoft's Internet Explorer, you can actually see if you have any JavaScript errors. You will need to turn on **Script Debugging** which is found in **Tools | Internet Options | Advanced| Script Debugging**. With **Script Debugging** turned on, you will see a yellow 'yield' icon on the bottom left hand corner for IE7 or IE8 if you have any JavaScript errors. Clicking on that icon will give you a window where you can click on **Show Details** to get more information about the error.

Thirdly, even if there are ways to inform you of JavaScript errors, such as enabling **Script Debugging**, as mentioned above, there are certain errors that cannot be detected by such means. For instance, your program syntax may be 100 percent correct, but your algorithm or program logic might be incorrect. This means that even if your JavaScript can be executed, your output could be incorrect.

Lastly, testing JavaScript will help you to identify cross-browser compatibility issues. Because there are approximately five major types of browsers (not accounting for different versions) to support—namely Microsoft's Internet Explorer, Mozilla's Firefox, Google's Chrome, Apple's Safari and the Opera Web Browser—you will certainly need to test to ensure that your website or application works across all browsers, because different browsers have different DOM compatibilities.

Ensuring that the program is correct means confirming and checking that the input is correct, and then that the output is what we intended it to be.

Types of errors

Before I start introducing the types of JavaScript errors, we need to understand the inner workings of JavaScript and the web browser. In general, a user requests a web document from the server, and this document is loaded into the user's web browser. Assuming that the web document has JavaScript embedded (either via an external JavaScript file or via inline JavaScript), the JavaScript will be loaded together with the web document (from top to bottom). As the web document is loaded by the web browser, the JavaScript engine of the web browser will begin to interpret the JavaScript embedded in the web document. This process will continue until the JavaScript (and the web document) is completely loaded into the user's web browser, ready for interaction. Then the user may start to interact with the web document by clicking on links or buttons that may have JavaScript events attached to them.

Now, with the above process in mind, we'll start introducing the different types of JavaScript errors, by using simple examples.

Loading errors

The first types of error that we'll discuss are loading errors. Loading errors are errors that are caught by the JavaScript engine of the web browser as the document is loading.

In other words, loading errors occur before the JavaScript has the opportunity to function. These errors are typically spotted by JavaScript engines before the code has the chance to execute.

With the previously-mentioned things in mind, let us now experience how such loading errors occur.

Time for action – loading errors in action

Now we'll see loading errors in action. We do not actually see it, but you will learn about some of the most common causes for loading errors.
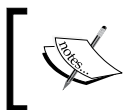
> The complete code for this example can be found in the source code folder
> `Chapter 1`, with a file name of `chapter1-loading-errors.html`

1. Open up your text editor and create a new document.

2. Enter the following code into your document:

```
<html>
<head><title>JavaScript Errors - Loading Errors</title></head>
<body>
<script type="text/javascript">/*
1. Loading Errors
*/


/*
// Example 1 - syntax errors
var tests = "This is a test";    // note two s
document.write(test);    // note one s
*/


/*
// Example 2 - syntax errors as the keyword "var" is not used
Var Messsage = "This is a test";    // note three s's
document.write(Message);    // note two s's
*/


/*
```

```
// Example 3 - error caused by using a key word
var for = "this is a test";
document.write(in);
*/
</script>
</body>
</html>
```

3. Now, uncomment the `/*` and `*/` wrapped around example 1, save the document and load it into your browser. You should see a blank page on your web browser.

4. Repeat the above step for example 2 and example 3. You should see a blank page for both examples 2 and 3.

What just happened?

You have just created an HTML document with erroneous JavaScript code. From the comments in the code, you should realize that the errors are caused largely due to syntax errors. And when such errors occur, there is simply no response from the JavaScript in the web browser.

Some examples of common syntax errors would include missing brackets, missing semi-colons, and incorrect variable names.

In general, as long as your code is correct in terms of syntax, then you should be able to avoid loading errors.

Now, you might ask, what happens if only certain parts of the JavaScript code are incorrect? This would depend on where the error has occurred.

Partially correct JavaScript

In general-JavaScript is executed or loaded from top to bottom. This means that the first line of code is loaded first, followed by the next, and so on until finally the last line of the code is loaded. This has important implications for partially-correct JavaScript.

Time for action – loading errors in action

Now we'll see partially-correct JavaScript code in action and its implications.

> The completed source code for this example can be found in the source code folder, with the file name `Chapter1-loading-errors-modified.html`.

1. Open your text editor, create a new document, and enter the following code into your document:

```
<html>
<head><title>JavaScript Errors - Loading Errors</title></head>
<body>
<script type="text/javascript">/*
1. Loading Errors - modified
*/

// this is correct code
var tests = "This is a CORRECT test";
document.write(tests);

// this is incorrect code. The variable name referred is incorrect
var Messsage = "This is a FIRSTtest";
document.write(Message);

// this is correct code
var testing = "this is a SECOND test";
document.write(testing);

</script>
</body>
</html>
```

2. Now save your document and load your document in your web browser. You should see the text **This is a test in** your browser.

What just happened?

If you trace the code, you should see that the JavaScript executes from top to bottom. It stops executing when it encounters an error where an incorrect variable name is referenced by `document.write()`. Because it stops executing when it encounters an error, the remaining JavaScript code will not be executed.

Things are slightly different if your JavaScript code is organized in terms of functions. In this situation, functions that have incorrect syntax will fail to execute, whereas syntactically-correct functions will continue to work, regardless of its order in the code.

By now, you should have a brief understanding of loading errors and how to prevent them by making sure that your code is syntactically correct.

Now let us move on to the next form of error—runtime errors.

Runtime errors

Do you remember how JavaScript is loaded together with the web document into the browser? After the web document is loaded completely into the web browser, it is ready for various events, which leads to execution of JavaScript code.

Runtime errors occur during execution; for instance, consider an HTML button that has a JavaScript event attached to it. Assuming that a JavaScript function is assigned to an event, then if the JavaScript function has an error, that function will not be executed when the user clicks on the HTML button.

Other forms of runtime error occur when you misapply an object, variable, or method, or when you reference objects or variables that do not exist yet.

Time for action – runtime errors in action

Now we shall see all three common causes of runtime errors in action.

> The code sample is saved in the source code folder of `Chapter 1`, entitled: `chapter1-runtime-errors.html`.

1. Open up your text editor, enter the following code into a new document:

```
<html>
<head><title>JavaScript Errors</title></head>
<script type="text/javascript">/*

2. Runtime Errors

*/

alert (window.innerHTML);

var Test = "a variable that is defined";
alert(Test); // if variables is wrongly typed, than nothing wil
happen
```

```
        // nothing happens when the user clicks on the HTML button, which
        invokes the following function
        function incorrectFunction(){
                alert(noSuchVariable);
        }
        </script>
        <body>
        <input type="button" value="click me" onclick="incorrectFunction()
        " />

        </body>
        </html>
```

2. Save the document and load it into your web browser.

3. After loading the document into your browser, you will see two alert boxes: the first box says **undefined** and the second alert box says **a variable that is defined**. Then you will see an HTML button that says **click me**.

4. Click on the button, and you will see that nothing happens.

What just happened?

The first alert that you have seen shows you an error that is caused by misapplying a method. `window.innerHTML` does not exist, as `.innerHTML` is applied to HTML elements and not to `window`. The second alert window says that a variable that is defined as the variable is defined before the `alert()` references it. Lastly, nothing happens when you click on the HTML button because the function that is to be invoked has the error of referencing to a variable that is not defined. Hence it is not executed during the event `onclick()`.

In this example, you should realize that the logic of your code is of great importance—you will need to define your variables or objects before using them in your code. Also, make sure that the method or properties applied are correct. Otherwise, you will end up with a runtime error.

Now, we'll move on to the last form of JavaScript error—logic errors.

Logic errors

Logic errors are difficult to explain. But in general, you can see logic errors as errors that occur when the code does not work the way that you intend it to. It is much easier to understand what logic errors are by experiencing them. So, let us take some action.

Time for action – logic errors in action

In this final example, you will see logic errors.

1. Open your text editor, enter the following code into a new document:

```html
<html>
<head><title>JavaScript Errors</title>
<script type="text/javascript">
/* Logic Errors */

//saving some input in wrong variables
function amountOfMoneySaved(amount){
    var amountSpent, amountSaved;
    amountSpent = amount; // where you really meant amountSaved
    var currentAmount = 100;
    var totalAmountSaved = currentAmount - amountSpent;
    alert("The total amount of money you have now is " +
            totalAmountSaved );
}

function checkInput(amount){

    if(amount>0 && amount<99)
        alert("is number");
    else
        alert("NOT number");
}

</script>
</head>

<body>
<!-- this shows an infinite loop, an obvious logic error-->
<script>
// an infinite loop
for(var i = 0; i<10; i--){
    document.write(i + "<br>");
}
</script>

<form id="moneyForm">
  You currently have 100 dollars.
  The amount of money you have saved is: <input type="text"
id="money" name="money" /><br />
```

```
    <input type="submit" value="Submit"
        onclick="amountOfMoneySaved(moneyForm.money.value)" />
    </form>
    </body>
    </html>
```

2. Now, save the code and open the document in your browser.

3. You will see two simple forms. The first form which has the text: **You currently have 100 dollars**. **The amount of money you have saved is**" " followed by an input box. And the second form contains the text: **Checking if you have entered a digit** followed by an input box.

4. Now try to enter a number that is larger than 99 (say, 999).

You may have noticed that after entering your input, the total amount of money appears to have decreased. This is an example of a logic error, where you are supposed to add the input, but instead the function subtracts the input. Why did the program not work the way it was intended to?

What just happened?

You have just witnessed a simple example of logic error in action. Logic errors can take many forms. You may have noticed a code snippet in the above example that is commented out.

```
<script type="text/javascript">// example 1: infinite loop
for(var i = 0; i<10; i--){
    document.write(i + "<br>");
}
</script>
```

This is an example of an infinite `for` loop. In this loop, you may have noticed that the statement `document.write(i+<br>");` should be executed 10 times (from `var i = 0` to when `i = 9`). However, the third expression in the initializer within the `for` statement is decreasing (`i--`).

As a result, the variable `i` will never be able to reach the condition where `i>10`. If you uncomment the code, you will notice that the statement `document.write(i"<br>");` will continue to execute until the web browser hangs; if you are using Firefox on a Windows machine, the web browser will hang and you will have to quit the browser by using the Task Manager.

Some advice for writing error-free JavaScript

By now, you should have a brief understanding of the types of JavaScript errors. While we typically cannot avoid errors, we should try to minimize errors as we write code. In this section, I'll briefly discuss some of the strategies that you can take, as a beginner JavaScript programmer, to minimize the amount of errors that can occur.

Always check for proper names of objects, variables, and functions

As seen in the above forms of errors, you should always make sure that you are using the correct names for your objects, variables, and functions. Because such errors will not be shown in your web browser, as you write your **code, it is always a good idea to check for** the correct use of names.

This also includes using unique names for different variables, objects, and functions. Remember that JavaScript is case-sensitive; therefore do remember to check that you are using the correct case for your variables, objects, and functions as well.

Check for proper syntax

Because you are using JavaScript, at least for this book you should check that you are using the correct syntax before you run your program. Previously, we went through some of the key features of the language syntax, for instance, ending each statement with a semi-colon, using proper and matching brackets, using correct or unique function names, and so on.

Plan before you code

Planning before the actual coding process helps to reduce the possibility of logic errors. This helps you to think through your program and spot obvious logic errors in your code. Planning can also help you to check for blind spots, such as missing features or functions.

Check for correctness as you code

As you write your program, it is always a good idea to check for errors as you complete certain portions of the code. For example, if your program consists of six functions, it is always wise (and less error prone) to check the correctness of each function. Making sure that each function that you have written is correct before moving to the next function is a good practice, and can save you a lot of trouble as you write large programs.

Preventing errors by choosing a suitable text editor

I personally believe that a suitable text editor (or IDE) is a crucial step in minimizing coding errors. Notice that I did not say that you need a "good" text editor, but rather a "suitable" text editor. This is because different programing languages have different features and different capabilities.

For instance, if you have programmed in Python, you will notice that you do not need to have the ability to check for matching brackets, because Python is based on code blocks (tabbing or spacing to denote blocks of code). However, in the case of JavaScript, you would certainly need your text editor to help you check for matching (or missing) brackets. Some code editors that can accomplish the above includes Dreamweaver (commercial) and Eclipse (free).

In addition to matching brackets checking, here are some other features that will be useful for you when you are coding in JavaScript:

1. Auto-tabbing or spacing after keywords or matching brackets: This will help you in visually inspecting the code structure, and will minimize code errors.

2. Auto-complete or auto-suggest feature: This means that as you type your code, the editor is smart enough to suggest to some of the words (or code) that you have used in your program so that you can quickly refer to them as you code. This is especially useful for checking user-defined variables, objects, and functions.

3. Syntax coloring: This will help you identify if you are misusing any keywords. Remember runtime errors? Runtime errors can be caused by the misuse of keywords. If you are using any of the keywords for user-defined variables, objects, or functions, syntax coloring will help you to identify this.

Summary

Whew, we've covered a lot in this chapter. The bulk of the content covered in this chapter forms the building blocks of what we need to use in the later chapters. Specifically, we covered the following topics:

◆ We learnt about HTML, CSS, and JavaScript in web pages. In general, HTML provides the content, CSS styles the web document, and JavaScript provides the behaviour and interactivity for the webpage.

◆ We've also learnt about the syntax of HTML, CSS, and JavaScript.

◆ We've also learnt about the key techniques of using ID and Class selectors in order for CSS to refer to various HTML elements and perform stylistic operations on the referenced HTML element.

- ◆ For JavaScript, we learnt about three important techniques for JavaScript to reference to HTML elements. These three techniques (or rather built-in methods) are: `document.getElementById()`, `document.getElementsByName()`, and `document.ElementsByTagName()`.

- ◆ Next we learnt about testing and why we need to test. In general, testing is to ensure that the program works correctly—that is, for the given input, we have the correct output. Also, testing helps to uncover syntax errors and confirm that the program works in the way that we intend it to work.

- ◆ We covered the types of JavaScript errors—namely loading errors, runtime errors, and logic errors. We've also covered some simple examples of each type of errors and the common causes of them.

- ◆ We covered some important tips and advice on how to write error-free code.

Now that we have covered the basic building blocks of JavaScript testing, you will see how we can make use of them to perform ad hoc testing, which we will cover in the next chapter. You will notice some of the functions and built-in methods used in this chapter will be used in the next chapter.

2

Ad Hoc Testing and Debugging in JavaScript

In this chapter, we'll formally move into testing the JavaScript programs that we actually create. But before I start, I'd like to brief you on what you can expect in this chapter. In this chapter, you will learn about two major ideas—the first idea being how different browsers can affect JavaScript testing, and the second major idea being how you can test your JavaScript program by using the alert(). You will also learn how to access the values on a form, manipulate the values and finally output the values in a meaningful manner.

You will also see many of the techniques introduced in the previous chapter being used extensively.

To be more specific, we shall learn about the following topics:

- ◆ The purpose of ad hoc testing
- ◆ What happens when your browser encounters an error in JavaScript
- ◆ Browser differences and the need to test in multiple browsers
- ◆ Common browser messages and what they mean
- ◆ How to find out if you are getting the right output and putting the right values in the correct places in your code
- ◆ How to access values on a form and how to access other parts of the web page
- ◆ Tips on what to do when your JavaScript program does not give you the expected result

- ◆ What to do if the script does not run

- ◆ How to perform a visual inspection

- ◆ How to use the `alert()` to test your JavaScript program

- ◆ Commenting out parts of your code in order to simplify testing

- ◆ Why ad hoc testing isn't always enough

So before I move on to the main topics of this chapter, I'll briefly mention the two basic ideas that you should understand before moving on to the rest of the chapter.

The purpose of ad hoc testing–getting the script to run

The first basic idea concerns the purpose of ad hoc testing. The main purpose of ad hoc testing is to quickly get your code up and running and then see if there are any errors with your code. As mentioned previously, the three different types of JavaScript errors entail loading, runtime, and logic errors.

The main advantage of ad hoc testing is that it allows you to test your JavaScript program without bogging you down. It is meant for those who want to save time, especially when testing small pieces of code.

What happens when the browser encounters an error in JavaScript

Now it's time for the second basic idea. In the previous chapter, I have briefly described how a web page is loaded in to the browser and then rendered in the web browser, waiting for interaction with the user. I have also mentioned that, in general, JavaScript fails silently; it does not explicitly tell or show you what errors (if any) have occurred. This happens when your browser does not have any form of debugging turned on.

However, modern web browsers feature built-in ways for the browser to tell the user that some form of errors have occurred on the web page. This happens when you explicitly turn on or install the debugging tools for the web browser. For some of the browsers, you will also need to explicitly turn on the error console in order to find out what error has occurred.

In case you are wondering what you need to do in order to make use of these built-in features, here are some simple instructions to help you to get started:

1. For Firefox—turn on your web browser and go to **Tools**. Click on **Error Console**.

2. For Internet Explorer—**you need to go to Tools | Internet Options | Advanced**. Scroll down to **Browsing** and check **Display a notification about every script error**.

You now understand the basic ideas about why we perform ad hoc testing. We will now move on to a more complex topic—how browser differences can affect your JavaScript program.

Browser differences and the need to test in multiple browsers

In general, browsers have different features. The one difference that matters the most to us, at least in this book, is the JavaScript engine used by different browsers. Different JavaScript engines process JavaScript in different manners. This has important implications for us. Certain JavaScript functions or methods that are supported by one web browser may not be supported by another.

The main essence of JavaScript is that it provides behavior to the web page through DOM manipulation; different browsers have different levels of support for DOM.

We will not attempt to go into a deep discussion about what is supported and what is not by various browsers. Instead, we'll point you to this website: `http://www.quirksmode.org/compatibility.html`.

This link provides a summary of the various incompatibilities of various web browsers according to different selectors. For our purpose here we should be more focused on the DOM selectors since we are concerned about JavaScript. Feel free to browse through the website for the details. But for now, the main idea that you need to understand is that browser differences result in incompatibilities and hence we need to test for browser compatibility.

Most beginner JavaScript programers would often want to know how they can find out the browser that their visitors are using. After all, if you can find out what browser your visitors are using, you'll be able to create compatible JavaScript code. That's true to a large extent; so now we'll start by learning how we can check the visitor's browser.

Time for action – checking for features and sniffing browsers

In this section, we would like to introduce you to the **navigator** object. The navigator object is a built-in object that provides you with information regarding the visitor's browser. What we are trying to do here is to show you how the navigator object works, and how you can make programing decisions based on the browser information.

> The source code for this example can be found in the source code folder, `Chapter 2`, with a file name of `browser-testing-sample-1.html` and `browser-testing-sample-2.html`.

1. Start your text editor if you have not already done so, and then enter the following code in your text editor:

```
<html>
<head><title>Testing for Browser - Example 1</title></head>
<body>
<script type="text/javascript">// Sample 1
var browserType ="Your Browser Information Is As Follows:\n";
for( var propertyName in navigator){
    browserType += propertyName + ": " + navigator[propertyName] +
"\n";
}
alert(browserType);
</script>
</body>
</html>
```

Here's what's happening in the previous code: we defined a variable `browserType`. After which we used a `for` loop and defined another variable, `propertyName`.

2. The line that says:`for( var propertyName in navigator )` means that we are trying to get all of the properties in the `navigator` object.

3. After doing this, we append the `propertyName` and the information into the `browserType` variable. And finally, we output the information in an alert box.

4. Now, load the file in to your web browser and you should see a pop-up window containing information about your web browser.

Notice that the alert box contains various types of information about your web browser. You can also access specific property of the browser for your own use. This is what we are going to do next.

Now that you have learned how to use the navigator object, it's time to see how we can make use of this information in order to perform programing decisions:

5. Create another new document, and enter the following code into it:

```html
<html>
<head><title>Testing for Browser - Example 2</title></head>
<body>
<script type="text/javascript">// Sample 2
var typeOfBrowser = navigator.appName;
document.write(typeOfBrowser);
if(typeOfBrowser == "Netscape"){
    alert("do code for Netscape browsers");
}
else{
    alert("do something else");
}
</script>
</body>
</html>
```

In the previous sample code, we have defined the variable `typeOfBrowser`, which is used to decide which to execute. An easy way would be to use the `if else` statement to choose the of code to execute, based on the browser name.

What just happened?

In the preceding examples, you have seen how to use the navigator object to perform "browser sniffing", and based on the given information, perform appropriate actions.

Apart from using the navigator object, you can also test browser differences based on the browser's capabilities. This means that you can test whether the user's browser has a certain feature or not. This technique is also known as feature testing. Now, we'll briefly see how you can perform capability testing.

Testing browser differences via capability testing

Capability testing is an important and powerful way to cope with browser incompatibilities. For instance, you might want to use a certain function that might not be supported on different browsers. You can include a test to see if this function is supported or not. Then, based on this information, you can execute the appropriate code for your visitor.

Time for action – capability testing for different browsers

In this section, we'll briefly introduce a simple-to-use method that can help you to quickly test for a certain feature. The method that we are going to use is the .hasFeature() method. Now, we'll dive right in and see it in action.

> The source code for this example can be found in the source code folder, Chapter 2, with a file name of browser-testing-by-feature-2.html and browser-testing-by-feature.html.

1. Start your text editor and then enter the following code in your text editor:

```
<html>
<head><title>Testing browser capabilities using .hasFeature()</title></head>
<body>
<script type="javascript/text">
var hasCore = document.implementation.hasFeature("Core","2.0");
document.write("Availability of Core is "+ hasCore + "<br>");

var hasHTML = document.implementation.hasFeature("HTML","2.0");
document.write("Availability of HTML is "+ hasHTML + "<br>");

var hasXML = document.implementation.hasFeature("XML","2.0");
document.write("Availability of XML is "+ hasXML  + "<br>");

var hasStyleSheets = document.implementation.hasFeature("StyleSheets","2.0");
document.write("Availability of StyleSheets is "+ hasStyleSheets + "<br>" );

var hasCSS = document.implementation.hasFeature("CSS","2.0");
document.write("Availability of CSS is "+ hasCSS  + "<br>" );

var hasCSS2 = document.implementation.hasFeature("CSS2","2.0");
document.write("Availability of CSS2 is "+ hasCSS2  + "<br>");

</script>
</body>
</html>
```

To make things clearer, I've defined variables for each of the features and the version number. In general, the usage of `hasFeature` is as follows:

```
.hasFeature(feature, version);
// feature refers to the name of the feature to test in string
// version refers to the DOM version to test
```

2. Now load the file in to your web browser and you should see various types of text being created dynamically on the screen.

Similarly, you can use the information that you have derived from the user's browser to perform various decisions in a similar manner as to what you have seen in the previous example.

So, for simplicity and explanation sake, here's how you can perform programing decisions using the `.hasFeature()`.

3. Create another new document, and enter the following code into it:

```html
<html>
<head><title>Testing browser capabilities using .hasFeature() -
Example 2</title></head>
<body>
<script type="text/javascript">
var hasCore = document.implementation.hasFeature("Core","2.0");
if(hasCore){
    document.write("Core is supported, perform code based on the
feature<br>");
}
else{
    document.write("Feature is not supported, do alternative code
to enable your program<br>");
}
</script>
</body>
</html>
```

The sample code above is self-explanatory as it is similar to the example seen in `browser-testing-sample-2.html`.

What just happened?

The previous example is a simple extension of what you can do to test for browser differences. It is similar to the first example, which "sniffs" for the browser information explicitly, while the method using `.hasFeature()` is based on capabilities.

There is no right or wrong way to test for browser differences. However, a general practice is to use `.hasFeature()` to test for program functionality. That is to say that we often use `.hasFeature()` in order to ensure that our JavaScript functionality will be available in different browsers.

The previous example shows some of the features that you can test for by using `.hasFeature()`. Following is a list of the remaining features that you can test for by using `.hasFeature()`:

- Events
- UI Events
- Mouse Events
- HTML Events
- Mutation Events
- Range
- Traversal
- Views

Now that you have some understanding of how you can test for browser differences, it is time for the next topic—getting the output and putting values in the right places.

Are you getting the correct output and putting values in the correct places?

In this section, we'll learn how to make sure that we are getting the output and putting the correct values in the correct places. This means that we need to understand how to use JavaScript with a HTML form.

Accessing the values on a form

In general, "getting" values would generally mean that a user would input some values into a form (in a HTML document of course), and then our program "gets" the input from the web form. Also, these values may or may not be manipulated by other functions; the initial user input may be passed as arguments to other functions and then manipulated.

This can be achieved by using JavaScript's built-in utilities; JavaScript provides a few ways for you to access the form values so that you can use these values later on. In general, JavaScript will **"get" the value from a form** `onsubmit` event.

Time for action – accessing values from a form

In the following example, we'll start off with a simple HTML form. You will learn about various techniques for accessing different form elements. What happens here is that you'll see how we first submit a form by using the `onsubmit` event. The `onsubmit` event allows us to send the form to a JavaScript function, which then helps us to extract the values from various form element types. So for this example, I need you to relax and understand the techniques mentioned earlier.

> The source code for this example is found in `Chapter 2` of the `source code` folder, with a name of `accessing-values-from-form.html`.

1. Once again, enter the following code into your newly-created document in your favorite editor:

```html
<html>
<head><title>Getting Values from a HTML form</title>
<script type="text/javascript">/*
In this example, we'll access form values using
the following syntax:

document.NameOfForm.NameOfElement

where:
NameOfForm is the name of corresponding form
NameOfElement is the name of the element ( within the
corresponding form)
*/
function checkValues(){
    var userInput = document.testingForm.enterText.value;
    alert(userInput);
    var userInputTextArea = document.testingForm.enterTextArea.
value;
    alert(userInputTextArea);
    var userCheckBox = document.testingForm.clickCheckBox.value;
   // this is for checkbox
    if(document.testingForm.clickCheckBox.checked){

    userCheckBox = true;
    }
    else{
```

```
        userCheckBox = false;


    }
    alert(userCheckBox);

    var userSelectBox = document.testingForm.userSelectBox.value;
    alert(userSelectBox);
    // here's another way you can "loop" through your form
elements
    alert(document.testingForm.radioType.length);
    for(var counter = 0; counter<document.testingForm.radioType.
length;counter++){
        if(document.testingForm.radioType[counter].checked){
            var userRadioButton = document.testingForm.
radioType[counter].value;
            alert(userRadioButton);
        }
    }
}
</script>
</head>
<body>
<h1>A simple form showing how values are accessed by JavaScript</
h1>
<form name="testingForm" onsubmit="return checkValues()">
<p>Enter something in text field:<input type="text"
name="enterText"  /></p>
<p>Enter something in textarea:<textarea rows="2" cols="20"
name="enterTextArea"></textarea></p>
<p>Check on the checkbox:<input type="checkbox"
name="clickCheckBox"  /></p>
<p>Select an option:
<select name="userSelectBox">
  <option value="EMPTY">--NIL--</option>
  <option value="option1">option1</option>
  <option value="option2">option2</option>
  <option value="option3">option3</option>
  <option value="option4">option4</option>
</select>
</p>
<p>Select a radio buttons:<br />
  <input type="radio" name="radioType" value="python" /> Python
  <br />
  <input type="radio" name="radioType" value="javascript" />
JavaScript
```

```
        <br />
        <input type="radio" name="radioType" value="java" /> Java
        <br />
        <input type="radio" name="radioType" value="php" /> PHP
        <br />
        <input type="radio" name="radioType" value="actionscript" />
    ActionScript 3.0
    </p>
    <input type="submit" value="Submit form" />
    </form>
    </body>
    </html>
```

You should notice that there are various input types, such as `text`, `textarea`, `checkbox`, `select`, and `radio`.

2. Save the form and then load it in to your web browser. You should see a simple HTML form on your screen.

3. Go on and enter values for the fields, and then click on **Submit form**. You should see a series of alert windows, which repeat the values that you have entered.

What just happened?

In the simple form example described earlier, you submitted a form via a JavaScript event `onsubmit`. The `onsubmit` event calls a JavaScript function `checkValues()` which then helps us to access the values from different form elements.

In general, the syntax for accessing form elements is as follows:

document.formName.elementName.**value**

where `formName` is the name of the form, and `elementName` refers to the name of the element.

As in the previous example, the form name is `testingForm`, as can be seen in `<form name="testingForm" onsubmit="return checkValues()">`, and the input text element has the name `enterText`, as can be seen in `<input type="text" name="enterText" />`. Therefore, based on this code snippet, we'll access the form values by doing the following:

`document.testingForm.enterText.value`

We can then assign this to a variable that can be saved for later use, as shown in the code example.

The previous example should be simple to grasp. But in this short example, I've also introduced a few more useful methods. Consider the following code snippet which is found in the example:

```
for(var counter = 0; counter<document.testingForm.radioType.
length;counter++){
        if(document.testingForm.radioType[counter].checked){
            var userRadioButton = document.testingForm.
radioType[counter].value;
            alert(userRadioButton);
        }
}
```

Notice that in the highlighted line I've made use of the `length` property; `document.testingForm.radioType.length` means that I am calculating how many elements by the name of `radioType` do I have in the form named `testingForm`. This property returns an integer that can then be used in loops such as the `for` loop, as seen in the previous code snippet. You can then loop through form elements and check for their values by using the method mentioned earlier.

Another important technique that you can use can be found in the following code snippet:

```
if(document.testingForm.clickCheckBox.checked){
    userCheckBox = true;
}
```

What happens in the highlighted line is that `document.testingForm.clickCheckBox.checked` returns a `true` or `false`. You can use this technique to check if the form element you are referring to has input or not. You can than make use of this information to perform decisions.

Another technique for accessing form values

As you may have noticed, we are accessing the form elements by making use of the `name` attribute. We would most probably (and most likely) make use of the `name` attribute to access the form elements, as it is easier to refer to those elements. But nonetheless, here's an alternate method that you can quickly look though:

Instead of writing

```
document.formName.elementName.value
```

You can write this:

```
document.forms[integer].elementName.value
```

where you are making use of the `forms` object, and `elementName` refers to the name of the input.

An example for the **preceding code sample would be:**

```
document.forms[0].enterText.value
```

Notice that the `forms` object is appended with `[0]`. **This means that the** `forms` object are treated similarly to an array; `forms[0]` refers to the first form in the web page, and so on.

Now that you have understood the basics of accessing the values for a form, you will learn how to make sure that you are getting the correct values in the correct places in the next section.

Accessing other parts of the web page

In this section, you will learn how to access other parts of the web page. In general, you have already learned the building block for accessing different parts of the webpage by making use of `getElementById`, `getElementsByTag`, and `getElementsByTagName`. Now you will make further use of these, along with the newly-learned techniques of accessing values from a form.

Time for action – getting the correct values in the correct places

In this example, you will see a general integration of the techniques that you have learned so far. You will learn how to access form values, manipulate them, perform operations on them, and finally, put the new output on other parts of the webpage. To help you better visualize what I am about to describe, following is a screenshot of the completed example:

Enter your information here   **Response**

Enter your name

Enter your place of birth

Enter your age

Enter your spending per month

Enter your salary per month

Enter your age you wish to retire at

Enter the amount of money you wish to have for retirement

Final response:

The example that you are about to use is a simple JavaScript program that checks to see if you can retire at the age that you want to. It will request some basic information from you. Based on the information provided, it will determine if you can retire at that time, based on the amount of money you would want to have at the time of retirement.

You will be building a form (2 forms in fact, loosely speaking), where the user will be required to enter basic information into the first form (on the left), and after entering the required information in each field, there will be another input field appearing dynamically on the right of the field (in the middle of the web page), if the input is correct.

As you enter the information, a JavaScript event will fire off a JavaScript function that checks for the correctness of the input. If it is correct, there will be a new field created on the right-hand side of the field that has just accepted the input, and the field on the left will be disabled.

After the fields on the left are filled correctly, you will notice a complete form is being filled out in the middle of the page. After you click on **Submit**, the code will perform the calculations and determine whether you can retire at the age you have specified, based on the amount of money that you require.

The basic requirements for this example are as follows:

- Correct values must be entered. For instance, if the field requires you to enter your age, the field must only accept integers and no characters should be allowed.

- If the fields require a text input, such as your name, no integers will be allowed.

> The completed source code for this example can be found in the source code folder for `Chapter 2`, with a file name of `getting-values-in-right-places.html`.

So now, let us get started with this example:

1. Let us start by building the basic interface for this example. So, enter the following code (the HTML and style) in to your text editor.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>Getting the right values</title>
<style>
input{
    padding:5px;
    margin:5px;
    font-size:10px;
}
```

```
.shown{
    display:none;
}
.response{
    padding:5px;
    margin:5px;
    width:inherit;
    color:red;
    font-size:16px;
    float:left;
}
#container{
    position:absolute;
    width:800px;
    padding:5px;
    border: 2px solid black;
    height:430px;
}
#left{
    height:inherit;
    width:370px;
    border-right:2px solid black;
    float:left;
    padding:5px;
}

#right{
    height:inherit;
    width:300px;
    float:left;
    padding:5px;
}
#bottom{
    float:left;
    bottom:5px;
    padding:5px;
}

#finalResponse{
    float:left;
    width:780px;
    height:250px;
    border:3px solid blue;
    padding:5px;
```

```
    }

    /* this is for debugging messages */
    #debugging{
        float:left;
        margin-left:820px;
        height:95%;
        width:350px;
        border:solid 3px red;
        padding:5px;
        color:red;
        font-size:10px;
    }
    </style>
    <script type="javascript/text">
    // some Javascript stuff in here
    var globalCounter = 0;
    </script>
    <body>

    <div id="container">
        <div id="left">
        <h3>Enter your information here</h3>
            <form name="testForm" >
                <input type="text" name="enterText" id="nameOfPerson"
    onblur="submitValues(this)" size="50"  value="Enter your name"/
    ><br>
                <input type="text" name="enterText" id="birth" onblur=
    "submitValues(this)" size="50"  value="Enter your place of birth"/
    ><br>
                <input type="text" name="enterNumber" id="age" onblu
    r="submitValues(this)" size="50" maxlength="2" value="Enter your
    age"/><br>
                <input type="text" name="enterNumber" id="spending"
    onblur="submitValues(this)" size="50" value="Enter your spending
    per month"/><br>
                <input type="text" name="enterNumber" id="salary" on
    blur="submitValues(this)" size="50" value="Enter your salary per
    month"/><br>
                <input type="text" name="enterNumber" id="retire" onbl
    ur="submitValues(this)" size="50" maxlength="3" value="Enter your
    age you wish to retire at" /><br>

                <input type="text" name="enterNumber"
    id="retirementMoney" onblur="submitValues(this)" size="50"
```

```
    value="Enter the amount of money you wish to have for retirement"/
    ><br>

            </form>
        </div>
        <div id="right">
        <h3>Response</h3>
            <form name="testFormResponse" id="formSubmit" onsubmit="ch
    eckForm(this);return false">

            </form>
        </div>
        <div id="finalResponse"><h3>Final response: </h3></div>

    </div>
    </body>
    </html>
```

2. You might want to save this file and load it in your browser to see if you are getting the same output as the previous screenshot that you have seen.

Notice that in the HTML form above, there is JavaScript event `onblur`. `onblur` is a JavaScript event that occurs whenever an element loses focus. So you should see that all input elements have an `onblur`, which fires off the `submitValues()` function.

You should also see that there is a `this` as an argument for `submitValues().this` is one of the most powerful JavaScript keywords, and refers to the corresponding element it is being referred to. An example would be `<input type="text" name="enterText" id="nameOfPerson" onblur="submitValues(this)" size="50"  value="Enter your name"/>`. In this code snippet, `submitValues(this)` will submit the HTML form element object by the name of `enterText`.

Now, it's time for the JavaScript programing. What happened, as explained previously, is that on the JavaScript event `onblur`, it will submit the HTML form element object to the function `submitValues()`. So, we'll start with this function first.

3. Now, enter the following code between the `<script type="javascript/text">` tags:

```
function submitValues(elementObj){
    // using regular expressions here to check for digits
    var digits = /^\d+$/.test(elementObj.value);

    // using regular expressions
    // here to check for characters which
    // includes spaces as well
    var letters =  /^[a-zA-Z\s]*$/.test(elementObj.value);

    // check to see if the input is empty
    if(elementObj.value==""){
        alert("input is empty");
        return false;
    }
    // input is not relevant; we need a digit for input elements
with name "enterNumber"
    else if(elementObj.name == "enterNumber" && digits == false){
        alert("the input must be a digit!");
        return false;
    }
    // input is not relevant; we need a digit for input elements
with name "enterNumber"
    else if(elementObj.name == "enterText" && letters == false){
        alert("the input must be characters only!");
        return false;
    }
    // theinput seems to have no problem, so we'll process the
input
    else{
        elementObj.disabled = true;
        addResponseElement(elementObj.value,elementObj.id);
        return true;

    }

}
```

I've commented on what the code is doing, but I'll focus on some of the techniques used in the previous function.

What we are trying to do here is to check the correctness of the input. For this example, we only accept either pure numbers or pure characters (including spaces). This is what the following code snippet is doing:

```
var digits = /^\d+$/.test(elementObj.value);
var characters =  /^[a-zA-Z\s]*$/.test(elementObj.value);
```

Here we are making use of regular expressions to check for the correctness of the input. `/^\d+$/` and `/^[a-zA-Z\s]*$/` are regular expressions, where both are appended with the `test` method. The `test` method tests for the value of the HTML form object's value. For instance, `var digits = /^\d+$/.test(elementObj.value)` will return `true` if the value is indeed digits, and `false` if it is not. Similarly, `var characters = /^[a-zA-Z\s]*$/.test(elementObj.value)` will return `true` if it is characters (which includes spaces) and `false` if it is otherwise.

In case you wish to learn more about using regular expressions, you can refer to `http://www.w3schools.com/jsref/jsref_obj_regexp.asp` and see how it works.

The previous information will be used during the decision-making process in `if-else` statements. The `if-else` statements check for the name of the HTML object; `enterNumber` expects an integer input. If it is not `enterNumber,` it is expecting a character input.

You should notice that if there are no problems with the input, we will disable the input element and pass the `value` and `id` of the HTML form object to a function `addResponseElement()`, after which we will `return true`, which signifies the successful execution of the code and the submission of the form values.

So now, we'll move on to the `addResponseElement()` function:

4. Continuing with the **current document, append the following code below** `submitValues()` **function:**

```
function addResponseElement(messageValue, idName){
    globalCounter++;
    var totalInputElements = document.testForm.length;
    var container = document.getElementById('formSubmit');
    container.innerHTML += "<input type=\"text\" value=\""
+messageValue+ "\"name=\""+idName+"\" /><br>";
    if(globalCounter == totalInputElements){
        container.innerHTML += "<input type=\"submit\" value=\
"Submit\" />";
    }
}
```

What `addResponseElement()` does is that it attempts to dynamically add the input element on the form to the right of original input form. Here, you should find `var container = document.getElementById('formSu bmit')` familiar. It looks for an HTML element with ID of formSubmit. After this, we will append HTML into this form, through the `innerHTML` method. `container.innerHTML += "<input type=\"text\" value=\""` `+messageValue+ "\"name=\""+idName+"\" /><br>";` attempts to append the input that is wrapped between the outermost inverted commas into `<form>` tags.

You should also notice `var totalInputElements = document.testForm.length;`. What this line of code does is determine the total number of input elements that `testForm` has, by using the `length` property. We are making use of this information to determine if we are on the last input field of the form, so that we can append a Submit button on the other form.

Next, we will create the function, which is called after the second form, which has a name of `testFormResponse`, is submitted.

5. Continuing with the **current document, append the following code below** `addResponseElement()` **function:**

```
function checkForm(formObj){
    var totalInputElements = document.testFormResponse.length;
    var nameOfPerson = document.testFormResponse.nameOfPerson.
value;
    var birth = document.testFormResponse.birth.value;
    var age = document.testFormResponse.age.value;
    var spending = document.testFormResponse.spending.value;
    var salary = document.testFormResponse.salary.value;
    var retire = document.testFormResponse.retire.value;
    var retirementMoney = document.testFormResponse.
retirementMoney.value;
    var confirmedSavingsByRetirement;
    var ageDifference = retire - age; // how much more time can
the user have to prepare for retirement
    var salaryPerYear = salary * 12; // salary per year
    var spendingPerYear = spending * 12; // salary per year

    // income per year, can be negative
    // if negative means cannot retire
    // need to either increase spending
    // or decrease spending
    var incomeDifference = salaryPerYear - spendingPerYear;
```

```
    if(incomeDifference <= 0){
        buildFinalResponse(nameOfPerson,-1,-1,-
1,incomeDifference);
        return true;
    }
    else{
        // income is positive, and there is chance of retirement
        confirmedSavingsByRetirement = incomeDifference *
ageDifference;
        if(confirmedSavingsByRetirement <= retirementMoney){
            var shortChange = retirementMoney -
confirmedSavingsByRetirement;
            var yearsNeeded = shortChange/12;

buildFinalResponse(nameOfPerson,false,yearsNeeded,retire,
shortChange);
            return true;
        }
        else{
            var excessMoney = confirmedSavingsByRetirement -
retirementMoney;
            buildFinalResponse(name,true,-1,retire,excessMoney);
            return true;
        }
    }
}
```

What happens in this function is pretty straightforward. The various form values are assigned to the various variables. Then we begin some simple calculations to see if the user will have enough money for retirement. You may refer to the comments in the function to understand the logic of the calculations.

In general, we'll call the function `buildFinalResponse()`, irrespective of whether the user can retire on time, and with the required amount of money. So here's the `buildFinalResponse()`.

Continuing with the current document, append the following code below `checkForm ()` function:

```
function buildFinalResponse(name,retiring,yearsNeeded,retire,
shortChange){
    var element = document.getElementById("finalResponse");
    if(retiring == false){
        element.innerHTML += "<p>Hi <b>" + name + "</b>,<p>";
        element.innerHTML += "<p>We've processed your information
and we have noticed a problem.<p>";
```

```
        element.innerHTML += "<p>Base on your current spending
habits, you will not be able to retire by <b>" + retire + " </b>
years old.</p>";
        element.innerHTML += "<p>You need to make another <b>" +
shortChange + "</b> dollars before you retire inorder to acheive
our goal</p>";
        element.innerHTML += "<p>You either have to increase your
income or decrease your spending.<p>";
    }
    /*
    else if(retiring == -1){
        element.innerHTML += "<p>Hi <b>" + name + "</b>,<p>";
        element.innerHTML += "<p>We've processed your information
and we have noticed HUGE problem.<p>";
        element.innerHTML += "<p>Base on your current spending
habits, you will not be able to retire by <b>" + retire + " </b>
years old.</p>";
        element.innerHTML += "<p>This is because you spend more
money than you make. You spend <b>" + shortChange + "</b> in
excess of what you make</p>";
        element.innerHTML += "<p>You either have to increase your
income or decrease your spending.<p>";
    }
    */
    else{
        // able to retire but....
        element.innerHTML += "<p>Hi <b>" + name + "</b>,<p>";
        element.innerHTML += "<p>We've processed your information
and are pleased to announce that you will be able to retire on
time.<p>";
        element.innerHTML += "<p>Base on your current spending
habits, you will be able to retire by <b>" + retire + "</b> years
old.</p>";
        element.innerHTML += "<p>Also, you'll have ' <b>" +
shortChange + "</b> amount of excess cash when you retire.</p>";
        element.innerHTML += "<p>Congrats!<p>";
    }
}
```

The function `buildFinalResponse()` is similar to the `addResponseElement()` function. It simply looks for the required HTML element, and appends the required HTML to the element.

Here, you can clearly see the JavaScript functions, methods, and techniques that you have learnt so far in this book.

Save the file. You can try playing with the example and see how it works for you.

What just happened?

In the previous example, you saw how to access the values of the form, perform operations on the input, and then place the output on various parts of the web page. You may have noticed that we made extensive use of `getElementById`. We have also made use of the `form` object and the `value` method in order to access the value of various elements in the form. Then, by making use of `getElementById`, we looked for the required HTML element and appended the output into the HTML element.

But, at this point of time, you may be wondering what you should do if you happen to make mistakes in the program. This is what we'll be focusing on in the next section.

Does the script give the expected result?

My opinion is that before we can begin any meaningful discussion, we must understand what is meant by "expected result".

"Expected result(s)" can have several meanings, at least for the purpose of this book. For instance, as mentioned in the previous chapter, the output should be correct for each input; as this refers to the eventual output. There is another output, which takes the form of "visual output". For instance, for every user interaction or event, our web applications would often provide a form of visual cue to allow the user to know that something is happening. In this case, our visual clues helping in the way that we intended would be deemed as an "expected result".

A simple tip, to check if the script gives you the expected results, is to use simple input and perform the calculations yourself. Make sure that your calculations are correct and test your program.

In the later part of this chapter, we'll discuss two relevant techniques in detail. But first, let us see what actions we can take if our script does not run.

What to do if the script doesn't run

If the script doesn't run, it is very likely that loading or runtime errors have occurred, depending on the way that your program is coded. For example, in the previous program that you have just created, you know that the program is not running if there is no response after you have entered the first input field and the focus is no longer on the first input field.

In this case, there are a few possibilities (all of which fall under the three basic forms of JavaScript errors as mentioned in the previous chapter). Firstly, there might be an error in the syntax of your input field for the JavaScript event, or, there could be a serious error in the function that is called by the JavaScript event. If not, it could be a logic error.

Whatever the errors may be, it is often difficult to guess what and where the errors are. Therefore, I'll introduce three important techniques for testing out your code, if your code does not run.

Visually inspecting the code

Visually inspecting the code means that you will be a human compiler, and visually check for errors in your code. My opinion is that there are certain pre-conditions and tips for visual inspection:

- There must be a good code block structure. This means that code should be properly spaced and indented for visual clarity. At one glance, you should be able to see which code is nested under which `if-else` statements, or which functions it belongs to.

- The code editor that you use makes a huge difference. A common error is the mismatching of brackets or inverted commas. Therefore, a code editor that allows for the highlighting of matching brackets will help you to spot such errors.

- Check for semicolons after each statement(s).

- Check to see if variables are initialized. If variables are used in later parts of the program but are not initialized, it will create serious errors.

The previous actions are some of the things I will do if my script doesn't run or if it doesn't run in the way that I intend it to. However, despite our best intentions, visual inspection of code can only be useful for small programs, such as programs that have less than 30 to 50 lines of code. If the programs get any larger, or if they contain various functions that are invoked during events, it might be better (and more efficient) to check our code by using the `alert` function.

Using alert() to see what code is running

The `alert` method can be used to check that what code is running is being used appropriately. We have not formally introduced the alert method yet. But just in case, you can use the alert function to create pop-up windows just about anywhere in a JavaScript program. The syntax is as follows:

```
alert(message)
```

where `message` can take almost any number of values (or variables if it has been defined or initialized). Due to this flexible nature of the `alert` method, it can also be used to show values, strings, and object types as well.

The issue in using `alert` stems from the location where the `alert` should be placed in the code. This will be demonstrated in the next hands-on example.

Using alert() to see what values are being used

As mentioned earlier, the `alert` method can be used to show almost any type of value. Therefore, a common usage would be to pass a variable into the `alert` method and see if the value is what we need or intended.

Similarly, we need to know where we should be applying the `alert` method to in order to ensure that our code inspection is correct.

At this point of time, an example would be the most appropriate way to see how we can make use of the `alert` method to inspect the code for errors. So, let us see how this works.

Time for action – using alert to inspect your code

This example is similar to what you have done in the previous example. In this example, you will be required to insert `alert` in the appropriate places in order to check which part of the code is running. In some cases, you will need to pass values to the `alert` method and see if the value is the one that you want.

To be honest, it would be tedious to tell you step-by-step where you should place the `alert` method, especially as the bulk of the code in this example is similar to the previous one. However, to make things easier for you to follow, we'll start immediately with the entire program, after which we'll explain to you the rationale behind the location of the `alert` methods and the values that are passed into the `alert` method.

> The source code of the following example can be found in `Chapter 2` of the source code folder, named `getting-values-in-right-places-using-alert.html`.

1. This example is similar to the previous one, except that the JavaScript has been changed slightly. Replace the JavaScript code from the previous example with the following code:

```
var globalCounter = 0;
function submitValues(elementObj){
    alert("submitValues");
    alert(elementObj.name);
    var totalInputElements = document.testForm.length;
    alert("total elements: " + totalInputElements);
    var digits = /^\d+$/.test(elementObj.value);
    var characters =  /^[a-zA-Z\s]*$/.test(elementObj.value);
```

```
        alert(characters);
        if(elementObj.value==""){
            alert("input is empty");
            return false;
        }
        else if(elementObj.name == "enterNumber" && digits == false){
            alert("the input must be a digit!");
            return false;
        }
        else if(elementObj.name == "enterText" && characters ==
false){
            alert("the input must be characters only!");
            return false;
        }
        else{
            alert("you've entered : " + elementObj.value);
            elementObj.disabled = true;
            alert(elementObj.value);
            addResponseElement(elementObj.value,elementObj.id);
            return true;


        }

    }

    function addResponseElement(messageValue, idName){
        alert("addResponseElement");
        globalCounter++;
        var totalInputElements = document.testForm.length;
        alert("totalInputElements");
        var container = document.getElementById('formSubmit');
        container.innerHTML += "<input type=\"text\" value=\""
+messageValue+ "\"name=\""+idName+"\" /><br>";
        if(globalCounter == totalInputElements){
            container.innerHTML += "<input type=\"submit\" value=\
"Submit\" />";
        }
    }

    function checkForm(formObj){
        alert("checkForm");

        var totalInputElements = document.testFormResponse.length;
        alert(totalInputElements);
```

```
    var nameOfPerson = document.testFormResponse.nameOfPerson.
value;
    alert(nameOfPerson);

    var birth = document.testFormResponse.birth.value;
    alert(birth);

    var age = document.testFormResponse.age.value;
    alert(age);

    var spending = document.testFormResponse.spending.value;
    alert(spending);

    var salary = document.testFormResponse.salary.value;
    alert(salary);

    var retire = document.testFormResponse.retire.value;
    alert(retire);

    var retirementMoney = document.testFormResponse.
retirementMoney.value;
    alert(retirementMoney);

    var confirmedSavingsByRetirement;

    var ageDifference = retire - age; // how much more time can
the user have to prepare for retirement
    alert(ageDifference);
    var salaryPerYear = salary * 12; // salary per year
    alert(salaryPerYear);
    var spendingPerYear = spending * 12; // salary per year
    alert(spendingPerYear);

    var incomeDifference = salaryPerYear - spendingPerYear;
    alert(incomeDifference);

    if(incomeDifference <= 0){
        buildFinalResponse(nameOfPerson,-1,-1,-
1,incomeDifference);
        return true;
    }
    else{
        confirmedSavingsByRetirement = incomeDifference *
ageDifference;
        if(confirmedSavingsByRetirement <= retirementMoney){
```

```
                var shortChange = retirementMoney -
    confirmedSavingsByRetirement;
                alert(shortChange);
                var yearsNeeded = shortChange/12;


    buildFinalResponse(nameOfPerson,false,yearsNeeded,retire,
    shortChange);
                return true;
            }
            else{
                var excessMoney = confirmedSavingsByRetirement -
    retirementMoney;
                alert(excessMoney);
                buildFinalResponse(name,true,-1,retire,excessMoney);
                return true;
            }
        }
    }


    function buildFinalResponse(name,retiring,yearsNeeded,retire,
    shortChange){
        alert("buildFinalResponse");

        var element = document.getElementById("finalResponse");
        if(retiring == false){
            alert("if retiring == false");
            element.innerHTML += "<p>Hi <b>" + name + "</b>,<p>";
            element.innerHTML += "<p>We've processed your information
    and we have noticed a problem.<p>";
            element.innerHTML += "<p>Base on your current spending
    habits, you will not be able to retire by <b>" + retire + " </b>
    years old.</p>";
            element.innerHTML += "<p>You need to make another <b>" +
    shortChange + "</b> dollars before you retire inorder to acheive
    our goal</p>";
            element.innerHTML += "<p>You either have to increase your
    income or decrease your spending.<p>";
        }
        else{
            // able to retire but....
            alert("retiring == true");

            element.innerHTML += "<p>Hi <b>" + name + "</b>,<p>";
```

```
        element.innerHTML += "<p>We've processed your information
    and are pleased to announce that you will be able to retire on
    time.<p>";
        element.innerHTML += "<p>Base on your current spending
    habits, you will be able to retire by <b>" + retire + "</b> years
    old.</p>";
        element.innerHTML += "<p>Also, you'll have <b>" +
    shortChange + "</b> amount of excess cash when you retire.</p>";
        element.innerHTML += "<p>Congrats!<p>";
    }
}
```

2. Save the document and load it in to your web browser. Play around with the example and see how the alert boxes notify you of which part of the code is being executed, and also the values being entered.

What just happened?

If you go through the previous example, you will notice that the `alert()` is most often placed at the beginning of functions, and when variables are being initialized. To check the functions, we often manually type in the name of the function and pass it as arguments to the `alert` method, to inform us of what is happening as we interact with the program. Similarly, we pass the variables that are defined (the values from the form elements) as arguments to the alert method to inform us of what values are being entered by the user.

Therefore, by using a single `alert()` method, we are able to find out what code is running and what values are being used. However, this method may be slightly too tedious or frustrating, because the alert boxes keep on popping up on your window. Here's a simple alternative for checking what code is running, and also to inspect the input elements.

A less obtrusive way to check what code is running and the values used

To test our code in a less obtrusive manner we would write a simple debugging function. This debugging function should print out the names of the functions, and some other variables. For simplicity's sake, we'll demonstrate a simple debugging function that prints the name of the function, and the HTML element being used. So, let us get started.

Time for action – unobtrusively checking what values are used

As mentioned above, we'll be demonstrating a very simple debugging function that helps you to identify which code is running and also which HTML element is in use. Here, you'll get a basic idea of some of the actions that you can perform in order to have a less obtrusive way of testing your code.

Again, this example is similar to the previous one, but there are some important elements that which we will be adding to the previous example. In essence, we will be adding a function, some HTML, and CSS to it.

However, you might find it tedious to refer back to the previous example and add the new elements to the previous example. Therefore, it is recommended that you stay with me on this example.
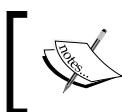
> Alternatively, you can view the source code in the source code folder, `Chapter 2`, with a file name of `getting-value-in-right-places-complete.html`.

So, without further ado, let us start right now:

1. Insert the following CSS code in between the `<style>` tags:

```
/* this is for debugging messages */
#debugging{
    float:left;
    margin-left:820px;
    height:95%;
    width:350px;
    border:solid 3px red;
    padding:5px;
    color:red;
    font-size:10px;
}
```

2. Now, for the HTML container which will contain the debugging messages, **enter the following code snippet before** `</body>` tag:

```
<div id="debugging"><h3>Debugging messages: </h3></div>
```

What happens here is that **the preceding HTML element will be used to** provide a visual separation **between the debugging messages and the** simple **application itself. Save the file now , load it to your web browser and** you **will see an example similar to the one shown in the next screenshot:**

Enter your information here	Response	**Debugging messages:**
Enter your name		
Enter your place of birth		
Enter your age		
Enter your spending per month		
Enter your salary per month		
Enter your age you wish to retire at		
Enter the amount of money you wish to have for		

Final response:

3. Next, you will need to append the following code to your JavaScript code:

```javascript
function debuggingMessages(functionName, objectCalled, message){
    var elementName;
    if(objectCalled.name){
        elementName = objectCalled.name;
    }
    else if(objectCalled.id){
        elementName = objectCalled.id;
    }
    else{
        elementName = message;
    }

    var element = document.getElementById("debugging");

    element.innerHTML += "Function name :" +functionName+
"<br>element :" +elementName+"<br>";
}
```

The previously-mentioned function is used to capture the name of the function used right now; this is equivalent to what code is in use right now, because our program is event driven and the functions are, in general, triggered by the user.

The three arguments are as follows:

- `functionName` refers to the functionName of the function used right now. In the next step, you shall see the method used to derive this value dynamically.

- `objectCalled` refers to the HTML object being used.

- `Message` refers to a string. This can be any message that you want; it is meant to provide some form of flexibility to the kind of debugging messages that you can write to the screen.

Also, we are making use of the `.innerHTML` method to append the messages into the HTML `div` element for the `id` "debugging".

4. Now finally, it's time to see how we can use this function. In general, we use the function as follows:

```
debuggingMessages("name of function", elementObj,"empty");
```

If you refer to the source code, you will see that the previously-mentioned function is being used sparingly in the program. Consider the following code snippet:

```
function submitValues(elementObj){
    //alert("submitValues");

    debuggingMessages("submitValues", elementObj,"empty");
    //alert(elementObj.name);
    var totalInputElements = document.testForm.length;
    //alert("total elements: " + totalInputElements);
```

In the previous case, the value of `"submitValues"` will be passed because `submitValues` is the name of the function. Notice that we also passed the function argument, `elementObj` into `debuggingMessages()` in order to notify us what is being used in the current function.

5. Finally, you might want to add the `debuggingMessages("name of function", elementObj,"empty")` to each function in your JavaScript program. If you are not sure where you should use this function, refer to the source code given.

If you are typing in the function yourself, then do take note that you might have to change the argument names in order to accommodate each of the functions. In general, `debuggingMessages()` can be used in place of the `alert()` method. So, if you are unsure of where you should use `debuggingMessages()`, you can use `debuggingMessages()` for every `alert()` used for inspecting the code in the previous example.

6. If you have executed the entire program, you will see something similar to the next screenshot:

What just happened?

You have just created a function that allows you to inspect your code in a less obtrusive manner, by making use of some built-in methods of JavaScript, which includes the .innerHTML method. What happens here is another example of how you can access values, manipulate them, and then output these values to the required HTML element, in order to make inspection less obtrusive.

If you look through the source code, you may have noticed that I used different messages during different situations; this will bring more flexibility to your debugging functions, if you use one.

Commenting out parts of the script to simplify testing

Commenting out parts of the script is another important and simple-to-use ad hoc technique for testing your JavaScript code. Essentially, you comment out the code that will not be used immediately.

Because we have not introduced how to do multiple line commenting, I'll take this chance to show you how to use it. The syntax is as follows:

```
/*
This is a multiple line comment
*/
```

Here's how commenting out parts of the script can be used to simplify testing: we would often comment out all other code that we would not use at first. For instance, the first function used in `getting-values-right-places-complete.html` is the `submitValues()` function.

We would make sure that the `submitValues()` function is correct before uncommenting the second function that is used, which is the `addResponseElement()` function.

The process goes on until all functions are uncommented, which means that the code is correct.

With all of these points in mind, we'll now move on to a simple workout based on the previous example.

Time for action – simplifying the checking process

In this example, there will not be any source code for you to copy. Instead, you can use the previous example found in `getting-values-right-places-complete.html` and try out the following steps:

1. Scroll to the JavaScript section of the source code. Comment out all functions except for `submitValues()` and `addResponseElement()`.

2. Save the file and load it to your web browser. Now test out the program.

You should notice that your program can still work, except that after all the input fields are filled correctly, you will not be able to submit the form successfully.

This is because you have commented out the function `checkForm()`, which is needed for the second form submission.

What does this mean? This means to say that the functions `submitValues()` and `addResponseElement()` work correctly, and now it is safe to move on.

3. Now, uncomment the `checkForm()`, `buildFinalResponse()`, and `debuggingMessages()` function, save the file and reload in your browser. Continue to test out your program until you submit the form.

You should have noticed that all things go well before the submission of the second form. This is, because expected as you have tested it in **the** previous step.

Now, after you have completed all of the input fields, submit the form. Because you have uncommented **the** `checkForm()` **and** `buildFinalResponse()` functions, you should now expect a response after submitting the form.

4. Finally, uncomment the `debuggingMessages()` function. Save the file and load it in to your browser

Now, similarly, use the program as usual, and you should see that all of the required functionalities are working as before.

What just happened?

You have just executed a useful way of testing your code by uncommenting different parts of the code. You may have noticed that we started from the first function that will be used, and then proceeded to the next one. This process will help us to spot the block of code that contains the error.

This technique can also be applied to code statements. We commented out the code in functions, because it is easier to follow based on the example.

Timing differences—making sure that the HTML is there before interacting with it

Remember that the essence of JavaScript is to provide behavior to web pages by manipulating DOM elements? Here's the catch—if the HTML is not available when, for instance, a JavaScript function that changes the color of a form is executed, then the JavaScript function will not work.

In this case, it is not due to JavaScript errors such as logic, runtime, and loading errors, but rather, due to timing problems.

As mentioned in the previous chapter, the web browser (client) downloads a web page from a server, and in general, reads the web page (document) from top to bottom. So, for instance, if you have a large HTML document (for instance an HTML document with large images within the body), your JavaScript might not be able to interact with the HTML DOM because there is no HTML to interact with.

There are two solutions that allow us to deal with this problem:

1. Using the JavaScript event `onload` with the `<body>` tag. This can be done as follows:

    ```
    <html>
    <head>
    <script>
    function aSimpleFunction()
    {
        alert(window.status);
    }
    </script>
    </head>
    <body onload="aSimpleFunction()">
    </body>
    </html>
    ```

    The highlighted line means that `aSimpleFunction()` is executed only when the contents in the `<body>` tag have finished loading. You can make use of this technique to ensure that your HTML contents have finished loading before you execute your JavaScript functions.

    Here's another (and possibly preferred method):

2. Placing your JavaScript function before the `</body>` tag.

This method is commonly used; you can see companies providing analytics service often requesting its users to place the tracking code (often in JavaScript, such as Google Analytics) just before the `</body>` tag. This means that the JavaScript snippet will be loaded after all contents in the `<body>` tag are loaded, ensuring that the HTML DOM will interact with the JavaScript.

Why ad hoc testing is never enough

Up to this point, you may have noticed that the methods introduced for ad hoc testing can get repetitive when applied to your code. For instance, the `alert` method requires you to manually type in the `alert` function in different parts of the code, containing different values in order for you to inspect the code. This can get tedious and inefficient, especially when the program begins to get larger. Simply put, it will not be able to scale when the program gets too large. At the same time, the `alert` method can be quite obtrusive. For this reason, we created a simple debugging function.

The simple debugging function that we have created is less obtrusive; you can interact with the program and receive an almost instant feedback on your screen. Although it has the advantage of being less obtrusive, it suffers from two major disadvantages. The first is the fact that it can be tedious and inefficient, which is similar to the `alert` method. The second disadvantage is that how well the debugging function can work relies largely on the skills of the JavaScript program. However, being beginners in JavaScript, we may or may not have the skills to create a robust debugging function.

Therefore, there are other, more powerful, tools to help us get the job done when the need arises, and we will be discussing these in the later chapters.

Summary

In this chapter, we built upon the basics learnt in the previous chapter, and expanded our knowledge of how we can perform ad hoc testing by using various techniques covered in the chapter.

In general, we have combined the various methods and techniques from the previous chapter and this chapter in order to help us perform ad hoc testing. We often look for the required element through `getElementById`, and then by accessing form values through the `form` object. We also used the `alert()` method to perform some form of ad hoc testing.

Specifically, we have covered the following topics:

- We have learnt how to access values on forms by using the `form` object and its methods, manipulating the values, and outputting the values in to other parts of the web page by using the techniques learnt in the previous chapter, such as `getElementById`. We appended HTML content to specific HTML elements by using `.innerHTML`.

- Actions that we can take if the script does not provide the expected output, namely by testing the script by using the `alert()` method and commenting out the code. This leads us to ad hoc testing.

- Various techniques to perform ad hoc testing, most notably, by using the `alert()` method. Due to its apparent obtrusiveness, we created a simple debugging function that provides a less obtrusive way of performing testing.

- Timing differences: We must always make sure that the HTML DOM is available before JavaScript can interact with it.

- Ad hoc testing is never enough due to scalability and efficiency problems

Now that we have understood and have tried ad hoc testing, it is time to learn some slightly more advanced stuff about JavaScript testing. As mentioned earlier, although ad hoc testing is quick and simple, it does not necessarily lead to better JavaScript code (on top of its other weaknesses). In the next chapter, we'll learn about validating JavaScript. Although it sounds like a simple concept, you'll learn more JavaScript concepts in terms of the actual coding and design process, and other factors that can help you to validate your JavaScript program.

3

Syntax Validation

*To build on what we have learned in **the previous chapters, we will now move on** to a slightly tougher topic—validating JavaScript. In this chapter you can expect two broad topics—the issues surrounding validation and testing of JavaScript code, and how to use JSLint and JavaScript Lint (which is a free JavaScript validator) to check your JavaScript code, and how to debug them. I'll explicitly show you how to spot validation errors using JSLint and then, how to fix them.*

*We will **briefly mention the difference between validating and testing JavaScript** and some of the issues that you might have to consider when you are validating or testing your **code. You will also understand the relationship between valid HTML and CSS with JavaScript, and how attempting to write quality code can** help you reduce errors in your JavaScript code. More importantly, we will learn about two free tools that are often used to **validate JavaScript code, how to** make use of it to check your code**, and most importantly, how to fix validation** errors that are **detected.***

In this chapter we shall learn about the following topics:

◆ The difference between validating and testing
◆ How a good code editor can help you spot validation errors
◆ What makes a code quality code
◆ Why do we need HTML and CSS to be valid before we start working on JavaScript
◆ Why JavaScript embedded in HTML may be reported as invalid
◆ Common JavaScript errors that are detected by validating
◆ JSLint and JavaScript Lint—how to use it to check your code
◆ Valid code constructs that produce validation warnings
◆ How to fix validation errors that are spotted by JSLint

So without further ado, let us get started with a lighter topic—the difference between validating and testing.

The difference between validating and testing

There's a thin line separating validating and testing. If you have some idea about sets (as in sets from mathematics), I would say that validation can lead to better testing results, while testing does not necessarily lead to a valid code.

Let us consider the scenario—you wrote a JavaScript program and tested it on major browsers such as the Internet Explorer and Firefox; and it worked. In this case, you have tested the code to make sure that it is functional.

However, the same code that you have created may or may not be valid; valid code is akin to writing a code that has the following characteristics:

◆ Well formed

◆ Has good coding style (such as proper indentation, well-commented code, properly spaced)

◆ Meets the specification of the language (in our case, JavaScript)

> There may come a point in time where you will notice that good coding style is highly subjective—there are various validators that may have different opinions or standards as to what is known as "good coding style". Therefore, if you do use different validators to validate your code, do not freak out if you see different advice for your coding style.

This does not mean that valid code leads to code that is functional (as you will see later) and that code that is functional leads to validated code as both have different standards for comparison.

However, valid code often leads to less errors, and code that is both functional and valid is often quality code. This is due to the fact that writing a piece of JavaScript code, that is both valid and correct, is much more difficult than just writing a code that is correct.

Testing often means that we are trying to get the code working correctly; while validation is making sure that the code is syntactically correct, with good style and that it meets the specification of the language. While good coding styles may be subjective, there is often a coding style that is accepted by most programmers, such as, making sure that the code is properly commented, indented, and there is no pollution of the global namespace (especially in the case of JavaScript).

To make the case clearer, following are three situations that you can consider:

Code that is valid but wrong–validation doesn't find all the errors

This form of errors would most probably be caused by logic errors in JavaScript. Consider what we have learned in the previous chapters; logic errors can be syntactically correct but they may be logically flawed.

A classic example would be an infinite `for` loop or infinite `while` loop.

Code that is invalid but right

This would most probably be the case for most functional code; a piece of JavaScript may be functionally correct and working, but it may be invalid. This may be due to poor coding style or any other characteristics in a valid code that are missing.

Later on in this chapter, you will see a full working example of a piece of JavaScript code that is right but invalid.

Code that is invalid and wrong–validation finds some errors that might be difficult to spot any other way

In this case, the code error can be caused by all three forms of JavaScript errors that are mentioned in *Chapter 1*, *What is JavaScript Testing*, loading errors, runtime errors, and logic errors. While it is more likely that errors caused by syntax errors might be spotted by good validators, it is also possible that some errors are buried deep inside the code, such that it is difficult to spot them using manual methods.

Now that we have some common understanding as to what validation and testing is about, let us move on to the next section which discusses the issues surrounding quality code.

Code quality

While there are many views as to what is quality code, I personally believe that there are a few agreed standards. Some of the most commonly mentioned standards may include code readability, ease of extension, efficiency, good coding style, and meeting language specifications, and so on.

For our purpose here, we will focus on the factors that make a piece of code valid—coding style and meeting specifications. In general, good coding style almost guarantees that the code is highly readable (even to third parties) and this will help us to spot errors manually.

Most importantly, having a good coding style allows us to quickly understand the code, specially if we need to work in teams or are required to debug the code on our own.

You will notice that we will focus on the importance of code validity for testing purposes in later parts of the chapter. But now, let us start with the first building block of quality code—valid HTML and CSS.

HTML and CSS needs to be valid before you start on JavaScript

In chapter one, we have a common understanding that JavaScript breathes life into a web page by manipulating the Document Object Model (DOM) of the HTML documents. This means that the DOM must be present in the code before JavaScript can operate on it.

> Heres an important fact that is directly related to HTML, CSS, and browsers—browsers are generally forgiving towards invalid HTML and CSS code as compared to compilers for languages like C or Python. This is because, all browsers have to do is parse the HTML and CSS so as to render the web page for its browsers. On the other hand, compilers are generally unforgiving towards invalid code. Any missing tag, declarations, and so on will lead to a compilation error. Therefore, it is ok to write invalid or even buggy HTML and CSS, yet get a "usual" looking web page.

Based on the previous explanation, we should see that we would need to have valid HTML and CSS in order to create quality JavaScript code.

A short list of reasons, based on my personal experience, as to why valid HTML and CSS is an important prerequisite before you start working on JavaScript are as follows:

◆ Valid HTML and CSS helps ensure that JavaScript works as intended. For example, consider a situation where you might have two div elements that have the same `id` (In the previous chapters, we have mentioned that the `div id` attribute is meant to give unique IDs for each HTML elements), and your JavaScript contains the piece of code that is supposed to work on the above mentioned HTML element with the id. This will result in unintended consequences.

◆ Valid HTML and CSS helps improve the predictability on how your web page will work; there is no point trying to fix buggy HTML or CSS using JavaScript. You are most probably better off if you start with valid HTML and CSS, and then apply JavaScript.

◆ Invalid HTML and CSS may result in different behaviour in different browsers. For example, an HTML tag that is not enclosed may be rendered differently in different browsers.

In short, one of the most important building blocks of creating quality JavaScript code is to have valid HTML and CSS.

What happens if you don't validate your code

You may disagree with me on the previous section as to why HTML and CSS should be valid. In general, validation helps you to prevent errors that are related to coding style and specifications. However, do take note that using different validators may give you different results since validators might have different standards in terms of code style.

In case you are wondering if invalid code can affect your JavaScript code, I would advise you to make your code as valid as possible; invalid code may lead to sticky issues such as cross-browser incompatibility, difficulty in reading code, and so on.

Invalidated code means that your code may not be foolproof; in the early days of the Internet, there were websites that were dependent on the quirks of the early Netscape browser. Back track to the time where the Internet Explorer 6 was widely used, there were also many websites that worked in quirks mode to support Internet Explorer 6.

Now, most browsers are supporting or are moving towards supporting web standards (though slightly different, they are supporting in subtle manners), writing valid code is one of the best ways to ensure that your website works and appears the way it is intended to.

How validation can simplify testing

While invalid code may not cause your code to be dysfunctional, valid code often simplifies testing. This is due to the focus on coding style and specifications; codes that are valid and have met specifications are typically more likely to be correct and much easier to debug. Consider the following code that is stylistically invalid:

```
function checkForm(formObj){
alert(formObj.id)
//alert(formObj.text.value);
var totalFormNumber = document.forms.length;
// check if form elements are empty and are digits
var maxCounter = formObj.length; // this is for checking for empty
values
alert(totalFormNumber);
// check if the form is properly filled in order to proceed
if(checkInput(formObj)== false){
alert("Fields cannot be empty and it must be digits!");
// stop executing the code since the input is invalid
return false;
}
else{
;
}
var i = 0;
var formID;
```

```
while(i < totalFormNumber){
if(formObj == document.forms[i]){
formID = i;alert(i);
}
i++;
}
if(formID<4){
formID++;
var formToBeChanged = document.forms[formID].id;
// alert(formToBeChanged);
showForm(formToBeChanged);
}
else{
// this else statement deals with the last form
// and we need to manipulate other HTML elements
document.getElementById("formResponse").style.visibility = "visible";
}
return false;
}
```

Find the preceding code familiar? Or did you fail to recognize that the previous code snippet was taken from *Chapter 2*, *Ad Hoc Testing and Debugging in JavaScript*.

The previous code is an extreme example of poor code style, especially in terms of indentation. **Imagine if you have to manually debug the second code snippet that you saw earlier! I am pretty sure that you will find it frustrating to check the code, because you will** have little visual sense of what is going on.

More importantly, if you are **working in a team, you will be required to write legible code;** in short, writing valid code typically leads to **code that is more legible, easier to follow, and** hence, **less erroneous.**

Validation can help you debug your code

As mentioned in the previous section, browsers are in general forgiving towards invalid HTML and CSS. While this **is true, there may be errors that are not caught, or are not** rendered correctly or gracefully. **This means that while the invalid HTML and CSS code** may appear fine **on a certain platform or browser, it may not be supported on others.**

This means that using valid code (valid code typically means standard code set by international organizations such as **W3C) will give you a much greater probability of** having your web page rendered correctly **on different browsers and platforms.**

With valid HTML and CSS, you can safely write your JavaScript code and expect it to work as intended, assuming that your JavaScript code is equally valid and error free.

Validation helps you to code using good practices

Valid code typically requires coding using good practices. As mentioned frequently in this chapter, good practices include the proper enclosing of tags, suitable indentation to enhance code readability, and so on.

If you need more information about good practices when using JavaScript, feel free to check out the creator of JSLint, Douglas Crockford, at `http://crockford.com`. Or you can read up John Resigs blog (the creator of JQuery) at `http://ejohn.org/`. Both are great guys who know what great JavaScript is about.

Validation

To summarize the above sections, the DOM is provided by HTML, and both CSS and JavaScript are applied to the DOM. This means that if there is an invalid DOM, there is a chance that the JavaScript that is operating on the DOM (and sometimes the CSS) might result in errors.

With this summary in mind, well focus on how you can spot validation errors by using color coding editors.

Color-coding editors—how your editor can help you to spot validation errors

If you are an experienced coder, you may skip this section; if not, you might want to understand the value of a good coding editor.

In general, a good editor can help you to prevent validation errors. Based on our understanding of what validation is, you should understand that your editor should do the following activities:

◆ Highlight matching brackets

◆ Multiple syntax highlighting

◆ Auto indentation after keywords, brackets, and others

◆ Auto completion of syntax

◆ Auto completion of words that you have already typed

You may have noticed that I have left out a few points, or added a few points, as to what a good editor should do. But in general, the points listed previously are meant to help you prevent validation errors.

> As a start, you can consider using Microsofts SharePoint Designer 2007, a free, feature-rich, HTML, CSS ,and JavaScript editor, which is available at `http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=baa3ad86-bfc1-4bd4-9812-d9e710d44f42`

For example, highlighting matching brackets is to ensure that your code is properly enclosed with brackets, and auto indentation is to ensure that you are using consistent spacing for your code blocks.

Although JavaScripts code blocks are often denoted by the use of curly brackets, it is important that we use indentation to visually display the structure of the code. Consider the following code snippets:

```javascript
function submitValues(elementObj){
    var digits = /^\d+$/.test(elementObj.value);
    var characters =  /^[a-zA-Z\s]*$/.test(elementObj.value);
    if(elementObj.value==""){
        alert("input is empty");
        return false;
    }
    else if(elementObj.name == "enterNumber" && digits == false){
        alert("the input must be a digit!");
        debuggingMessages(arguments.callee.name, elementObj, "INPUT
must be digit");

        return false;
    }
    else if(elementObj.name == "enterText" && characters == false){
        alert("the input must be characters only!");
        return false;
    }
    else{
        elementObj.disabled = true;
         return true;

    }
}
```

The next code snippet is as follows:

```javascript
function submitValues(elementObj)
{
var digits = /^\d+$/.test(elementObj.value);
var characters =  /^[a-zA-Z\s]*$/.test(elementObj.value);
if(elementObj.value=="")
{alert("input is empty");
return false;
```

```
}
else if(elementObj.name == "enterNumber" && digits == false)
{alert("the input must be a digit!");
return false;
}else if(elementObj.name == "enterText" && characters == false)
{alert("the input must be characters only!");
return false;
}
else
{
elementObj.disabled = true;
return true;
}
}
```

I am quite sure that you would find the second code snippet to be messy, as it has inconsistent indentation, **and you may have problems figuring out which statement** belongs **to which conditional block.**

Stylistically speaking, the second code sample is what we call "poor code style". You will be surprised that this might lead to validation errors.

In case you are wondering what /^[a-zA-Z\s]*$/ and /^\d+$/are, they are actually **regular expression objects. Regular expressions originated from** Perl (a programing language) and, due to their usefulness, many programing languages now have ther own form of regular expressions. Most of them work in the same way. If you wish to find out more about regular expressions for JavaScript, feel free to visit `http://www.w3schools.com/jsref/jsref_obj_regexp.asp` for a brief introduction to how regular expressions work.

Common errors in JavaScript that will be picked up by validation

Ill briefly mention some of the most common validation errors that are picked up by validators. Following is a short list of them:

- ◆ Inconsistent spacing or indentation
- ◆ Missing semi colons
- ◆ Missing closing brackets
- ◆ Using a function or variable that is **not declared at the point of being called** or referenced

> You may have noticed that some of the validation errors are not exactly "errors"—as in syntax errors—but rather stylistic ones. As mentioned in the previous sections, differences in coding style do not necessarily lead to functional errors but rather stylistic errors. But the good thing about good coding style is that it often leads to less errors.

At this point, it might be difficult for you to visualize what these common errors actually look like. But don't worry, you will get to see such validation errors in action when we introduce the JavaScript validation tools.

JSLint–an online validator

JSLint is the first JavaScript validation code that we will focus on. You can access JSLint by visiting this URL: `http://www.jslint.com`. The JSLint online validator is a tool created by Douglas Crockford.

> Douglas Crockford works at Yahoo! as a JavaScript architect. He is also a member of the committee that designs future versions of JavaScript. His views on JavaScript style and coding practices are generally agreed upon . You can read more about him and his ideas at his website: `http://www.crockford.com`.

In general, JSLint is an online JavaScript validator. It helps to validate your code. At the same time, JSLint is smart enough to detect some forms of code errors, such as infinite loops. The JSLint website is not a particularly large website, but nonetheless, two important links that you must read are as follows:

- For basic instructions, visit `http://www.jslint.com/lint.html`
- For a list of messages, visit `http://www.jslint.com/msgs.html`

I will not attempt to describe to you what JSLint is about and how to use it; I personally believe in getting our hands dirty and giving it a test drive. Hence, for a start, we'll test the code that we wrote in *Chapter 2*, *Ad Hoc Testing and Debugging in JavaScript*, and see what kind of validation errors (if any) occur.

Time for action – using JSLint to spot validation errors

As mentioned earlier, well test the code that we wrote in *Chapter 2*, *Ad Hoc Testing and Debugging in JavaScript*, and see what validation errors we get. Take note that the completed and validated code for this example can be found in `Chapter 3` of the `source code` folder, in the file named `perfect-code-for-JSLint.html`.

1. Open up your web browser and navigate to `http://www.jslint.com`. You should see the home page with a huge text area. This is the area where you are going to copy and paste your code.

2. Go to the `source code` folder of `Chapter 2` and open up the file named: `getting-values-in-right-places-complete.html`. Then, copy and paste the source code into the text area mentioned in step 1.

3. Now click on the button with the name **JSLint**.

Your page should refresh almost immediately, and you will receive some form of feedback. You may have noticed that you received many (yes, a lot of) validation errors. And, most probably, some of them do not make sense to you. However, you should be able to identify that some of the validation errors were introduced in the section on common JavaScript validation errors.

Now, scroll further down and you should see the following phrase in the feedback area:

```
xx % scanned
too many errors
```

This tells you that JSLint has only scanned a part of the code and stopped scanning the code because there were too many errors.

What can we do about this? What if there are too many validation errors and you cannot spot all of them in one go?

Do not worry, as JSLint is robust and has option settings, which are found at `http://www.jslint.com/#JSLINT_OPTIONS` (this is actually found at the bottom of the home page of JSLint). One of the options that requires your input is the **maximum number of errors**. For our purposes, you may want to enter an insanely large number, such as 1,000,000.

4. After entering an insanely large number for the input box for **maximum number of errors**, click on the button **The good parts**. You will see a few checkboxes have been selected.

After step 4, you have now officially selected the options known as **The Good Parts** by the author of this tool. This is a setting that automatically sets what the author feels are the most important validation checks.

These options include: **Strict white space**, **allow one var statement per function**, and so on.

5. Now click on the **JSLint** button. Your browser will show the new validated result. Now you may take a look at the types of validation errors that have been detected by JSLint.

What just happened?

You have just used JSLint to spot for validation errors. This is a simple process for JSLint: copy and paste your code into the text area and click on **JSLint**. Do not be surprised that there are so many validation errors; we are just starting out and we will learn how to fix and avoid such validation errors.
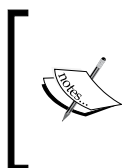
> You may have noticed that the JavaScript that is embedded in the HTML form results in an error that says **missing use strict statement**. This error stems from the fact that JSLint believes in the use of the **use strict** statement, which allows the code to run under strict conditions. You will learn how to fix and avoid such problems in later parts of this chapter.

You will continue to see many errors. In my opinion, this is evidence that valid code is not easy to achieve; but this is what we will achieve in the next section.

As you have seen, there are various validation options, and at this stage, it is good enough if we pass the set requirements for **The Good Parts**. Therefore, well focus on how to fix these validation errors in the next section. But before that, I'll quickly discuss the valid code constructs that produce validation warnings.

Valid code constructs that produce validation warnings

You may have noticed that although our code construct is valid, it has produced validation warnings. You may be wondering if you should fix these or not. Here's some basic discussion to help you to decide.

Should you fix valid code constructs that produce validation warnings?

This depends on your objective. As I mentioned in *Chapter 1*, *What is Javascript Testing?*, a code should at least be correct and work the way that we intend it to. Therefore, if your objective is to just create functionally-correct code, then you might not want to spend the time and effort to correct those validation warnings.

However, because you are reading this book, it is very likely that you want to learn how to test JavaScript, and validation is an important part of testing JavaScript as you will see later in this chapter.

What happens if you don't fix them

The main issue with invalidated code is that it will be much more difficult to maintain the code, in terms of readability and scalability. This problem becomes enhanced when you are working in teams and others have to read or maintain your code.

Valid code promotes good coding practices, which will help you to avoid problems down the road.

How to fix validation errors

This section will continue with the errors mentioned in the previous section, and together we'll attempt to fix them. Wherever possible, I'll provide some form of explanation as to why a particular piece of code is rendered as invalid. At the same time, the whole process of writing valid and functionally-code can be cumbersome. Therefore, I'll start off with validation errors that are much easier to fix, before I move on to tougher ones.

As we go along fixing the validation errors that we saw in the previous section, you may realize that fixing validation errors may require some form of compromise as to how you write your code. For example, you will understand that using `alert()` sparingly in your code is not considered a good coding style, at least according to JSLint. In this case, you will have to consolidate all of your `alert()` statements and group them into a function, while still maintaining the functionality of your code. More importantly, you will also realize that (perhaps) the best way to write valid code is to start writing valid code right from the first line of the code; you will see that correcting invalid code is an extremely tedious process, and there are times when you can only minimize your validation errors.

Along the way, you will get the chance to practice important JavaScript functions and, at the same time, learn how to code in a better style. Thus, this is probably the most important section of this chapter and I urge you to get your hands dirty with me. Before I get started on fixing the code, I'll first summarize the types of errors that are spotted by JSLint.

- Missing "`use strict`" statement
- Unexpected use of `++`
- Missing space after `)`, `value`, `==`, `if`, `else`, `+`

- ◆ Function names (such as debuggingMessages) are not defined or a function was used before it was defined
- ◆ Too many `var` statements
- ◆ `===` used instead of `==`
- ◆ `alert` is not defined
- ◆ `<\/` used instead of `</`
- ◆ HTML event handlers used

Without further ado, well get started with the first validation error: `use strict`.

Error—missing "use strict" statement

The "`use strict`" statement is a relatively new feature in JavaScript that allows our JavaScript to run in a strict environment. In general, it catches little-known errors, and "forces" you to write stricter and valid code. John Resig, an expert in JavaScript, has written a nice summary about this topic, and you can read about it by following this link: `http://ejohn.org/blog/ecmascript-5-strict-mode-json-and-more/`.

Time for action – fixing "use strict" errors

This error is extremely easy to fix. But be careful; enabling `use strict` may prevent your code from working, if your code is not valid. Here's how we can fix this validation error:

1. Open up your text editor, copy and paste the same code that we have been using, and append the following code snippet on the first line of your JavaScript code:

   ```
   "use strict";
   ```

2. Save your code and test it out on JSLint. You will see that the error is now gone.

You may notice that there is another missing `use strict` error that is related to your HTML form; do not worry, we will fix that in a later sub-section of this chapter. Now let us move on to the next error.

Error—unexpected use of ++

There is nothing programmatically wrong with this line of code. What we intend to achieve by using `++` is to increment `globalCounter` whenever the function `addResponseElement()` is called.

However, JSLInt believes that there is a problem with using ++. Take the following code snippets as an example:

```
var testing = globalCounter++ + ++someValues;
var testing2 = ++globalCounter + someValues++;
```

The previous statements would look confusing to most programmers and hence it is considered bad style. More importantly, both of these statements are programmatically different and produce different results. For these reasons, we need to avoid statements like ++, --, and so on.

Time for action – fixing the error of "Unexpected use of ++"

This error is relatively easy to fix. All we need to do is avoid ++. So navigate to the addResponseElement() function, and look for globalCounter++. Then change globalCounter++ to globalCounter = globalCounter + 1. So, now your function has changed from this:

```
function addResponseElement(messageValue, idName){
    globalCounter++;
    var totalInputElements = document.testForm.length;
    debuggingMessages( addResponseElement","empty", "object is a
value");
    var container = document.getElementById('formSubmit');
    container.innerHTML += "<input type=\"text\" value=\""
+messageValue+ "\"name=\""+idName+"\" /><br>";
    if(globalCounter == totalInputElements){
        container.innerHTML += "<input type=\"submit\" value=\
"Submit\" />";
    }
}
```

To this:

```
function addResponseElement(messageValue, idName) {
    globalCounter = globalCounter + 1;
    debuggingMessages( "addResponseElement", "empty", "object is a
value");
    document.getElementById('formSubmit').innerHTML += "<input
type=\"text\" value=\"" + messageValue + "\"name = \"" + idName + "\"
/><br>";
    if (globalCounter === 7) {
        document.getElementById('formSubmit').innerHTML += "<input
type=\"submit\" value=\"Submit\" />";
    }
}
```

Compare the highlighted lines, and you will see the change in the code. Now let us move on to the next error.

Error—functions not defined

This error is caused by the way that JavaScript engines and web pages are being rendered by web browsers. In *Chapter 1*, *What is Javascript Testing*, we mentioned briefly that web page (and JavaScript) are being parsed from top to bottom on the client side. This means that anything that appears at the top will be read first, followed by that at the bottom.

Time for action – fixing the error of "Functions not defined"

1. Because this error is caused by the incorrect flow of the JavaScript functions, we will need to change the sequence of the functions. What we have done in *Chapter 2*, *Ad Hoc Testing and Degugging in Javascript*, is that we wrote the functions that we will be using first. This may be incorrect, as the functions may require data or functions that are only defined in later parts of the JavaScript code. Here's a very simplified example:

```
<script>
function addTWoNumbers() {
    return numberOne() + numberTwo();
}
function numberOne(x, y) {
    return x + y;
}
function numberTwo(a, b){
    return a + b;
}
</script>
```

Based on the previous code snippet, you will realize that `addTwoNumbers()` requires data returned from `numberOne()` and `numberTwo()`. The issue here is that the JavaScript interpreter will read `addTwoNumbers()` first before reading `numberOne()` and `numberTwo()`. However, both `numberOne()` and `numberTwo()` are being called by `addTwoNumbers()`, resulting in an incorrect flow of code.

This means that in order for our code to work correctly, we will need to rearrange the order of the functions. Continuing with the previous example, this is what we should do:

```
<script>
function numberOne(x, y) {
    return x + y;
}
function numberTwo(a, b){
    return a + b;
}function addTWoNumbers() {
    return numberOne() + numberTwo();
}
</script>
```

In the previous code snippet, we have rearranged the sequence of the functions.

2. Now, we are going to rearrange the function's sequence. For our purposes, all that we need to do is to arrange our functions such that the first function that originally appeared in our code will now be the last, and the last function will be the first. Similarly, the second function that originally appeared in the JavaScript code will now be the second-to-last function. In other words, we will reverse the order of the code.

3. Once you have reversed the order of the functions, save the file and test the code on JSLint. You should notice that the validation errors relating to functions not being defined are now gone.

Now, let us move on to the next validation error.

Too many var statements

According to JSLint, we have used too many `var` statements. What does this mean? This means that we have used more than one `var` statement in each function; in our case we have obviously used more than one `var` statement in each and every function.

How did this happen? If you scroll down and check the settings of JSLint, you will see a checkbox selected that says **Allow one var statement per function**. This means that the maximum number of `var` we can use is one.

Why is this considered to be good style? Although many coders may think that this is cumbersome, the author of JSLint would most probably believe that a good function should do only one thing. This would typically mean operating on only one variable.

There's certainly room for discussion, but as we are all here to learn, let us get our hands dirty by fixing this validation error.

Time for action – fixing the error of using too many var statements

In order to fix this error, we will need to do some form of code refactoring. Although code refactoring typically means consolidating your code for it to become more concise (that is, shorter code), you may realize that refactoring your code to fit validation standards is a lot of work.

1. What we will do in this section is that we will change (almost) all single `var` statements that save a value into a function.

The code that is mainly responsible for this particular validation error is found `checkForm` function. The statements that we will need to refactor are as follows:

```
    var totalInputElements = document.testFormResponse.length;
 var nameOfPerson = document.testFormResponse.nameOfPerson.value;
 var birth = document.testFormResponse.birth.value;
 var age = document.testFormResponse.age.value;
 var spending = document.testFormResponse.spending.value;
 var salary = document.testFormResponse.salary.value;
 var retire = document.testFormResponse.retire.value;
 var retirementMoney = document.testFormResponse.retirementMoney.
value;
 var confirmedSavingsByRetirement;
 var ageDifference = retire - age;

 var salaryPerYear = salary * 12;

 var spendingPerYear = spending * 12;

 var incomeDifference = salaryPerYear - spendingPerYear;
```

2. Now we'll start to refactor our code. For each of the variables defined, we need to define a function with the following format:

```
function nameOfVariable(){
    return x + y; // x + y represents some form of calculation
}
```

I'll start off with an example. For instance, for `totalInputElements` this is what I will do:

```
function totalInputElements() {
    return document.testFormResponse.length;
}
```

3. Based on the previous code, do something similar to what you are going to see here:

```
/* here are the function for all the values */
function totalInputElements() {
    return document.testFormResponse.length;
}

function nameOfPerson() {
    return document.testFormResponse.nameOfPerson.value;
}

function birth() {
    return document.testFormResponse.birth.value;
}

function age() {
    return document.testFormResponse.age.value;
}

function spending() {
    return document.testFormResponse.spending.value;
}

function salary() {
    return document.testFormResponse.salary.value;
}

function retire() {
    return document.testFormResponse.retire.value;
}

function retirementMoney() {
    return document.testFormResponse.retirementMoney.value;
}

function salaryPerYear() {
    return salary() * 12;
}

function spendingPerYear() {
    return spending() * 12;
}

function ageDifference() {
```

```
        return retire() - age();
    }

    function incomeDifference() {
        return salaryPerYear() - spendingPerYear();
    }

    function confirmedSavingsByRetirement() {
        return incomeDifference() * ageDifference();
    }

    function shortChange() {
        return retirementMoney() - confirmedSavingsByRetirement();
    }

    function yearsNeeded() {
        return shortChange() / 12;
    }

    function excessMoney() {
        return confirmedSavingsByRetirement() - retirementMoney();
    }
```

Now, let us move on to the next error.

Expecting <\/ instead of <\

For most of us, this error is probably one of the most intriguing. We have this validation error because the HTML parser is slightly different to the JavaScript interpreter. In general, the extra backslash is being ignored by the JavaScript compiler, but not by the HTML parser.

Such validation errors may appear unnecessary, but Doug Crockford knows that this has some form of impact on our web page. Therefore, let us move on to how to fix this error.

Time for action – fixing the expectation of '<\/' instead of '</'

Although this error is one of the most intriguing, it is one of the easiest to fix. All that we need to do is to find all of the JavaScript statements that contain `</` and change them to `<\/`. The function that is mainly responsible for this error is `buildFinalResponse()`.

1. Scroll down to the function `buildFinalResponse()`, and change all statements that have `</` to `<\/`. After you are done, you should have the following code:

```
function buildFinalResponse(name, retiring, yearsNeeded, retire,
shortChange) {
    debuggingMessages( buildFinalResponse", -1, "no messages");

    var element = document.getElementById("finalResponse");

    if (retiring === false) {
        element.innerHTML += "<p>Hi <b>" + name + "<\/b>,<\/p>";

        element.innerHTML += "<p>We've processed your information
and we have noticed a problem.<\/p>";

        element.innerHTML += "<p>Base on your current spending
habits, you will not be able to retire by <b>" + retire + " <\/b>
years old.<\/p>";

        element.innerHTML += "<p>You need to make another <b>" +
shortChange + "<\/b> dollars before you retire inorder to acheive
our goal<\/p>";

        element.innerHTML += "<p>You either have to increase your
income or decrease your spending.<\/p>";
    }
    else {
        // able to retire but....
        //alertMessage("retiring === true");

        element.innerHTML += "<p>Hi <b>" + name + "<\/b>,<\/p>";
        element.innerHTML += "<p>We've processed your information
and are pleased to announce that you will be able to retire on
time.<\/p>";
        element.innerHTML += "<p>Base on your current spending
habits, you will be able to retire by <b>" + retire + "<\/b>years
old.<\/p>";
        element.innerHTML += "<p>Also, you'll have' <b>" +
shortChange + "<\/b> amount of excess cash when you retire.<\/p>";
        element.innerHTML += "<p>Congrats!<\/p>";
    }
}
```

Notice that all `</` have **been changed to** `<\/`. You may also want to search through the code and see if any such errors are remaining.

Now, with this error fixed, we can move on to the next validation error.

Expected '===' but found '=='

In JavaScript and in most programing languages, `==` and `===` are significantly different. In general, `===` is stricter than `==`.
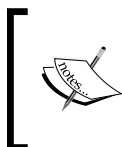
> The key difference between `===` and `==` in JavaScript is that `===` is a strict equal operator and it will return a Boolean true if, and only if, both the operands are equal **and of the same type. On the other hand, the** `==` operator returns a Boolean true if both the operands are equal, even if they are of different types.

According to JSList, `===` should be used when comparing a variable to a truth value, because it is stricter than `==`. In terms of code strictness, JSLint is certainly correct in ensuring code quality. Therefore, let us now correct this error.

Time for action – changing == to ===

Due to the reasons mentioned earlier, we will now change all `==` to `===`, **for statements that** require comparison. Although this error is relatively easy to fix, we need to understand the importance of this error. `===` is much stricter than `==`, and therefore it is more secure and valid to use `===` **instead of** `==`.

Going back to your source code, search for all comparison statements that contain `==` **and** change them to `===`. `==` is found largely at `if`, **and** `else-if` statements, because it is used for comparisons.

Once you are done, you may want to test out your updated code at JSLint and see if you have cleared all such errors.

Now, let us move on to yet another cumbersome error: "Alert is not defined".

Alert is not defined

In general, using `alert` by itself leads to 'pollution' of the global namespace. Although it is convenient, this is bad code practice according to JSLInt. Therefore, the strategy that we are going to use to fix this validation error is to use some form of code refactoring (again).

In our code, you should notice that we are largely using `alert()` to provide feedback in terms of the function names, error messages, and so on. We will need to use our `alert()` such that it can take in various forms of messages.

Time for action – fixing "Alert is not defined"

What we will do is that we will consolidate all `alert()` statements into one function. We can pass a parameter to that function so that we can change the messages in the alert box depending on the situation.

1. Go back to your code, and define the following function at the top of your `<script>` tag:

```
function alertMessage(messageObject) {
    alert(messageObject);
    return true;
}
```

`messageObject` is the parameter that we will use to hold our message.

2. Now, change all `alert()` statements to `alertMessage()` such that the message for `alert()` is the same as `alertMessage()`. Once you are done, save the file and run the code in JSLint again.

If you tried running your code in JSLint, you should see that the "damage" done by `alert()` has been minimized to only one time, instead of over ten to twenty times.

In this situation, what we can do is minimize the impact of the `alert()` because, for our purposes, we do not have a ready alternative to show messages in an alert box.

Now it is time for the next error—avoiding HTML event handlers.

Avoiding HTML event handlers

Good coding practices often state the need to separate programing logic and design. In our case, we have embedded event handlers (JavaScript events) within the HTML code. According to JSLint, such coding could be improved by avoiding HTML event handlers altogether.

> Although the ideal case is to separate programing logic from design, there is nothing functionally wrong in using HTML intrinsic event handlers. You may want to consider whether it is worth it (in terms of time, maintainability, and scalability) to adhere to (almost) perfect coding practices. In the later part of this sub-section, you may find that it can be cumbersome (or even irritating) to try to validate (and functionally correct) code.

In order to solve this validation error, we will need to use event listeners. However, due to the problems posed by the compatibility of event listeners, we will be using JavaScript libraries to help us to deal with inconsistencies among the support for event listeners. We will be using JQuery in this example.

JQuery is a JavaScript library created by John Resig. You can download JQuery by visiting this link: `http://jquery.com`. As described on this website, "JQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, and animating, and Ajax interactions for rapid web development." In my personal experience, JQuery certainly makes life easier by fixing many sticky issues such as DOM incompatibilities, providing built-in methods to create animation, and many other things. I certainly urge you to follow a starter tutorial by going to: `http://docs.jquery.com/Tutorials:` `Getting_Started_with_jQuery`

Time for action – avoiding HTML event handlers

In this section, you will learn how to avoid HTML event handlers by coding in a different style. In this case, we will not only remove the JavaScript events embedded in each of the HTML input elements, we will also need to write new functions for our JavaScript application in order for it to work in the same manner. In addition to that, we will be using a JavaScript library that will help us to remove all of the difficult stuff relating to event handling and using event listeners.

1. Open up the same document and scroll to the `<body>` tags. Remove all of the HTML event handlers that are found in the form. This is what your form's source code should look like after you have removed all of the HTML event handlers:

```
<form name="testForm" >

    <input type="text" name="enterText" id="nameOfPerson"
size="50"  value="Enter your name"/><br>

    <input type="text" name="enterText" id="birth" size="50"
value="Enter your place of birth"/><br>

    <input type="text" name="enterNumber" id="age" size="50"
maxlength="2" value="Enter your age"/><br>

    <input type="text" name="enterNumber" id="spending" size="50"
value="Enter your spending per month"/><br>

    <input type="text" name="enterNumber" id="salary" size="50"
value="Enter your salary per month"/><br>
```

```
    <input type="text" name="enterNumber" id="retire" size="50"
maxlength="3" value="Enter your the age you wish to retire at"
/><br>

    <input type="text" name="enterNumber" id="retirementMoney"
size="50" value="Enter the amount of money you wish to have for
retirement"/><br>

</form>
```

2. Now scroll to the `</style>` tag. After the `</style>` tag, enter the following code snippet:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/
jquery.js">
</script>
```

What you are doing in the preceding line is enabling JQuery in your code. This will allow you to make use of the JQuery library when fixing your code. Now it's time to write some JQuery code.

3. In order to maintain the functionality of our code, we will need to use the `.blur()` method provided by JQuery. Scrolling to the end of your JavaScript code, append the following code snippet:

```
jQuery(document).ready(function () {
    jQuery('#nameOfPerson').blur(function () {
        submitValues(this);
    });
    jQuery('#birth').blur(function () {
        submitValues(this);
    });
    jQuery('#age').blur(function () {
        submitValues(this);
    });
    jQuery('#spending').blur(function () {
        submitValues(this);
    });
    jQuery('#salary').blur(function () {
        submitValues(this);
    });
    jQuery('#retire').blur(function () {
        submitValues(this);
    });
    jQuery('#retirementMoney').blur(function () {
        submitValues(this);
    });
```

```
            jQuery('#formSubmit').submit(function () {

                checkForm(this);

                return false;
        });
    });
```

Here's a short explanation of how JQuery works: `jQuery(document).`
`ready(function ()` is used to start our code; it allows us to use
the methods provided in JQuery. In order to select an element, we
use `jQuery('#nameOfPerson')`. As mentioned earlier, we need to
maintain the functionality of the code, so we will use the `.blur()`
method provided by JQuery. In order to do that, we append `.blur()` to
`jQuery('#nameOfPerson')`. We are required to call `submitValues()`,
and we will need to enclose `submitValues()` within `.blur()`. Because
`submitValues()` is a function, we will enclose it as such:

```
jQuery('#nameOfPerson').blur(function () {
    submitValues(this);
});
```

At this point of time, we should have completed the necessary corrections
in order to achieve valid and functional code. I'll briefly summarize the
corrections in the next section.

Summary of the corrections we have done

Now we will refresh our memory by quickly going through what we have done to fix the
validation errors.

First, we pasted the original code into JSLint and noticed that we had a large number of
validation errors. Fortunately, the validation errors could be grouped such that similar
errors could be fixed by correcting a single code error.

Next, we started off with the correction process. In general, we tried to fix the validation
errors, starting from those which seemed to be the easiest. The first validation error that we
fixed was the missing `use strict` statement error. What we did was enter `use strict` on
the very first line of our JavaScript code, and that error was fixed.

The second validation error that we fixed was the "functions not defined error". This was
caused by an incorrect flow of the JavaScript functions. Therefore, we switched the flow of
functions from this:

```
function submitValues(elementObj){
    /* some code omitted */


}
```

```
function addResponseElement(messageValue, idName){
    /* some code omitted */


function checkForm(formObj){
    /* some code omitted */
}

function buildFinalResponse(name,retiring,yearsNeeded,retire,
shortChange){
    /* some code omitted */
}
function debuggingMessages(functionName, objectCalled, message){
    /* some code omitted */
}
```

To this:

```
function debuggingMessages(functionName, objectCalled, message) {
    /* some code omitted */
}
function checkForm(formObj) {
    /* some code omitted */
}

function addResponseElement(messageValue, idName) {
        /* some code omitted */
}

function submitValues(elementObj) {
    /* some code omitted */
}
```

Notice that we simply reversed the sequence of the functions to fix the error.

We then moved on to an error that is quite time-consuming—using too many `var` statements within a function. In general, our strategy was to refactor almost all of the `var` statements into standalone functions. These standalone functions' main purpose was to return a value, and that's all.

Next, we moved into yet another time-consuming validation error, and this was "expected `<\/` instead of `</`. In general, this error is referring to the closing HTML tags. So what we did was to change `/>` to `\/>` for all closing HTML tags. For example, we changed the following code:

```
function buildFinalResponse(name,retiring,yearsNeeded,retire,
shortChange){
    debuggingMessages( buildFinalResponse", -1,"no messages");
```

```
        var element = document.getElementById("finalResponse");
        if(retiring == false){
            //alert("if retiring == false");
            element.innerHTML += "<p>Hi <b>" + name + "</b>,<p>";
            element.innerHTML += "<p>We've processed your information and
    we have noticed a problem.</p>";
            element.innerHTML += "<p>Base on your current spending habits,
    you will not be able to retire by <b>" + retire + " </b> years old.</
    p>";
            element.innerHTML += "<p>You need to make another <b>" +
    shortChange + "</b> dollars before you retire inorder to acheive our
    goal</p>";
            element.innerHTML += "<p>You either have to increase your
    income or decrease your spending.</p>";
        }
        else{
            // able to retire but....
            alert("retiring == true");
            element.innerHTML += "<p>Hi <b>" + name + "</b>,</p>";
            element.innerHTML += "<p>We've processed your information and
    are pleased to announce that you will be able to retire on time.</p>";
            element.innerHTML += "<p>Base on your current spending habits,
    you will be able to retire by <b>" + retire + "</b>years old.
    </p>";
            element.innerHTML += "<p>Also, you'll have' <b>" + shortChange
    + "</b> amount of excess cash when you retire.</p>";
            element.innerHTML += "<p>Congrats!<p>";
        }
    }
```

To this:

```
    function buildFinalResponse(name, retiring, yearsNeeded, retire,
    shortChange) {
        debuggingMessages( buildFinalResponse", -1, "no messages");
        var element = document.getElementById("finalResponse");
        if (retiring === false) {
            element.innerHTML += "<p>Hi <b>" + name + "<\/b>,<\/p>";
            element.innerHTML += "<p>We've processed your information and
    we have noticed a problem.<\/p>";
            element.innerHTML += "<p>Base on your current spending habits,
    you will not be able to retire by <b>" + retire + " <\/b> years
    old.<\/p>";
            element.innerHTML += "<p>You need to make another <b>" +
    shortChange + "<\/b> dollars before you retire inorder to achieve our
    goal<\/p>";
```

```
        element.innerHTML += "<p>You either have to increase your
income or decrease your spending.<\/p>";
    }
    else {
        // able to retire but....
        //alertMessage("retiring === true");
        element.innerHTML += "<p>Hi <b>" + name + "<\/b>,<\/p>";
        element.innerHTML += "<p>We've processed your information and
are pleased to announce that you will be able to retire on time.<\/
p>";
        element.innerHTML += "<p>Base on your current spending habits,
you will be able to retire by <b>" + retire + "<\/b>years old.<\/p>";
        element.innerHTML += "<p>Also, you'll have' <b>" + shortChange
+ "<\/b> amount of excess cash when you retire.<\/p>";
        element.innerHTML += "<p>Congrats!<\/p>";
    }
}
```

Note that the highlighted lines are where we have changed from `/>` to `\/>`.

After fixing the previous error, we moved on to an error that is conceptually more difficult to understand, but easy to solve. That is, "expected `===` instead of saw `==`". According to JSLint, using `===` is stricter and more secure as compared to using `==`. Therefore, we needed to change all `==` to `===`.

The next error, "`Alert` is not defined", is conceptually similar to the "Too many `var` statement" error. What we need to do is to refactor **all** `alert()` statements to call the `alertMessage()` function that accepts a parameter `messageObject`. This allows us to use only one `alert()` for almost the whole JavaScript program. Whenever we need to use an alert box, all we need to do is to pass an argument into the `alertMessage()` function.

Finally, we moved on to fix one of the toughest validation errors: "Avoiding HTML event handlers". Due to the complexities involved with event listeners, we engaged the help of JQuery, a popular JavaScript library, and wrote some JQuery code. Firstly, we removed all of the HTML event handlers from our HTML form. Our HTML form changed from this:

```
<form name="testForm" >
    <input type="text" name="enterText" id="nameOfPerson" onblur="submi
tValues(this)" size="50"  value="Enter your name"/><br>
    <input type="text" name="enterNumber" id="age" onblur="submitValues
(this)" size="50" maxlength="2" value="Enter your age"/><br>
    <input type="text" name="enterText" id="birth" onblur="submitValues
(this)" size="50"  value="Enter your place of birth"/><br>
    <input type="text" name="enterNumber" id="spending" onblur="submitV
alues(this)" size="50" value="Enter your spending per month"/><br>
```

```
    <input type="text" name="enterNumber" id="salary" onblur="submitVal
ues(this)" size="50" value="Enter your salary per month"/><br>
    <input type="text" name="enterNumber" id="retire" onblur="submitVal
ues(this)" size="50" maxlength="3" value="Enter your the age you wish
to retire at" /><br>
    <input type="text" name="enterNumber" id="retirementMoney" onblur
="submitValues(this)" size="50" value="Enter the amount of money you
wish to have for retirement"/><br>
    </form>
```

To this:

```
<form name="testForm" >

  <input type="text" name="enterText" id="nameOfPerson" size="50"
value="Enter your name"/><br>

  <input type="text" name="enterText" id="birth" size="50"
value="Enter your place of birth"/><br>

  <input type="text" name="enterNumber" id="age" size="50"
maxlength="2" value="Enter your age"/><br>

  <input type="text" name="enterNumber" id="spending" size="50"
value="Enter your spending per month"/><br>

  <input type="text" name="enterNumber" id="salary" size="50"
value="Enter your salary per month"/><br>

  <input type="text" name="enterNumber" id="retire" size="50"
maxlength="3" value="Enter your the age you wish to retire at" /><br>

  <input type="text" name="enterNumber" id="retirementMoney" size="50"
value="Enter the amount of money you wish to have for retirement"/
><br>

</form>
```

In order to support the new HTML form, we linked in the JQuery library, and added some code to listen for the HTML form events, like this:

```
<script type="text/javascript"> src="http://ajax.googleapis.com/ajax/
libs/jquery/1.4.2/jquery.js"></script>
<script type="text/javascript">
/* some code omitted */

jQuery(document).ready(function () {
```

```
        jQuery('#nameOfPerson').blur(function () {
            submitValues(this);
        });
        jQuery('#birth').blur(function () {
            submitValues(this);
        });
        jQuery('#age').blur(function () {
            submitValues(this);
        });
        jQuery('#spending').blur(function () {
            submitValues(this);
        });
        jQuery('#salary').blur(function () {
            submitValues(this);
        });
        jQuery('#retire').blur(function () {
            submitValues(this);
        });
        jQuery('#retirementMoney').blur(function () {
            submitValues(this);
        });

        jQuery('#formSubmit').submit(function () {
            checkForm(this);
            return false;
        });
    });
    </script>
```

The completed code can be found in the `source code` folder for `Chapter 3`, with a file name of `perfect-code-for-JSLint.html`. You can compare this with your edited code to see if you have understood what we were trying to do. Now, you may want to copy and paste the code into JSLint and see how it goes. You will only see errors pertaining to the use of Jquery, one validation error that complains about the use of `alert()`, and another error about using too many `var` statements.

What just happened?

We have corrected the bulk of the validation errors, from an insanely large number of validation errors to less than ten validation errors, out of which only two or three of the validation errors are related to our code.
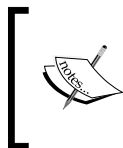
> You may have noticed the `jQuery not defined` error. Although JSLint has captured the JQuery library that was externally linked, it does not explicitly read the code, thus resulting in the `jQuery not defined` error.

Now that we have fixed the validation errors, let us now move on to another free validation tool, the JavaScript Lint.

JavaScript Lint–a tool you can download

JavaScript Lint can be downloaded at `http://www.javascriptlint.com`, **and it works in** a manner similar to JSLint. The key difference is that JavaScript Lint is a downloadable tool, whereas JSLint works as a web-based tool.

Like JSLint, JavaScript Lint is capable of spotting the following common errors:

- Missing semicolons at the end of a line
- Curly braces without an `if`, `for`, and `while`
- Statements that do not do anything
- Case statements in a switch that turn decimal points into a number

You can read more about its functionality by visiting its home page at `http://www.javascriptlint.com`.

To learn about how to use JavaScript Lint, you may follow the tutorials found at the website.

- If you are using Windows, you may need to read the set-up instructions found at `http://www.javascriptlint.com/docs/running_from_windows_explorer.htm`
- If you are using Linux based operating systems, you can check out the instructions found at `http://www.javascriptlint.com/docs/running_from_the_command_line.htm`
- Finally, if you wish to integrate JavaScript Lint into your IDE such as Visual Studio, you can read more about how to do this by visiting `http://www.javascriptlint.com/docs/running_from_your_ide.htm`

We will not be discussing "how to fix validation errors spotted by JavaScript Lint" because the principles are similar to JSLint. However, we challenge you to fix the remaining errors (apart from those caused by JQuery)

Challenge yourself–fix the remaining errors spotted by JSLint

Ok, this is the first challenge that I will I present to you. Fix the remaining errors spotted by JSLint, which are as follows:

- **alert is not defined**": This is found in the `alertMessage()` function

- **too many var statements"**: This error is found in the `submitValues()` function

Here are some ideas for you to get started:

- In our JavaScript application, is there any way that we can avoid the `alert()`? How can we display messages that can capture the attention of our audience but at the same time be valid?

- For the error found at the `submitValues()` function, how can we refactor the code such that there is only one `var` statement in the function? Can we refactor the `var` statement into a function and have it return a Boolean value?

OK, now you might want to give it a go, but be careful, because the solutions that you propose or intend to use may cause other validation errors. So you might want to think about your solutions before implementing them.

Summary

We've finally reached the end of this chapter. I'll first start off by summarizing some of the strategies and tips we have used to write valid code, and follow this with a summary of the rest of the chapter.

Some of the strategies that we have used to write valid code (according to JSLint) are as follows:

- Properly space your code, especially after mathematical signs, `if`, `else`, `( )`, and so on

- Use only one `var` statement per function

- Consider the flow of your program; code in such a way that the required data or functions come at the top of the program

- Use the `alert()` function sparingly. Instead, consolidate your `alert()` functions into one function

- Use `===` instead of `==`; this makes sure that your comparison statements are more accurate

- Avoid HTML event handlers by using listeners. Alternatively, you may engage the help of JavaScript libraries such as JQuery in order to provide event listeners to your code.

Finally, we covered the following topics:

- ◆ The difference between testing and validating
- ◆ How validation helps us to write good code
- ◆ What issues may occur if we do not validate our code—if we do not validate our code, it might not be scalable, less readable, and result in unexpected errors
- ◆ How we can use JSLint and JavaScript Lint to validate our code

Now that we have learned how we can test JavaScript by validation tools, you might want to think about the strategy that we can adopt when we intend to test our code. As shown in the example in this chapter, writing valid code (or correcting invalid code) is an extremely tedious process. More importantly, there are some validation warnings or errors that do not affect our program in its entirety. In such a situation, do you think that it is worth the effort to validate our code? Or do you think we should be a perfectionist and write perfect code? This will very much depend on our testing plan, which will dictate the scope of testing, the things to test, and many other things. These topics will be covered in next chapter, *Chapter 4*, *Planning to Test*. So I'll end off this chapter, and see you in the next chapter.

4

Planning to Test

Welcome to the fourth chapter. Before we move into a more formal testing process, we must first understand what testing is about. In this chapter, we will learn how to make a plan for testing your JavaScript program. We will learn about the various testing concepts that you should know, after which I will present to you a brief guideline which will be used as a basis for the next chapter.

Before we move into the various testing concepts, we will first need to establish a brief understanding of the following issues:

- ◆ Do we really need a test plan in order to carry out testing?
- ◆ When should we develop the test plan for our code?
- ◆ How much testing do we need for our program?

After covering the above issues, we will learn about the following testing concepts and ideas:

- ◆ Black box testing, white box testing, and related concepts
- ◆ Boundary conditions
- ◆ Unit testing
- ◆ Web page functional testing
- ◆ Integration testing
- ◆ Non-functional testing, such as performance testing
- ◆ Usability testing
- ◆ Testing order—which of the above tests do we perform first?
- ◆ Regression testing—which is typically done when we make changes to the code

In order to get a better overview of when and where testing plays its part, we will first start with a very brief introduction to the software lifecycle.

A very brief introduction to the software lifecycle

Understanding the software lifecycle will help you to develop a deeper insight into the software development process and, more importantly, when and where testing will play its part.

In general, the software lifecycle has the following stages:

1. Analysis
2. Design
3. Implementation
4. Testing
5. Deployment
6. Maintenance

In the first stage, we generally perform an analysis to understand what the needs of the stakeholders are. For instance, if you are carrying out a customized project for a customer, you will need to understand the user requirements, system requirements, and the business goals. Once you have understood the needs, you will need to design the software. Things to do in this stage include drawing data flow diagrams, designing the database, and so on. The next stage is the implementation stage. We can see this as the actual coding process.

Next comes testing, which is the main focus of this book. In this chapter, we will learn how to plan our test based on various testing concepts. After the testing stage, we will deploy the project, and finally we maintain the project. Because this is a cycle, we theoretically move back to the analysis stage during or after the maintenance stage. This is because a software or program is evolutionary; as needs and requirements change, so does our software.

Although the terminologies and number of stages may be slightly different from what you see in other related content, the process is generally the same. The main takeaway here is that testing typically comes after implementation.

The agile method

You may have heard about the agile methodology, which includes the agile software development methodologies, and of course, agile testing methods.

In general, agile software development and testing methods typically happen with the end users or customers in mind. There is often little documentation, and a focus on short software development cycles, which typically last for one to four weeks.

So how does this relate to the software development cycle that you have read about in the previous section? In general, testing is not an individual phase by itself, but rather is closely integrated with the development process, with code being tested from the customer perspective, as early as possible, when code becomes stable enough to perform testing.

The agile method and the software cycle in action

It might be difficult for you to visualize how the previous theories come into place. The process of creating the sample code for this book closely mimics the software lifecycle and agile methodology. So I thought I'll very briefly share with you my experience when I was creating the code samples for this book, based on the theories that we have learnt about.

Analysis and design

Technically speaking, the analysis and design stage took place when I was thinking about what kind of code samples would meet the objectives of the book. I thought that the code should be simple enough to follow, and most importantly should demonstrate the various features of JavaScript. The code should set up the stage for code testing in the later chapters.

Implementation and testing

The implementation stage occurred when I was writing the code samples. As I created functions for snippets of code, I tested whenever I could, and asked myself if the code could demonstrate the use of JavaScript and facilitate testing purposes later on.

So, what happened here is that I used some form of agile testing as I tested as often as I could.

Deployment

Deployment of the code in the business world typically occurs after the code has been transferred to the end user. However, in my case, deployment involved sending my code samples to the editors.

Maintenance

The maintenance stage occurred when I fixed bugs discovered by the editors after the code was submitted. Despite the best of intentions, code is not always error-free.

Do you need a test plan to be able to test?

You will most likely require a test plan in order to carry out testing. This is because a plan helps you keep a clear objective on what to test. It also helps you to figure out what kind of tests you want to perform on your program.

Most importantly, as you will realize, in order to carry out a thorough test you will need to implement various tests, including testing concepts based on white box testing and black box testing, web page testing, unit testing, integration testing, and so on. A test plan also serves as a record of your test data, bugs, test results, and possible solutions for your bugs. This means that in order to ensure that you do not miss anything, it is good to have a solid plan as to what to test, when to test, and how to test your program.

When to develop the test plan

In theory, if you look at the software development cycle, you will see that testing comes after implementation. Development of the test plan should take place after you have completed implementation (the actual coding process) of the program. This is because it is only at this point that you have confirmed what features, methods, and modules you have; planning what to test based on what you have already done makes good business sense, because you know what to focus on.

However, in practice, it is advisable to start planning before the implementation process. Depending on your situation, it is certainly possible that you can develop a High Level Test Plan (HLTP) or High Level Test Case (HLTC). An HLTP is required if you are developing a large and complex system, and is meant to address the overall requirements. Other supporting test plans are used to address the details of the system. An HLTC is somewhat similar to an HLTP, except that it covers test cases of the main functionalities that are directly related to the overall requirements of the system.

Another point that you should take note of is that, in practice, the test plan can be broadly categorized into system test and user acceptance test. System test covers all forms of functional testing and non-functional testing (which you learn about later), whereas user acceptance testing is a phase where testing is carried out by end users prior to transferring ownership to them.

How much testing is required?

You might be anxious to determine what you need to test and what you do not. Although there are many different arguments as to how much testing is required, I personally believe the aspects of your program listed in the following sections should define the scope of your test plan.

What is the code intended to do?

Firstly, you need to understand what the code is intended to do. For instance, the business requirements for our code in the previous chapters is to calculate whether the user can retire on time, based on his inputs, such as his current age, the age at which he wants to retire, his current spending, current salary, and so on. Therefore, we created code that meets the business needs. Once we know what our code is intended to do, we can test whether the code satisfies our business needs.

Testing whether the code satisfies our needs

By testing the code to see if it satisfies our business needs, we mean that for each input, we need to get the correct output. Going back to our example in *Chapter 2*, *Ad hoc Testing and Debugging in JavaScript* and *Chapter 3*, *Syntax Validation*, I would need to ensure that if the total left-over income is less than the amount of money that is needed for retirement, the output would be "unable to retire", at least in a pseudo sense. What we need to do from a testing point of view is to make sure that whenever the mentioned condition is true, the output would be "unable to retire".

This can be achieved through a concept called white box testing, where testing is carried based on the assumption that the tester knows what the code is about. I'll cover the specific details of white box testing and other testing concepts in the following chapters. To give you a heads up, some of the testing concepts that you will encounter will include unit testing, where you test codes in small units, and boundary values testing, where you test for the maximum or minimum acceptable values of your code.

The next thing that we will need to consider is how to test for or detect invalid actions by users.

Testing for invalid actions by users

"Never trust users" is a phrase which we most commonly hear when developing for the Web. This is because there may be malicious users who attempt to "break" your applications by giving invalid input. Using the example from previous chapters, the input fields for the name can only accept characters and spaces, and the input fields for the age and salary can only accept numbers, and not characters. However, if someone were to attempt to enter characters into the age or salary field, this would be an invalid action.

Our program will have to be robust enough to test or check for invalid actions; incorrect input will result in incorrect output.

A short summary of the above issues

By knowing what your code is intended for and what it is supposed to do, and understanding the need to detect invalid actions by users, you have already defined the scope of your test plan. Your tests should revolve around these criteria.

We can now move on to the various testing concepts that you will be using for various aspects of your test, and the building blocks of a test plan—major testing concepts and strategies.

Major testing concepts and strategies

In this section, we will cover different types of testing concepts and strategies. I will not attempt to go into too much detail with regards to each concept, but rather I need you to get the gist of it and see where each of these concepts is coming from. After you have gained familiarity with these concepts, we will move on to creating the actual test plan. As a start, I will begin with the business strategies that developers follow (whether you are performing a project for an external or an internal client), so that you can gain a high-level idea of how testing is conducted. In general, no matter what testing concepts, methodology, or ideology you subscribe to, you will face the following test cases:

- Functional requirement testing
- Non-functional requirement testing
- Acceptance testing

Functional requirement testing

Functional requirement testing is meant to test the code, a function, or a module of a software system. For instance, going back to the code that we wrote for the previous chapters, the functional requirements consists of the following:

1. Check user's input for validity.
2. If the input from step 1 is valid, a new input box will appear on the right-hand side of the current input box, after the users mouse moves on to the next input box.
3. Provide the correct calculation output based on the users input. For example, if the user requires 1,000,000 dollars for retirement, and he only has 500,000 dollars by the time he retires, then he will not be able to retire.

Examples of functional requirement testing that are covered in this chapter are as follows:

- Web page tests
- Boundary testing
- Equivalence partitioning

Non-functional requirement testing

Non-functional requirement testing refers to testing requirements that are not related to the functionality or specific behaviour of the software. Rather, it is a requirement that specifies criteria that can be used to judge the operation of a software.

For example, a functional requirement would be that our software should be able to store the values that our users have entered, and a non functional requirement is that the database should be updated in real-time.

Another example that is related to our sample code in previous chapters is that a functional requirement would be a software, which is able to calculate whether our user is able to retire on time, and a non-functional requirement would be one in which our user interface should be intuitive. Do you see the difference between non functional requirements and functional requirements, now?

Examples of non functional requirement testing that are covered in this chapter are as follows:

 ◆ Performance testing
 ◆ Usability testing
 ◆ Integration testing

Other non-functional requirements that you are likely to encounter throughout your career as a software developer are as follows:

 ◆ Fast loading of pages
 ◆ Search engine optimized web pages
 ◆ Documentation of the software that you have created
 ◆ Efficiency of the system
 ◆ Reliability of the software
 ◆ Interoperability of the software code that you have produced. For instance, you can code JavaScript across major browsers

Acceptance testing

Acceptance testing is usually the final phase of the entire testing process. This is often done prior to the final acceptance of the software by the customer. Acceptance testing can be further divided into two parts. The software vendor performs the acceptance testing first, and then acceptance testing by the end users (known as user acceptance testing) is performed.

Acceptance testing is the time where your customer (or the end-user) will perform actual testing (similar to actual usage of the system) on the software that you have created. A typical process will include the creation of test cases by the end users that reflect business use of the software.

If you are using agile testing methods, such test cases are often referred to as stories. It depends on how the customer will use them in a business setting. And after the user acceptance tests, you will transfer ownership of the product to your customers.

With the most common testing scenarios covered, we will move on to the specifics of the testing concepts. We will start with one of the most commonly-heard testing concepts, the black box testing concept.

Black box testing

Black box testing belongs to the "box approach", where a piece of software is regarded as a box and the box contains various functions, methods, classes, and so on. Metaphorically, a "black box" typically means that we cannot see what is inside the box. This means that we implement the test without knowing the internal structure of our program; we take an external perspective of the program, using valid and invalid inputs in order to determine if the output is correct.

Because we have no knowledge about the internal structure and code of the program, we can only test the program from a user's point of view. In this case, we might try to determine what the major functions are, and then attempt to implement our test based on these functions.

The main advantage of black box testing is that the test results are often unaffiliated, because the tester has no knowledge of the code. However, the disadvantage is that because the tester has no idea of what the code is about, the tester may create tests or perform tests that may be repetitive, or tests that fail to test the most important aspects of the software. Or worse, the tester may miss out an entire function or method.

That is why, in the real world, test cases are prepared in the early phases of the development cycle, so that we will not miss out on certain requirements. The advantage is that testers will have access to the required test cases, but at the same time, the testers need not have full knowledge of the code.

Some examples of black box testing include usability testing, boundary testing, and beta testing.

Usability tests

In simple terms, usability testing typically involves testing from the user's point of view, to see if the program we have created is easy to use. The key objective here is to observe users using our program, to discover errors or areas of improvement. Usability testing generally includes the following aspects:

- **Performance**: especially in terms of the number of clicks (or actions) that a user has to take in orders to complete a particular task, such as signing up as a member, or purchasing a product form a website, and so on.
- **Recall**: can users remember how to use the program after not using it for a certain period?
- **Accuracy**: does our program design result in mistakes by the end users?
- **Feedback**: feedback is certainly one of the most important AJAX-related application issues. For instance, after submitting an AJAX form, a user will typically wait for some form of feedback, (in the form of visual feedback, such as a success message). But imagine this—if there is no form of visual feedback or success message, how will the user know if he has submitted the form successfully or unsuccessfully?

Boundary testing

Boundary testing is a form of testing method where the maximum and minimum values are tested. Boundary testing sometimes includes the testing of error values and typical values.

For instance, in the program in the previous chapters, the maximum number of characters we allow for the entry of names are 20 characters.

Equivalence partitioning

Equivalence partition testing is a technique that divides a range of data into partitions from which test cases can be derived. For instance, for input boxes accepting a users' age, it should exhibit the following partition:
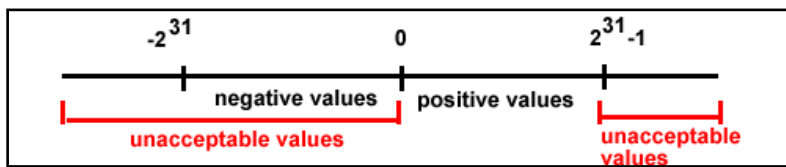
Note that only positive values are accepted for our example to read in a users' age, as a person's age should technically be positive. Therefore, any negative values are unacceptable values.

For the range that is less than -2^{31} and larger than $2^{31}-1$, it is asssumed that the integers can only hold values between -2^{31} and $2^{31}-1$ due to hardware and EMCA operator requirements.

Beta testing

Beta testing has been popularized by the current popular Web 2.0 companies, such as Google, where web applications are often released to a limited audience other than the core programing team. Beta testing occurs after alpha testing, where most of the bugs and faults have been detected and fixed. Beta testing is often used as a way to gain feedback from prospective users.

Such a process is commonly seen in open source projects, such as Ubuntu (an open source operating system based on Linux), jQuery (a JavaScript library), and Django (a Python-based web framework). Such open source projects or software typically have a series of alpha and beta releases. They also typically have release candidates prior to releasing a major version of the software or project.

White box testing

White box testing is also known as clear box testing, glass box testing, or transparent testing. White box testing can be seen as the opposite of black box testing; we test the program with knowledge of the internal structure of our program. We take an internal perspective of the program, and use this perspective when we implement our test plan.

White box testing typically occurs when the test has access to the internal code and data structures of the program. Because we take an internal perspective of our program and with knowledge of our source code, we design the test plan based on our code.

We might find ourselves tracing the path of how our code is executed and work out what are the input and output values for various functions or methods of our program.

Some examples of white box testing include Branch testing, and Pareto testing.

Branch testing

Branch testing is a concept where each branch of the code should be tested at least once. This means that all functions or code that has been written should be tested. In software testing, there is a measure known as code coverage, which refers to how much source code of a program has been tested. Some of the more important types of branch testing coverage includes the following:

- ◆ Functional coverage: where we make sure that each function of the code has been called (tested)

- ◆ Decision coverage: where each of the `if else` statements has been tested. There might be cases where the `if` part of the code works but not the `else` part of the code, and vice versa.

Pareto testing

Pareto testing is what I personally call "real world" testing, and is conducted under strict time and money constraints. This is because Pareto testing only focuses on the most used functions; the most frequently used functions are what matter the most and hence we should focus our time and effort on testing these functions. Alternatively, we may see Pareto testing such that most bugs come from a small handful of functions of our programs; therefore, by spotting these functions, we can test our program much more effectively.

Pareto testing is derived from an idea called "Pareto Principle" or perhaps better known as the "80-20 principle". What the Pareto Principle suggests is that roughly 80% of the effects come from 20% of the causes. For instance, 80% of the sales revenue may come from 20% of the sales team or customers. Or another example would be 80% of the world wealth is control by 20% of the world's population. Applied in our case here, we can say that 80% of the bugs or program errors come from 20% of our code, and therefore we should focus testing on that particular 20% of the code. Alternatively, we can say that 80% of the program's usage activity comes from 20% of our code. Similarly, we can focus testing on that particular 20% of the code. Just for the record, pareto testing can be regarded as a general testing principle, and not just a form of white box testing.

Unit tests

Unit testing breaks up code into logical chunks for testing, and generally focuses on one method at a time. A unit can be seen as the smallest possible chunk of code, such as a function or method. This means that in the ideal situation, each unit should be independent from all other units.

When we are performing unit testing, we attempt to test each function or method as we complete it, thus making sure that whatever code we have works before we move on to the next function or method.

This helps to reduce errors, and you may have noticed that we have somehow applied the idea of unit testing when developing the JavaScript program in the previous chapters. As we create each function, we try to test it whenever possible.

Some of the benefits of unit testing includes minimization of errors, and allowing ease of change, because each function or method is tested individually in isolation and, to a good extent, simplifies integration.

The main benefit, in my opinion, is that unit tests are flexible and allow ease of documentation. This is because as we write and test new functions, we can easily take note of what the problems are, and whether the code can work correctly. In effect, we are doing incremental documentation—documenting the results as we test.

Unit testing is also an integral part of integrated testing, especially in the bottom-up approach, as we test our program from the smallest possible unit before moving on to larger units. For example, as I was creating the code for *Chapter 2*, *Ad Hoc Testing and Debugging in Javascript*, I essentially carried out unit testing informally. I carried out unit testing by treating each of the functions as individual units, and tested each JavaScript function with the related HTML input field, in order to make sure that the correct output was achieved. This technique can be seen as part of performing continuous integration as new code is being written.

Continuous integration is a process where developers integrate their code frequently, in order to prevent integration errors. This is often done with the help of automated builds of the code (and includes tests) to detect integration testing. As we create new code, it is important that we integrate with the existing code to make sure that no compatibility issues or new bugs (or even old bugs) are introduced. Continuous integration is becoming popular as it integrates unit tests, revision control, and build systems.

Web page tests

As mentioned previously, web page testing is a form of functional testing, and typically refers to the testing of the user interface, from the user's point of view. For our purposes here, we would test our JavaScript program in conjunction with HTML and CSS.

Web page testing also includes testing for correctness in terms of different browsers and platforms. We should at least focus on the major web browsers such as Internet Explorer and Firefox, and see if the presentation and JavaScript program works under different browsers.

To have a brief idea regarding the usage of browsers, you might want to head down to `http://www.w3schools.com/browsers/browsers_stats.asp` to see which browsers are popular, in decline, or on the rise.

> It appears that Google Chrome is gaining a lot of momentum, and it has a good chance of becoming a popular web browser; in less than two years, Google Chrome has increased its market share from 3.15 percent to 14.5 percent, based on the statistics provided by w3schools. This increase in popularity is in part due to its JavaScript engine performance.

The other main focus of web page tests also includes checking for the most frequently-used user behaviors, such as illegal and legal values, login, logout, erroneous behavior of the users, SQL, HTML injection, checking of HTML links, images, the possibility of robot attacks, and so on.

As SQL, HTML injection, and robot attacks are out of the scope of this book, we will focus on the other issues, such as making sure that the web page will work under different browsers, testing for illegal and legal values, erroneous behavior, and frequent behaviors.

Performance tests

Performance tests have a wide range of genres such as load testing, stress testing, endurance testing, isolation testing, spike testing, and so on. I will not attempt to bog you down with the details. Instead, I will focus on two of the more common issues that you will face as a JavaScript programer.

Firstly, performance can refer to the amount of time required for the client to download a piece of JavaScript. You may argue that download time depends on the Internet connection. But there is one simple thing that you can do to reduce the size of your JavaScript without refactoring or rewriting it, and that is compressing your JavaScript code. A good example of this would be the JQuery library, which we introduced in *Chapter 3*, *Syntax Validation*. If you visit the JQuery home page at `http://jquery.com`, you may have noticed that JQuery comes in two forms—a production version and a development version. The production version is minified, and the file size is 24KB, whereas the development version is 155KB. Obviously, the production version is smaller in file size and hence improves performance in terms of downloading the JavaScript.
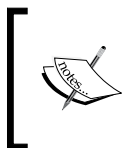
> Compressing your code—or minifying your code—refers to an act where you remove all unnecessary white spaces and lines from your code in order to reduce file size. Some code minifiers automatically remove comments, replace functions, variables, and even encode in different codings.

Secondly, performance can also refer to the speed at which a particular piece of code executes for any given amount of input. In general, we need to use external libraries or tools to help us find out which parts of our code are performing relatively slower than the others, or where the bottlenecks are. Related tools, and how we can apply performance testing, will be covered in *Chapter 6*, *Testing more complex code*.

Integration testing

Integrated testing is among the last steps of the testing process prior to acceptance testing. Because we have made sure that the basic building blocks of the program work correctly as an individual unit, we will now need to make sure if they can work together.

Integration testing refers to the testing of all of the different components of our program. The different components can refer to the various units that we have talked about so far. The main objective of integration testing is to ensure that the functional, performance, and reliability requirements are met. We also test the different units together and see if they can work; we'll need to check for any irregularities when combining the units together.

Integration testing can take different forms, such as top-down and bottom-up approach.

In the top-down approach, we start with the highest-level integrated module, followed by the sub-modules or functions of each module. On the other hand, bottom-up testing starts from the lowest level components before moving on to the upper-level components.

Based on the sample code that we have seen so far, it would be difficult to understand how integrated testing works. In general, if we view the HTML code as a unit, CSS as a unit, and each individual JavaScript function as a unit, we can see that integrated testing would include testing all three together and making sure that it is correct.

In the bottom-up approach, we begin testing from the basic units of code. As we test the basic units of code, we move up to test larger units of code. This process is similar to unit testing.

Regression testing—repeating prior testing after making changes

Regression testing focuses on uncovering errors in a program when a program is being modified or upgraded. In real-life situations, we tend to make changes to a program— whether this is upgrading it, adding new features, and so on. The key point is that as we make changes to a program, we need to test the new components to see if they work in conjunction with the old components.

We need to perform regression testing because research and experience have shown that as a program is being modified, new or old errors may appear. For instance, an old, previously-fixed bug may be re-introduced into the program when a new feature is being added, or the new feature itself may contain a bug that affects the existing features. This is where regression testing comes in: we perform previous tests to make sure that the old components are still running and that no old faults have re-emerged. We test the new features with the old components to ensure that the entire system is working. Sometimes, in order to save time and resources, we may only perform testing on the new features in conjunction with the old components. **At this point, we can apply impact analysis to** determine the impact area of the application, by adding or modifying code.

Regression testing is as real as it gets. This is because as a program grows, the chances are that you will make changes to your code. As you make changes to your code **there is a** likelihood **that bugs or incompatibilities may be introduced to your program, and regression** testing helps you to spot such mistakes.

Testing order

We have now covered the required background knowledge, so it is time to understand what kind of tests we should start with. The order in which we carry out the tests depends on whether we want to implement bottom-up testing or top-down testing. There is nothing wrong with either order of testing, but I personally prefer bottom-up testing: I'll typically start with unit testing first, followed by other types of tests (depending on what the program is like), and finish off with integration testing.

The main reason for taking this approach is that unit testing allows us to find errors in the code much earlier; this prevents bugs or errors from piling up. In addition, it provides flexibility in how you choose to document the test results.

However, if you prefer the top-down approach, you can always start by testing the program as if you were an end user.

In the real world, especially in terms of testing web applications, it can be difficult to differentiate (at least conceptually) between bottom-up testing and top-down testing. This is because although the user-interface and programing logic are separated, we really need to test both at the same time in order to understand if it works the way that we want it to.

Nonetheless, the testing order should finish with user acceptance testing, because the end users are the ones who will be using our code eventually.

In the next section, we will show you how to document your test plan. You will notice that we will be performing tests from the users' point of view. Now, it is time to document our test plan.

Documenting your test plan

Now that we have covered the required testing concepts, it is time to learn how we can create the test plan. At the same time, we will document our test plan; this will serve as a basis for the next part of this chapter, where we will apply the test.

The test plan

Our test plan will consist of some of the concepts we have covered earlier, such as web page testing, boundary testing, integration testing, and others. Because we are applying the test on the code we have used in *Chapter 2, Ad Hoc Testing and Debugging in Javascript*, we have the advantage of knowing what the code is about. Therefore, we can design our test process in such a way that it can incorporate ideas from both black box testing and white box testing.

You might want to go to the `source code` folder and open the `sample_test_plan.doc` file, which is our sample test plan. This is a very simple and informal test plan, which contains only the bare minimum of the required components. If you are writing documentation for your own reference, you can save on time and effort by using a simple document. However, if you are preparing a test plan for a client, you will need a more elaborate document. For simplicity sake, we'll use the sample document provided in the `source code` folder to help you understand the planning process quickly. I will briefly run through the components of our test plan and at the same time, I will introduce to you the main components of our test plan.

Versioning

In the first component, you will notice that there is a version table, which documents the changes in the test plan. In the real world, plans change and therefore, it is a good habit to keep track of the things that have changed.

> Another way to keep versioning easy and maintainable is to use version control software such as Git or BitBucket. Such versioning tools keep a log of the changes that you have made in your code; this will enable to trace what changes you have made, and this makes creating tests plans a lot easier. You can visit `http://git-scm.com/` to learn more about Git, and `http://bitbucket.org/` to learn more about BitBucket.

Test strategy

The next important component that you should notice is the test strategy. The test strategy represents the main thoughts and ideas that we will be using for our test plan. You will see that we are employing both white box and black box testing, along with unit testing and integration testing. Because our JavaScript program is web-based, we are implicitly carrying out a form of web page testing, although this is not mentioned in the subsequent parts of the chapter. For each phase of the test, we will decide on the test values required. Also, if you look at the `sample_test_plan.doc`, you will see that I have added, in the form of a brief description of the expected values, the result or response for each part of the test.

Testing expected and acceptable values by using white box testing

The first thing that we will be doing is white box testing by using unit testing. Because we have a strong **understanding of the code and user interface (the HTML and CSS code), we** will apply the test at the user-interface level. This means that we will test the program by entering the various test values that we have decided upon.

In this case, we will use the program as we have already used in *Chapter 2*, *Ad Hoc Testing and Debugging in Javascript*, and *Chapter3*, *Syntax Validation*, and see if the program works the way that we intended it to. We will be using values that are expected and acceptable here.

The input will be what the program requires us to enter—for input fields that require us to enter down our name, place of birth, and so on, we will enter characters into it. Input fields that require numbers as inputs, such as age, the age at which we would like to retire, salary, expenses, and so on, we will enter numbers.

The details of the input are as follows (the input values are for demonstration purposes only):

Input fields	Input value (case 1)	Input Value (case 2)
Name	Johnny Boy	Billy Boy
Place of birth	San Francisco	San Francisco
Age	25	25
Spending per month	1000	1000
Salary per month	100000	2000
Age at which you wish to retire	55	55
Amount of money I want by retirement age	1000000	1000000

For each of the input values, we would expect a corresponding input field to be created dynamically in the middle of the screen, under the header **Response**, and at the same time, the original input field would be disabled. This is known as the expected output, result, or response for the test. This goes on for the rest of the input fields for the first form. An example of the dynamically-created field is shown in the following screenshot:

Notice that in the middle of the screenshot, under the header **Response**, there are two input fields. These input fields are created dynamically.

Testing expected and unacceptable values by using black box testing

The second thing that we will be doing is to perform black box testing by employing boundary value testing. There are two parts to this test: we will first test the boundary values of the program to see if the output is correct. The inputs are similar to what we have used for white box testing, except that we will use unusually large numbers, or unusually large number of characters, for each input. We will also use single number and single characters as part of our inputs. The output for each of the inputs should be similar to what we have seen in white box testing.

To be more specific, we will be using the following test values (note that the test values are purely for demonstration purposes only; when you are creating your program you have to decide what suitable boundary values should be used):

Input fields	Minimum Value	Common Value	Maximum value	Comments
Name	A single character, such as 'a'	Eugene	An extremely long string, not more than 255 characters.	Range of values (X): Single character 1 <= X <= 255 characters
Place of birth	A single character, such as a	New York City	An extremely long string, not more than 255 characters.	Range of values (X): Single character 1 <= X <= 255 characters
Age	1	25	No more than 200 years old	Range of values (X): 1 <= X <= 200
Spending per month	1	2000	1000000000	Range of values (X): 1 <= X <= 1000000000
Salary per month	2	5000	1000000000	Notice that that we are assuming that our user earns more than he spends. Range of values (X): 1 <= X <= 1000000000
Age at which you wish to retire	This age should be greater than the present age	This age should be greater than the present age	This age should be greater than the present age	Range of values (X): 1 <= X <= 200
Amount of money I want by retirement age	We will be using 1 here	A suitable number, such as 1000000	No more than a trillion dollars	Range of values (X): 1 <= X <= 1000000000

If you refer to the `sample test` document, you will realize that I have provided a sample range of values for each of the input fields.

Remember that we've touched on equivalence partitioning in the earlier sections? In practice, given a boundary value, we would test three values relating to the given test value. For example, if we want to test a boundary value of '50', then we will test on 49, 50, and 51. However for simplicity's sake, we will be testing on the intended value only. This is because in the next chapter we will be carrying out the actual test for the given values; it can get repetitive and tedious. I just want you to know what the real world practices are.

The second part of this test is that we will test for expected illegal values. In the first scenario, we will be using values that are both accepted and unaccepted. The input will be similar to what we have used for the white box testing phase, except that we will use characters as inputs for input fields that require numbers, and vice versa. The expected output each time that we enter an unaccepted value is that there will be an alert box telling us that we have entered a wrong value.

For details, check the following table:

Input fields	Input Value	Input Value Case 1	Input Value Case 2	Input Value Case 3
Name	Digits or empty values	1	~!@#$%^&*()"	Testing
Place of birth	Digits or empty values	1	~!@#$%^&*()"	testing
Age	Characters and empty values	a	~!@#$%^&*()"	-1
Spending per month	Characters and empty values	a	~!@#$%^&*()"	-1
Salary per month	Characters and empty values	a	~!@#$%^&*()"	-1
Age at which you wish to retire at	Characters and empty values	a	~!@#$%^&*()"	-1
Amount of money I want by retirement age	Characters and empty values	a	~!@#$%^&*()"	-1

In general, for each of the expected illegal values, we should expect our program to alert us with an alert box, telling us that we have entered the wrong type of values.

In the second test scenario, we will attempt to enter non-alphanumeric values, such as exclamation marks, asterisk signs, and so on.

In the third test scenario, we will test for negative values for input fields that require numbers. The input values for the third test scenario are as follows:We are using -1 to save some typing; so negative values such as -100000 don't make any difference.

Testing the program logic

For this part of the test plan, we will attempt to test the program logic. Part of ensuring program logic is to ensure that the inputs are what we need and want. However, certain aspects of the program logic cannot be guaranteed simply by validating the input values alone.

For instance, an implicit assumption that we have about the user is that we assume the user will enter a retirement age that is bigger than his present age. While this assumption is logically sound, users may or may not enter the value according to conventional assumptions. Therefore, we need to guarantee the logic of the program is correct by ensuring that the retirement age is greater than the present age.

The inputs for this test are as follows:

Input fields	Input value of first form
Name	Johnny Boy
Place of birth	San Francisco
Age	<u>30</u>
Spending per month	1000
Salary per month	2000
Age at which you wish to retire	<u>25</u>
Amount of money I want by retirement age	1000000

The key thing to note here is that the value for "age at which you wish to retire " is smaller than "age".

We should expect our program to spot this logical error; if it does not, we will need to fix our program.

Integrated testing and testing unexpected values

The final phase is integrated testing, where we test the entire program and see if it works together, which includes the first form, the second form which is derived from the first form, and so on.

In the first test scenario, we begin slow and steady by testing expected and acceptable values. The input values for the first test scenario are as follows (**the input values are for demonstration purposes only**):

Input fields	Input Value (case 1)	Input Value (case 2)	Input Value (case 3)	Input Value (case 4)
Name	Johnny Boy	Johnny Boy	Johnny Boy	Johnny boy
Place of birth	San Francisco	San Francisco	San Francisco	San Francisco
Age	25	25	25	25
Spending per month	<u>1000</u>	<u>1000</u>	1000	1000
Salary per month	<u>100000</u>	<u>2000</u>	2000	<u>100000</u>
Age at which you wish to retire	55	55	<u>28</u>	<u>28</u>
Amount of money I want by retirement age	2000000	2000000	1000000	100000

Take note of the input values that are underlined. These input values are designed to determine if we will get the correct response based on the input. For example, after entering all of the values and submitting the dynamically-generated second form, the input values for case 1 and case 3 will result in an output stating that the user will not be able to retire on time, whereas the input values for case 2 and 4 will result in an output stating that the user will retire on time.

Here's a screenshot that shows what the output looks like if the user can retire on time:

Final response:

Hi **Eugene**,

We've processed your information and are pleased to announce that you will be able to retire on time.

Base on your current spending habits, you will be able to retire by **30**years old.

Also, you'll have' **49940000** amount of excess cash when you retire.

Congrats!

The next screenshot shows the output if the user cannot retire on time:

Final response:

Hi **Eugene**,

We've processed your information and we have noticed a problem.

Base on your current spending habits, you will not be able to retire by **30** years old.

You need to make another **999970000** dollars before you retire inorder to acheive our goal

You either have to increase your income or decrease your spending.
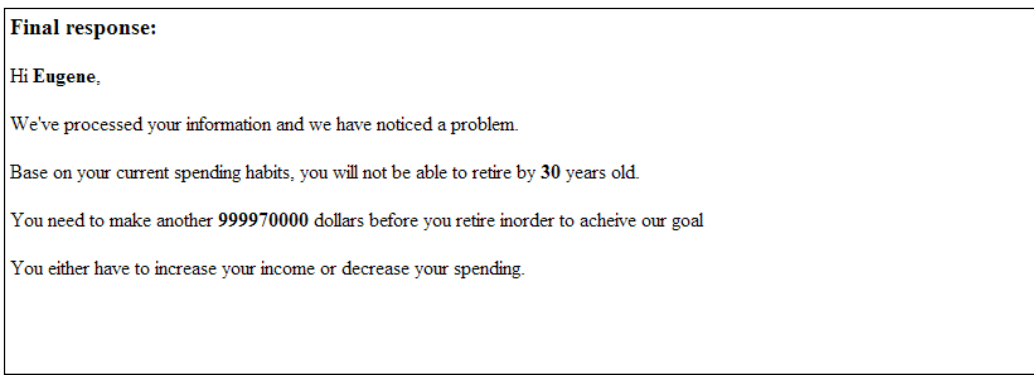
Take note of the differences in text for the two **different cases.**

For **the full details of the results of the test case, open the** `sample_test_plan.doc` file, which can be found in **the** `source code` folder of this chapter.

Now it's time for the second test scenario. **In the second test scenario, we first finish filling** up the values in the first form. Before we submit the second form, **which was created dynamically, we will attempt to change the values. The input values will include** the values that we **have used for both white box testing and black box testing. The input values for the** first test scenario are as follows:

Input fields	Input value of first form	Input Value the second form (random values)
Name	Johnny Boy	25
Place of birth	San Francisco	100
Age	25	Johnny Boy
Spending per month	1000	Some characters
Salary per month	100000	More characters
Age at which you wish to retire at	20	Even more characters
Amount of money I want by retirement age	1000000	1000000

The main objective of this phase of the test is to test the robustness of the second form, which we have not verified up to this point of time. If the second form fails, we will need to change our code to enhance the robustness of our program.

We'll now move on to the next component of our test plan—errors or bugs found.

Bug form

The last component helps us to record the bugs that we have found. This area allows us to take note of what the errors are, what caused them, and the function or feature in which these errors occurred. In general, whenever we spot an error, we need to take note of the exact function that resulted in the error, and comment on what the possible solutions may be.

Summary of our test plan

The components introduced above are some of the most important components of a test plan. In general, for each phase of the test, we have stated our test data and our expected output. Note that we are using this documentation as an informal way of reminding ourselves of what tests needs to be done, the required inputs, expected outputs, and more importantly the bugs that we have found. One thing not mentioned in this sample documentation is the action to be performed for those bugs that are discovered; this will be covered in the next chapter.

Summary

We effectively carried out the planning process for our test plan. Although our test plan is informal, we have seen how we can apply various testing concepts, coupled with different test data values to test our program that we have created in previous chapters.

Specifically, we covered the following topics:

- We first started off with a brief introduction to the key aspects of software engineering. We've learned that testing takes place after the implementation (coding) stage.

- We've learned to define the scope of our test by asking what our code is suppose to do, making sure that it does what it is supposed to do, and finally testing for invalid actions by users.

- Next we covered various testing concepts such as white box testing, black box testing, unit testing, web page testing, performance testing, integrated testing, and regression testing.

- We also learnt that we need to test our program from different aspects, thus enhancing the robustness of the program.

- Although the testing concepts introduced in this chapter may be different in certain aspects, we can group them as: testing expected but acceptable values, expected but unacceptable values, and unexpected values in general. We've also learnt to test for logical errors based on our understanding of the code that we have written.

◆ Finally we planned and documented our test plan, which includes the test process description, test values, expected output and other important components, such as versioning and a bug form.

Although testing methodologies can be substantially different depending on organization types and types of applications, the methods that are listed here are generally more suitable for lightweight web applications. However, the concepts also form the building blocks of large-scale, complex web applications.

This chapter marks the end of planning for your test. Now brace yourself as we move on to the next chapter, where we will carry out the test plan.

5

Putting the Test Plan Into Action

Welcome to the fifth chapter. This chapter is pretty straightforward; we basically put the plan discussed in Chapter 4, Planning to Test, into action.

Heres how we are going to implement our test plan. Well first start by testing the expected and acceptable values, and follow this by testing the expected but unacceptable values. Next, well test the logic of our program. Lastly, we'll perform integrated testing and testing of unexpected values or actions.

Apart from performing the above tests, here's what we will also cover in this chapter:

- ◆ Regression testing in action—youll learn how to perform regression testing by fixing bugs and then testing your program again
- ◆ The differences between client-side testing and server-side testing
- ◆ How using Ajax may make a difference to testing
- ◆ What to do when a test returns a wrong result
- ◆ What happens if your visitor turns off JavaScript
- ◆ How to enhance performance by compressing your JavaScript code

So let us get our hands dirty, and start testing right away.

Applying the test plan: running your tests in order

In this section, we'll simply apply the test plan to our program. For simplicity's sake, we will record any bugs or errors in the Bug Report Form found in the sample test plan from the previous section. In addition to that, at the end of each test, we will record a Pass or Fail text in the `sample_text_plan.doc`, which we created in the previous chapter. However, take note that in the real world (especially if you are working on a custom project for your client), it is extremely important that you document the results, even if your tests are correct. This is because, very often, producing the correct test results is part and parcel of handing over the code to your client.

Just a reminder—the test plan that we are going to use was created in the previous chapter. You can find the test plan in the `source code` folder of *Chapter 4*, entitled `sample_test_plan.doc`. If you are in a hurry and would like to see the entire completed test plan where all tests have been carried out already, head to the `source code` folder of *Chapter 5* and open up `sample-testplan-bug-form-filled.doc`.

In case you do not wish to flip the pages or open up your computer just to see the list of the tests, the list of tests are as follows:

- ◆ Test Case 1
    - ❑ Test Case 1a: White Box Testing
    - ❑ Test Case 1b: Black Box Testing
        - ❑ Test Case 1bi: Boundary Value Testing
        - ❑ Test Case 1bii: Testing for illegal values
- ◆ Test Case 2: Testing Program's logic
- ◆ Test Case 3: Integration Testing
    - ❑ Test Case 3a: Testing the entire program with expected values
    - ❑ Test Case 3b: Testing the robustness of the second form.

With this in mind, let us proceed to the first test.

Test Case 1: Testing expected and acceptable values

Testing expected and acceptable values refers to the white box test phase. We will now execute the test as per our plan (this is First test scenario).

Time for action – Test Case 1a: testing expected and acceptable values by using white box testing

In this section, we will start our test by using values that we have predetermined during the planning phase. The source code that you are using for this part of the chapter is `perfect-code-for-jslint.html`, which can be found in the `source code` folder of *Chapter 3*. What we will do here is enter the expected and acceptable values. We will start testing by using the input values for input value case 1 as per our sample test document.

1. Open the source code in your favorite web browser.

2. When you open your program in your web browser, the focus should be on the first input field. Enter the name **Johnny Boy** as per our plan. After you have entered **Johnny Boy** in the first input field, go on to the next field.

   As you change your focus to the next field, you will see a new input field appearing on the right-hand side of the original input field that contains the value you have entered. If this happens, then you have received a correct and expected output for the first input. If you do not understand what this means, feel free to refer back to *Chapter 4*, *Planning to Test*, and look at the screenshot given for the expected output.

3. For the second input, we are required to enter a place of birth. Enter **San Francisco**, as per the plan. Click on (or tab to) the next field.

   Similarly to the first input field, after you move to the next field you will see a new input field containing your input value. This means that you have the correct output at this point.

4. This step is similar to the above step, except that the input value is now a number. Enter your age as **25**. Then move on to the next field. You should also see a new input field on the right.

5. Now repeat the previous steps for the remaining fields for the form on the left. Repeat this action until you see a **Submit** button appearing in the middle of the screen.

   If a new input field is dynamically created for each of your input, and each of the new input fields created dynamically contains the exact same input that you have entered, then you have received the correct output. If not, the test has failed. However, based on our tests, we have received the correct output.

6. Now, refresh the page in your browser, and repeat the test for the input values found in input value Case 2. You should also receive the correct output.

Assuming that both test cases produce the correct output, then congratulations, there are no bugs or errors found in this phase of the test. There isn't anything special or tedious in this part of the test because we already knew that we would receive the expected output based on our input. Now, we will move to something more exciting—testing expected but unacceptable values.

Test Case 1b: Testing expected but unacceptable values using black box testing

In this section, you will continue to execute our test plan. As you continue with the tests, you will see that our program is not robust enough and has some inherent errors in it. You will learn that you will be required to take note of these; the information will be used later when we debug the program (this is second test scenario).

Time for action – Test case 1bi: testing expected but unacceptable values using boundary value testing

For this part of the test, we will continue to use the same source code as we have used in the previous section. We'll start by performing boundary values testing. Therefore, we will begin the test by using the "minimum values", followed by "maximum values". We will skip the common values test case as that was similar to what we did in the previous test.

1. Once again, refresh the page in your web browser.

2. We'll first enter a single character **a** for the input field of **name**. After you have entered the value, use your mouse to click on the next input field. You should see an input field dynamically created on the right-hand side of the first input field, as for the previous test.

The output for this test is similar to what you have seen and experienced in the previous test. What we are trying to test for is whether the program accepts a minimum value. For this phase of the test, we naïvely chose to accept a single character as an acceptable input. Because this is acceptable, we should see an input field that contains the value of **a** dynamically generated on the right-hand side of the original input field. If you see that, you have the correct output.

3. Similarly, we will enter a single character **a** for the input field for **place of birth**. After you have entered the value, use your mouse to click on the next input field. You will see an input field dynamically created on the right-hand side of the first input field, as seen in the previous test.

You should also receive the **correct output for this input value. Now let us move on** to the next input value.

4. We'll now **enter the number 1 as planned for the input field age. Similarly, after you** have entered the value, move the focus to the next input field.

5. We'll repeat the test by entering the values as planned.

In general, we **should not receive any errors at this point of the test. Similar to the** first test which we have performed earlier, we should see familiar output for each of **the inputs. However, I would like to point out an important point for this phase of** the test:

We have naïvely chosen a minimum value that might not be practical. Consider the various input fields that accept a single character value. To a large extent, our original program logic doesn't seem to suit the real world cases. In general, we should expect to have at least two or three characters for input fields that accept character values. Therefore, we will take this as a bug in our program and we'll take note of this on our "Bug Report Form". You may open the `sample-testplan-bug-form-filled.doc` document and see how we can take note of this flaw.

Now that we have cleared the **minimum values test case, it is time to move to the** next test case—**maximum values.**

6. As usual, refresh your web browser to clear all of the values that were previously entered. We'll now begin by entering an extremely long string, of more than 255 characters.

As **explained earlier, we should also receive a similar output—a** dynamically-generated input field that contains our input value.

7. Similarly, enter the values for the remaining input fields using long strings or large values. You should not face any errors.

While we do not have any obvious errors, you may have noticed **that we have a** similar problem **to the one we experienced earlier on. Our program does not have** a boundary value for maximum values as well. It appears that **if you try to enter** values that are larger than your maximum values, the program will still accept them, as long as the values are not illegal. Similarly, if you try to enter a string that is more than 200 characters, the program will still accept it because it is a legal value. This means that our program does not limit the maximum number of characters that a user can enter. **This can be regarded as a bug. We'll also take note of this** programing error in our Bug Report Form. You might want to pop over to have a look on how we recorded this error. **Now that we have completed the first phase of** our test for expected and unacceptable values, it is time to move on to the second phase of this test—testing for expected illegal values.

Time for action – Test case 1bii: testing expected but unacceptable values using illegal values

There are three input cases for this phase of the test. What we will do in the first case of the test is enter numeric values for input fields that require character inputs and vice versa.

Input Case 1:

1. We'll once again refresh our browser to clear out the old values. Next we'll begin to enter the expected illegal values. For the "name" input field, we'll enter a digit. This can be any number, such as "1". Go on and test it. After you have entered the digit, try to move your mouse cursor to the next input field.

As you attempt to shift the focus to the next input field, you should see an alert box telling you that you have entered an incorrect type of value. If you see the alert box as per our test plan, then there is no error at this point.

2. In order to test the next field, we will need to enter a correct value for the first field before we can move on to the next field. Alternatively, we can refresh the browser and go directly to the second field. Assuming that you are using the first method, let us enter a hypothetical name, **Steve Jobs**, and move on to the next input field. Similarly, well try to enter a digit for the **place of birth**. After you have entered a digit for the input field, try to move to the next field.

Once again, you will see an alert box telling you that you have entered an invalid input and that you need to enter a text input. So far so good; there are no errors or bugs, and we can continue to the next field.

3. Well need to either refresh the browser and go directly to the third field, or we will need to enter valid values for the **name** and **place of birth** fields before we can move on to the third field. Regardless of the method used, we'll try to enter a string for the **age** field. Once you have done that, attempt to move on to the next input field.

You will get an alert box again, telling you that you have entered an input of the wrong type. This is as per the plan, and is expected. Therefore, no errors or bugs yet.

4. Repeat the previous steps for the remaining fields, and attempt to move on to the next field as you enter the expected but illegal values.

For all of the remaining fields, you should receive alert boxes telling you that you have entered an input of the wrong type, which is what we expect and have planned for.

Input Case 2:

Now that we have completed the first test scenario, it is time to move on to the second test scenario, **where we try to enter non-alphanumeric values.**

1. The testing process is fairly similar to the first test. We will first refresh the browser, and then **immediately enter the non-alphanumeric values for the first input field—the name** input field. As per our **plan, we will enter ~!@#$%^&*()** as the **input, and then attempt to move on to the next input field.**

   For the first input field, which requires a character input, you should see an alert box telling that **only text input is allowed. If you see that, then our program works** as planned. Now let us move to the next step.

2. For the next input field, we'll repeat the previous step **and we should expect the** same output as well.

3. Now for the third input field, we proceed to enter **the same non-alphanumeric input** values. The only difference we should expect for this step is that the alert, **which** informs us that we have entered a wrong input, **will tell us that we need to enter** digits and not text.

4. We repeat the **previous steps for the remaining fields, and in general we should** expect to see an alert box informing us that we need to either enter text or enter digits, **depending on which input field it is. If this is the case, then all is well; there** are **no related errors or bugs for this test scenario.**

Input Case 3:

Now it is time to perform **the third test scenario, where we enter negative values for input** fields that require numerical inputs.

1. Once again, we'll refresh the browser to clear the old values. We'll proceed to enter the values as planned for the first two input fields. We will enter **Johnny Boy** and **San Francisco** for the input fields of **name** and **place of birth**, respectively.

2. Once you have performed the **previous step, enter -1** for the remaining input fields. As you enter **-1** for these fields, you should see that our program does not detect negative values. Instead, it gives an incorrect response telling us that we should enter digits.

   In reality, our program should be robust enough to spot negative values. However, as shown in the **previous tests, our program appears to have the incorrect response** to an illegal value. **Our program does spot the error, but it returns an incorrect** response. The response given is an alert box, telling you that the **input must be a digit**. This is technically incorrect, because our input is a digit, albeit a negative one.

This means that our program does spot negative values, but it returns an incorrect response. This means that we have a serious bug here. We need to take note of this bug in our sample documentation by documenting this error on the "Bug Report Form". You may make a look at how I have documented this in the `sample test plan` document.

Whew! This subsection is kind of long and tedious. That's right, testing can be tedious, and by now you should see that a good program design will incorporate the issues that we tested in this section. You will notice that, at least for our purposes here, checking of the input values to make sure that the input is what we need is fundamental to our program's success; if the input values are wrong, there is no point in testing the remaining program, as we are almost certain to receive a wrong output for a wrong input.

Test Case 2: Testing the program logic

In this subsection, we will attempt to test the robustness of the program in terms of the program logic. Although we have somewhat tested the program logic by ensuring that the input is correct, there is one more aspect that we need to test according to our test plan, and that is the present age and the retirement age.

Time for action – testing the program logic

In general, we will attempt to enter a retirement age that is less than the current age. Now let us test the robustness of the program:

1. Let us refresh the browser, and then we'll enter the values as per our plan. Well first enter **Johnny Boy** and then **San Francisco** for the input fields of **name** and **place of birth**, respectively.

2. Now, take note of this step: we will now enter **30** for **age and continue with the** other fields.

3. When you reach the input field **age at which you wish to retire**, you will want to enter a value that is less than the **age** field. As per our test plan, we will enter **25**. After this, we will attempt to move on to the next field.

   Because we were able to successfully move on to the next field, this means that our program is not robust enough. Our program should not accept a retirement age value that is less than the present age value. Therefore, even if our program does produce a final outcome, we can be sure that the output is not what we want, because the logic is already incorrect.

   As such, we will need to take note of the logical error found in this phase of the test. Well take note of this on the Bug Report Form once again. Now we will move on to the final stage of our test.

Test Case 3: Integration testing and testing unexpected values

We have reached the final phase of our test. In this subsection, we will move on to integrated testing by first testing the whole program by using expected and acceptable values, followed by breaking the flow of form submission by changing the values of the second form.

Time for action –Test Case 3a: testing the entire program with expected values

There are four sets of test values for the first test. In general, we will enter all values, and then submit the form to see if we are getting the response that we are expecting: the input values for input Case 1 and input Case 3 will result in an output stating that the user is not able to retire on time, and the input values for input Case 2 and input Case 4 will result in an output stating that the user will be able to retire on time. With that in mind, let us start with the first set of input values:

1. Going back to your web browser, refresh your program, or re-open the source code if you have closed the program. We'll enter the values as planned: **Johnny Boy** and **San Francisco** for **name** and **place of birth**.

2. Next, we'll enter **25** for **age** and then **1000** for **spending per month**. Well repeat these steps for the remaining values, until we see the **Submit** button that is dynamically generated on the second form.

3. Once you see the **Submit** button, click on the button to submit the values. You should see some text being generated in the **Final Response** box. If you see that the output contains the name, retirement age, the correct output value for the required amount of money we need to retire, and more importantly the response **you will be able to retire by 55 years old**, as shown in the following screenshot, then there are no bugs in the program.

Final response:

Hi **Johnny Boy**,

We've processed your information and are pleased to announce that you will be able to retire on time.

Base on your current spending habits, you will be able to retire by 55years old.

Also, you'll have' **33640000** amount of excess cash when you retire.

Congrats!

4. Now let us move on to entering the values for Case 2. Similarly, well refresh the browsers, and then **begin to enter all of the values as planned.**

5. When you see the **Submit** button that **is created dynamically, click on the button to** submit the form. In **this test case, you will see that the user will not {kind of crucial** difference!} be able to retire on time, as **shown in the following screenshot:**

Final response:

Hi **Johnny Boy,**

We've processed your information and we have noticed a problem.

Base on your current spending habits, you will not be able to retire by **55** years old.

You need to make another **1640000** dollars before you retire inorder to acheive our goal

You either have to increase your income or decrease your spending.

If you receive the output as **shown in the previous screenshot, then there are no** errors up to this point. So let's move on to the input values for the **third case.**

6. Refresh your browser again, and then start entering the values as planned. The values to take note of include the **salary per month** and **age at which you wish to retire**. In general, we have set the values in order to test if we can create the output to either be able to retire on time or be unable to retire on time.

7. Continue entering values until you see the **Submit** button that is dynamically generated. Click on the **Submit** button to submit the form. You will see the output as shown in the next screenshot:

Final response:

Hi **Johnny Boy,**

We've processed your information and we have noticed a problem.

Base on your current spending habits, you will not be able to retire by **28** years old.

You need to make another **964000** dollars before you retire inorder to acheive our goal

You either have to increase your income or decrease your spending.

If you received the previous output, then there are no errors or bugs.

8. Now, let us move on to the final case—case 4. We'll basically repeat the steps as done previously. I just need you to take note of the input values of **salary per month**. Notice that the input value is **100000**, and that the retirement age did not change. We are trying to simulate a situation where the user will be able to retire on time.

9. Continue to enter the values until you see the **Submit** button that is dynamically generated. Click on the **Submit** button to submit the form. You will see the output as shown in the next screenshot:

Final response:

Hi **Johnny Boy**,

We've processed your information and are pleased to announce that you will be able to retire on time.

Base on your current spending habits, you will be able to retire by **28**years old.

Also, you'll have' **2564000** amount of excess cash when you retire.

Congrats!

Once again, if you received the output shown in the previous screenshot, then you have received the correct output. And with this, we have completed the first part of this test phase.

In general, we have tested the whole program to see if we are getting the expected output. We used different values to generate the two possible outputs of being able to retire on time or being unable to retire on time. Not only have we have received the correct output, we have also tested the robustness of our functions in terms of calculating the outcome.

With the previous factors in mind, it is time to move on to the second phase of the test—testing the robustness of the second form.

Time for action – Test Case 3b: testing robustness of the second form

If you have been following me right from the first chapter, you may have noticed that we have have only disabled the input fields for the form on the left, and not the input fields on the right. Apart from doing it deliberately, to show you different aspects of JavaScript coding, we have set it up such that we can demonstrate to you other aspects of integrated testing. So now, well attempt to change the values of the dynamically-generated form and see what happens.

1. Well first refresh the browser, and then begin entering the input values according to the plan. After you have finished entering all of the values, change the values in the second form as per the test plan.

2. Now, submit the form, and you will see the output as displayed in the next screenshot:

Final response:

Hi **25**,

We've processed your information and are pleased to announce that you will be able to retire on time.

Base on your current spending habits, you will be able to retire by **Even more characters**years old.

Also, you'll have' **NaN** amount of excess cash when you retire.

Congrats!

Oops! Apparently, there is a fatal flaw in our program. There is no checking mechanism or whatsoever for our second form. The second form is present in case our users may want to change the values. Right from the start, we naïvely chose to believe that the user will enter legal and acceptable values on the second form, should they choose to change their input. Now that we know this might not be the case, we'll make a note of this on our Bug Report Form".

What just happened?

In general, we have executed the entire test plan. Along the way, we have uncovered bugs and errors that we will be fixing later. You may find the steps repetitive; that is true—testing can be repetitive sometimes. But, luckily, our program is quite small and hence testing it is manageable.

Now that we have completed the test, it is time to think about what we can do about those errors. We'll start talking about this in the next section.

What to do when a test returns an unexpected result

In general, when a test returns an unexpected or incorrect result, it means that there is a bug or error in our program. Based on our tests, you must certainly have noticed that there are weak points in our program. The weak points or errors that resulted in a test returning an unexpected result are as follows:

 - Our program does not support negative values
 - The code that we have written does not support boundary values (both maximum and minimum values)
 - The second form does not check for correctness in the input values; if we make any changes to the values in the second form, the program fails

These points mean that our code is not robust enough and we need to fix it; we will do this right away in the next section.

Regression testing in action

In this section, we will get our hands dirty by performing regression testing. We will attempt to simulate a situation that warrants regression testing by writing code that fixes the errors found when we initially applied our test plan. After writing the code, we will first test the code that we have written, after which we will test the entire application to see if it works in coherence.

Time for action – fixing the bugs and performing regression testing

We'll fix each of the bugs that we've uncovered, one by one. We'll start by writing a function that allows our program to support boundary values. The completed source code, where all of the errors have been corrected, is found in *Chapter 5* of the `source code` folder, and is entitled `perfect-code-for-JSLInt-enhanced.html`.

Before we move on to the actual coding process for the first bug, let us think about what we can do to support boundary values.

Firstly, if we go back to our sample test plan, you will notice that in our "Bug Report Form", we have documented that we can try to change the function that checks for form input such that it can check for minimum and maximum values. For simplicity's sake, we will enable boundary values by checking the length of the input. For example "Neo" would mean that there are three input characters and "1000" would have four input digits.

Secondly, because the checking of the input of the first form is done at `submitValues()`, we'll attempt to add in the required checking mechanism of this function. With that in mind, we can start the coding process:

1. Open the original source code that we wrote in *Chapter 3*, *Syntax Validation*, in your favorite source code editor, and look for the function `submitValues()`. Next, add the following code after the `debuggingMessages()` function:

```
// this is the solution for checking the length of the input
// this will allow us to enable boundary values
// starting with minimum values: we will accept character
// length of more than or equal than 3
// and less than 100 characters
if (elementObj.name === 'enterText') {
    if (elementObj.value.length <= 3) {
        alertMessage("Input must be more than 3 characters!");
        var element = document.getElementById(elementObj.id);
        jQuery(element).focus();

        return true;
    }
    if (elementObj.value.length >= 100) {
      alertMessage("Input must be less than 100 characters!");

        var element = document.getElementById(elementObj.id);
        jQuery(element).focus();

        return true;;
    }
}

// now for checking the maximum value of digits
// upper boundary is set at 10 digits
if (elementObj.name === 'enterNumber') {
    if (elementObj.value.length >= 10) {
        alertMessage("Input must be less than 10 digits!");
        var element = document.getElementById(elementObj.id);
    jQuery(element).focus();
        return true;
    }
}
```

What happened in the previous code is that we have added in a few `if` statements. These statements check for the type of input via the `.name` property, and then check to see if it is more than the minimum input or less than the maximum output. We have set a minimum input length of three characters and a maximum input characters of less than 100 length for text inputs. For input that requires numerical inputs, we have set a maximum input length of 10 digits. We did not set a minimum input length since it is possible that the user may not have any income.

2. Save your file and test the program. Try entering less than three characters or more than 100 characters. You should receive an alert box showing that you have too large or too small inputs. Similarly, test the input fields that require numerical inputs and see if the program detects an input length of more than 10 digits. If you have received the correct alert boxes for each of the different cases, then you have fixed the error.

   Now that we have fixed the issue regarding boundary values, it is time to move on to the next error that we have documented on our "Bug Report Form", which is the third error (bug number 3 in our `sample-testplan-bug-form-filled.doc`) that **we uncovered, which relates to negative values.**

   The error is that our program sees a negative input as a non-digit value and produces a wrong output message of **input must be digit**. Therefore, in this case we would need to fix this error by tracing back to **the source of the problem—the** functions that are responsible for checking the input.

   Take note that the function that checks the input is `submitValues()`. Now, let us move to the actual coding process:

3. Go back to your source code and start with the `submitValues()` function. We'll need to have a **mechanism that checks for negative input, and this will have to** return the correct output, which says that **input must be positive**. So here's what we can do:

```
// this is the solution for checking negative values
// this only applies to input fields that requires numeric
inputs
if (elementObj.name === 'enterNumber') {
    if (elementObj.value < 0) {
        alertMessage("("Input must be positive!");
    var element = document.getElementById(elementObj.id);
    jQuery(element).focus();
        return true;
    }
}
```

By adding the above code, you will be able to check for negative values. The above code should be placed within the `submitValues()` function, and before the `if` statement which checks for the length of the input.

4. Save your program and test it. Upon encountering fields that require numeric inputs, try entering a negative value, say **-1**. If you receive an alert box stating that **input must be positive**, then we have done it right.

The code for `submitValues()` should include the following lines shown below:

```
function submitValues(elementObj) {

// code above omitted
    // this is the solution for checking negative values
    // this only applies to input fields that requires numeric
inputs
    if (elementObj.name === 'enterNumber') {
        if (elementObj.value < 0) {
            alertMessage("Input must be positive!");
            var element = document.getElementById(elementObj.id);
        jQuery(element).focus();
            return false;
        }
    }
  // code below is omitted

}
```

The lines in the previous snippet are those lines that we added in this subsection. Because we have made sure that we are on the same frequency, we can move on to the fourth error (bug number 4 in our `sample_test_plan.doc`), which is regarding the program logic.

At the start of this chapter, we found out that our program does not detect that the retirement age can be smaller than the user's present age. This can be fatal for our program. Therefore, we need to add a mechanism that makes sure that the retirement age is greater than the user's present age.

Because the issue lies with the checking of inputs, we will need to turn our attention to `submitValues()`.

5. Let us go back to the source code, and add the following code to `submitValues()`:

```
    // this is to make sure that the retirement age is larger than
present age
    if (elementObj.id === 'retire') {
        if (elementObj.value < document.getElementById('age').
value) {
            alertMessage('Retirement age must be higher than
age');
            var element = document.getElementById(elementObj.id);
        jQuery(element).focus();

            return false;
        }
    }
```

You should enter this **code prior to the code done up in the previous subsection.**

Now, go ahead and test your code. Try entering a retirement age that is less than the current age. You should receive an alert message that says **retirement age must be larger than age**.

If you received this alert, then congratulations, you have got it right! Once again, to quickly sum up this section and to make sure that we are on the same page, `submitValues()` should include the lines of code as shown next:

```
function submitValues(elementObj) {

// code above omitted

    // this is to make sure that the retirement age is larger than
present age
    if (elementObj.id === 'retire') {
        if (elementObj.value < document.getElementById('age').
value){
            alertMessage('retirement age must be larger than
age');
            var element = document.getElementById(elementObj.id);
        jQuery(element).focus();

            return true;
        }
    }
  // code below omitted

}
```

Now let us move on to the last error (bug number 5 in our `sample-testplan-bug-form-filled.doc`) that we have uncovered by checking the second form.

We have created our JavaScript program such that as we enter the values for each input field, a new input field is created dynamically. This means that after all of the input fields have been completed, a new form will be created. In case you didn't notice, the new input fields that are created allow users to change their values.

The issue here is that the user may change the input values in the new form, which can result in fatal errors as we have no checking mechanisms in place to check the values in the second form. So, we naïvely chose to believe that the user will act accordingly, and only enter valid values. But obviously, we were wrong.

Therefore, in order to check the second form, we would most likely have to create a new function that checks the second form.

Although the second form is generated dynamically, we can still get the values within those fields through the methods that we have learned so far. Remember that because JavaScript has created the fields in the second form, these fields technically exist in memory and are therefore still accessible.

With that in mind, we'll need to create a function that works on these fields.

6. Open the source code, and scroll to the last function, which uses jQuery statements. Before this function, create the following function:

```
function checkSecondForm(elementObj) {
  // some code going here
}
```

7. Well first start by checking for empty values. Therefore here's what we can do to check for empty values:

```
if(document.testFormResponse.nameOfPerson.value === "") {
        alertMessage("fields must be filled!");
        return false;
    }
    if(document.testFormResponse.birth.value === "") {
        alertMessage("fields must be filled!");
        return false;
    }
    if(document.testFormResponse.age.value === "") {
```

```
        alertMessage("fields must be filled!");
        return false;
    }
    if(document.testFormResponse.spending.value === "") {
        alertMessage("fields must be filled!");
        return false;
    }
    if(document.testFormResponse.salary.value === "") {
        alertMessage("fields must be filled!");
        return false;
    }
    if(document.testFormResponse.retire.value === "") {
        alertMessage("fields must be filled!");
        return false;
    }
    if(document.testFormResponse.retirementMoney.value === "") {
        alertMessage("fields must be filled!");
        return false;
    }
```

In general, we apply what we have learned in the third chapter by using `===` instead of `==` when checking for empty values. We basically check the values that are found in the dynamically-generated fields, and check to see if they are empty.

Now that we have the code that checks to see if the fields are empty, it is time to write code that checks for the correct type of input.

8. We can apply the techniques learned in *Chapter 3*, *Syntax Validation*, to check for the correctness of the input. In general, we are using regular expression, as we did in the previous chapters, to check for the input's type. Heres what we can do:

```
    var charactersForName =  /^[a-zA-Z\s]*$/.test(document.
testFormResponse.nameOfPerson.value);
    var charactersForPlaceOfBirth =  /^[a-zA-Z\s]*$/.
test(document.testFormResponse.birth.value);
    var digitsForAge = /^\d+$/.test(document.testFormResponse.age.
value);
    var digitsForSpending = /^\d+$/.test(document.
testFormResponse.spending.value);
    var digitsForSalary = /^\d+$/.test(document.testFormResponse.
salary.value);
    var digitsForRetire = /^\d+$/.test(document.testFormResponse.
retire.value);
    var digitsForRetirementMoney = /^\d+$/.test(document.
testFormResponse.retirementMoney.value);
```

```
    // input is not relevant; we need a digit for input elements
with name "enterNumber"
    if (charactersForName === false || charactersForPlaceOfBirth
=== false) {
        alertMessage("the input must be characters only!");
        debuggingMessages( checkSecondForm", elementObj, "wrong
input");
        return false;
    }

    else if (digitsForAge === false || digitsForSpending === false
|| digitsForSalary === false || digitsForRetire === false ||
digitsForRetirementMoney === false ){
        alertMessage("the input must be digits only!");
        debuggingMessages( checkSecondForm", elementObj, "wrong
input");
        return false;
    }
    // theinput seems to have no problem, so we'll process the
input
    else {
        checkForm(elementObj);
        alert("all is fine");
        return false;
    }
```

For a complete version of the previous code, please check the `source code` folder of *Chapter 5*, and refer to the file `perfect-code-for-JSLInt-enhanced.html`.

However, remember that in the earlier debugging sessions we have created new checking mechanisms in order to **support boundary values**, **prevent negative values**, and to make sure that the retirement age is greater than the user's current age.

Because the second form may be changed, the previous errors can occur in the second form as well. Therefore, we'll need to add those checking mechanisms as well. To see if you have done it correctly, check the `checkSecondCode()` function in the `source code` folder for the file entitled `perfect-code-for-JSLInt-enhanced.html`. Here's a code snippet of `checkSecondCode()`:

```
// above code omitted
    if (elementObj.id === 'retire') {
        if (elementObj.value < document.getElementById('age').
value) {
            alertMessage('retirement age must be larger than age');
             var element = document.getElementById(elementObj.id);
        jQuery(element).focus();
```

```
            return true;
        }
    }

    // this is the solution for checking negative values
    // this only applies to input fields that requires numeric
inputs
    if (elementObj.name === 'enterNumber') {
        if (elementObj.value < 0) {
            alertMessage("Input must be positive!");
            var element = document.getElementById(elementObj.id);
        jQuery(element).focus();

            return true;
        }
    }

    if (elementObj.name === 'enterText') {
        if (elementObj.value.length <= 3) {
            alertMessage("Input must be more than 3 characters!");
            var element = document.getElementById(elementObj.id);
        jQuery(element).focus();

            return true;
        }
        if (elementObj.value.length >= 100) {
            alertMessage("Input must be less than 100
characters!");
            var element = document.getElementById(elementObj.id);
        jQuery(element).focus();

            return true;
        }
    }
    if (elementObj.name === 'enterNumber') {
        if (elementObj.value.length >= 10) {
            alertMessage("Input must be less than 10 digits!");
            var element = document.getElementById(elementObj.id);
        jQuery(element).focus();
            return true;
        }
    }

  // remaining code omitted
}
```

What just happened?

We have now finished executing the entire test plan, including regression testing. Notice that at each phase of the coding process we carried out small tests to make sure that our solution works correctly; we have used unit testing once again in our regression testing process.

Also note that we tested the program incrementally; we tested each new function or code that we created and made sure that it worked correctly, before we moved on to fixing the next error.

By going through this process, we will have a much better chance of creating good programs and will avoid introducing new errors into our code.

Apart from performing regression testing as a part of an ongoing process as our program changes, there are other important issues regarding the testing of our program. Let us move to the first important issue—performance issues.

Performance issues—compressing your code to make it load faster

As I mentioned in *Chapter 4*, *Planning to Test*, the performance of the code that we write is dependent on various factors. Performance in general refers to the speed of the execution of your code; this is dependent on the algorithms you use for your code. Because algorithm issues are beyond the scope of this book, let us focus on something that is much easier to achieve, like enhancing the programs performance by compressing your code.

In general, after compressing your code, the file size of your code will be smaller and hence lowers disk usage in the cache that is required to store the code before execution. It also reduces the amount of bandwidth required to transfer your JavaScript file from the web server to the client. So now, let us see how we can compress our JavaScript code.

There are two ways in which we can go about doing this:

1. We can compress our entire program, which means that we will compress our CSS, HTML, and JavaScript together.
2. We can remove all of the local JavaScript and place it in an external file, and then, compress only the external JavaScript. To keep things simple, I'll start by using the first method.

Firstly, I want you to visit `http://jscompress.com/` and copy and paste our source code into the input box. Theres an option called **"Minify (JSMin)"**. This option will compress HTML, CSS, and JavaScript all together. Once you have copied the code into the input box, click on **Compress JavaScript**.

You will then see the page refresh and the minified code will be displayed within the input box. Copy and paste that code into a new file, and then save it as `testing-compressed.html`.

If you go to the `source code` folder, you will notice that I have already done the compression process for you. Check the size of the file for `testing-compressed.html` and the code that we wrote earlier. Based on the source code that we have, the compressed version is 12KB, whereas the original version is 18KB.

Now let us try the second method—placing all of the JavaScript in an external JavaScript file and compressing that. Heres what we will do:

1. Cut all of the JavaScript found between the `<head>` and `</head>` tags, and paste it into a new document called `external.js`.

2. Save `external.js`, and also save your changes to the HTML document.

3. Go back to your HTML document, go to the `<head>` and `</head>` tags and insert the following: `<script type="text/javascript" src="external.js">` between them. Then save the file.

So there you have it! We have compressed your code so that your code gets loaded faster from the web server to the client side.

It seems that we have managed to reduce the file size by compressing the code. Of course, the difference is not that obvious because our code isn't much. However, in reality code can go up to thousands or even tens of thousands of lines of code, we have seen with the jQuery library. In such cases, code compression will help to enhance performance.
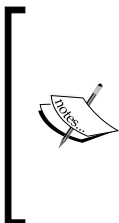
> If you are a developer who is working under the terms of an Non-Disclosure Agreement (NDA), there is a likelihood that you are not allowed to use any of the external services that I have previously mentioned. If this is the case, you might want to consider using Yahoo's YUI Compressor, which allows you to work directly from the command line. For more information, visit `http://developer.yahoo.com/yui/compressor/#using`.

Does using Ajax make a difference?

Let me start by briefly explaining what happens when you are using Ajax. JavaScript is part of the Ajax equation; the execution of JavaScript is responsible for sending information to and loading information from the server. This is achieved by using `XMLHttpRequest` object.

When the sending and loading of data to and from the server is done using Ajax, the testing responsibilities are different; you will not only have to test for the various errors that we have covered in the previous chapters, but you will also have to test if each error has resulted in the successful sending and loading of information and the correct visual response to the user.

However, because you are sending and receiving requests to and from the server, you might have to perform some form of server-side testing. This brings us to the next part of the topic— the difference between JavaScript testing and server-side testing.

Difference from server-side testing

As mentioned in the previous section, when you are performing tests for Ajax, you might have to perform server-side testing. In general, the concepts that you have picked up to this point in the book can also be applied to server-side testing. Therefore, conceptually, there should be little difference between JavaScript testing and server-side testing.

However, do take note that server-side testing typically includes serve-side code and most probably includes databases such as MySQL, PostgreSQL, and others. This means that the complexities involving server-side testing can take a different form when compared to JavaScript testing.

Nonetheless, you will be expected to have a good understanding of the server-side language used, the database used, and so on. This is the bare minimum for you to get started with planning your tests.

> If you are performing server-side testing as a part of Ajax testing, you will most certainly want to know about Hypertext Transfer Protocol response status codes. These status codes are a way to determine whether your request was successful or not. They even tell you what kind of errors occurred, should any occur. For more information, visit: `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`.

What happens if you visitor turns off JavaScript

We have briefly covered the issues of whether we should write applications that support users who have had their JavaScript turned off. Although there are different points of view on whether or not we should support such users, one of the best ways, in my humble opinion, is that we should at least inform our users that their browser does not support JavaScript (or that JavaScript is turned off) and they might be missing out on something. In order to achieve this, we can use the following code snippet:

```html
<html>
<body>
<script type="text/javascript">
document.write("Your browser supports JavaScript, continue as
usual!");
// do some other code as usual since JavaScript is supported
</script>
<noscript>
```

```
Sorry, your browser does not support JavaScript! You will need to
enable JavaScript in order to enjoy the full functionality and
benefits of the application
</noscript>
</body>
</html>
```

Note that we used the `<noscript>` tag, which is a way to show user's alternative content when JavaScript is turned off or is not supported.

Now that we have almost reached the end of this chapter, you must be getting the hang of it. Let us see if you can improve upon your skills by trying out the following assignment.

Have a go hero – enhance the usability of our program

Now that you have come this far, you might want to take a shot at this task—enhance the usability of this program by doing the following:

◆ Make sure that the user enters the required information, starting from the first field to the last field.

The other issue that we might have with our program is that the user might click on any input field other than the first one and begin entering the information. Although this may not directly affect the correctness of our program, there might be a chance that the result is not what we expect.

◆ With regards to the second form, is there any way that you can inform your user which input fields have the wrong input? Can your user change an input that is incorrect?

When we are fixing the bug related to the second form, we only created mechanisms to detect the correctness of the input in the second form. However, should the user enter an incorrect value in the second form, the user might not know immediately which fields were entered incorrectly.

Here are some tips to help you get started with this exercise:

◆ Right from the start, you can disable all of the input fields apart from the first one. Then as the first field gets the correct input, you can enable the second input field. Similarly, when the second input field is completed correctly, the third input field gets enabled, and so on and so forth.

◆ For the second issue, you might want to take a look at our code and see if you can edit the conditions found in the `if else` statements for the function `checkSecondForm()`. What I have done is to lump all of the possibilities into a single `if` or `else if` statement, thus making it impossible to detect which field has gone wrong. You can try to split up the conditions such that each of the `if` and `else if` statements contain only a single condition. That way, well be able to create a custom response for each individual input field in the second form, should anything go wrong.

Summary

Wow, we have covered a lot in this chapter. We have executed the test plan and have uncovered bugs. Next we successfully fixed the bugs that we uncovered. After fixing each bug, we performed regression testing in order to make sure that the original functionality was retained and that no new bugs were introduced into the program.

Specifically, we covered the following topics:

◆ How to execute a test plan and how to document bugs that we uncovered

◆ How to perform regression testing after fixing each bug

◆ How to compress the code in order to enhance performance

◆ Testing differences if we use Ajax

◆ Differences between client-side testing and server-side testing

The previous learning points may seem small, but now that you have gone through this chapter, you should know that carrying out a test plan and subsequently fixing the bugs can be tedious.

Now that we have covered the execution of test plan, its time to move on to something slightly more difficult—testing more complex code. Notice that we have been dealing with JavaScript in a one-dimensional manner: we placed all of our JavaScript in our HTML file, along with CSS. We have been developing JavaScript code as this was the only piece of JavaScript code that we were using. But, in reality, it is usual to see web applications using more than one piece of JavaScript code; this additional code is usually attached via an external JavaScript file.

More importantly, this is not going to be the only issue that we will face in the real world. As our code gets more complex, we will need to use more sophisticated testing methods, or even use tools such as built-in consoles, to help us test more efficiently and effectively.

We will cover the previously-mentioned issues in the next chapter, *Chapter 6, Testing more complex code*. See you there!

6

Testing More Complex Code

Welcome to the sixth chapter. In this chapter, we will learn more about JavaScript testing. More specifically, we'll learn how to test more complex code, where there will be more interactions between entities. Until now, we have been performing tests on relatively simple code, using fairly straightforward techniques.

More specifically, we'll cover the following:

- ◆ Types of errors that can occur when combining scripts
- ◆ How we can deal with the errors that occur when combining the scripts together
- ◆ Various JavaScript libraries available on the Internet right now, and the issues that we need to consider when testing them
- ◆ How to test the GUI, widgets add-ons for libraries, and other considerations
- ◆ How to use the console log
- ◆ Performing exception handling by using JavaScript built-in objects
- ◆ Testing an application by using JavaScript built-in objects

Let us get started with the basic concepts by covering the kinds of errors that can occur when combining scripts.

Issues with combining scripts

So far, we have been focused on writing and testing only one JavaScript code within our HTML document. Consider the real-life situation where we typically use external JavaScript; what happens if we use more than one JavaScript file? What kind of issues can we expect if we use more than one external JavaScript file? We'll cover all of this in the subsections below. We'll start with the first issue—combining event handlers.

Combining event handlers

You may or may not have realized this, but we have been dealing with event handlers since *Chapter 3*, *Syntax Validation*. In fact, we actually mentioned events in *Chapter 1*, *What is JavaScript Testing*. JavaScript helps to bring life to our web page by adding interactivity. Event handlers are the heartbeat of interactivity. For example, we click on a button and a pop-up window appears, or we move our cursor over an HTML `div` element and the element changes color to provide visual feedback.

To see how we can combine event handlers, consider the following example, which is found in the `source code` folder in the files `combine-event-handlers.html` and `combine-event-handlers.js` as shown in the following code:

In `combine-event-handlers.html`, we have:

```
<html>
  <head>
    <title>Event handlers</title>
    <script type="text/javascript" src="combine-event-
                handlers.js"></script>
  </head>
  <body>
    <div id="one" onclick="changeOne(this);"><p>Testing One</p></div>
    <div id="two" onclick="changeTwo(this);"><p>Testing Two</p></div>
    <div id="three" onclick="changeThree(this);"><p>Testing
                Three</p></div>
  </body>
</html>
```

Notice that each of the `div` elements is handled by different functions, namely, `changeOne()`, `changeTwo()`, and `changeThree()` respectively. The event handlers are found in `combine-event-handlers.js`:

```
function changeOne(element) {
    var id = element.id;
    var obj = document.getElementById(id);
    obj.innerHTML = "";
```

```
        obj.innerHTML = "<h1>One is changed!</h1>";
        return true;
    }

    function changeTwo(element) {
        var id = element.id;
        var obj = document.getElementById(id);
        obj.innerHTML = "";
        obj.innerHTML = "<h1>Two is changed!</h1>";
        return true;
    }
    function changeThree(element) {
        var id = element.id;
        var obj = document.getElementById(id);
        obj.innerHTML = "";
        obj.innerHTML = "<h1>Three is changed!</h1>";
        return true;
    }
```

You might want to go ahead and test the program. As you click on the text, the content changes based on what is defined in the functions.

However, we can rewrite the code such that all of the events are handled by one function. We can rewrite `combine-event-handlers.js` as follows:

```
    function combine(element) {
        var id = element.id;
        var obj = document.getElementById(id);
        if(id == "one"){
            obj.innerHTML = "";
            obj.innerHTML = "<h1>One is changed!</h1>";
            return true;
        }
        else if(id == "two"){
            obj.innerHTML = "";
            obj.innerHTML = "<h1>Two is changed!</h1>";
            return true;
        }
        else if(id == "three"){
            obj.innerHTML = "";
            obj.innerHTML = "<h1>Three is changed!</h1>";
            return true;
        }
        else{
         ; // do nothing
        }
    }
```

When we use `if else` statements to check the `id` of the `div` elements that we are working on, and change the HTML contents accordingly, we will save quite a few lines of code. Take note that we have renamed the function to `combine()`.

Because we have made some changes to the JavaScript code, we'll need to make the corresponding changes to our HTML. So `combine-event-handlers.html` will be rewritten as follows:

```
<html>
  <head>
    <title>Event handlers</title>
    <script type="text/javascript" src="combine-event-
              handlers.js"></script>
  </head>
  <body>
    <div  id="one" onclick="combine(this);"><p>Testing One</p></div>
    <div  id="two" onclick="combine(this);"><p>Testing Two</p></div>
    <div  id="three" onclick="combine(this);"><p>Testing
           Three</p></div>
  </body>
</html>
```

Notice that the `div` elements are now handled by the same function, `combine()`. These rewritten examples can be found in `combine-event-handlers-combined.html` and `combine-event-handlers-combined.js`.

Naming clashes

Removing name clashes is the next issue that we need to deal with. Similar to the issue of combining event handlers, naming clashes occur when two or more variables, functions, events, or other objects have the same name. Although these variables or objects can be contained in different files, these name clashes do not allow our JavaScript program to run properly. Consider the following code snippets:

In `nameclash.html`, we have the following code:

```
<html>
  <head>
  <title>testing</title>
    <script type="text/javascript" src="nameclash1.js"></script>
  </head>
  <body>
    <div id="test" onclick="change(this);"><p>Testing</p></div>
  </body>
</html>
```

In `nameclash1.js`, we have the following code:

```
function change(element) {
    var id = element.id;
    var obj = document.getElementById(id);
    obj.innerHTML = "";
    obj.innerHTML = "<h1>This is changed!</h1>";
    return true;
}
```

If you run this code by opening the file in your browser and clicking on the text **Testing**, the HTML contents will be changed as expected. However, if we add `<script type="text/javascript" src="nameclash2.js"></script>` after the `<title></title>` tag, and if the content of `nameclash2.js` is as follows:

```
function change(element) {
    alert("so what?!");
}
```

Then we will not be able to execute the code properly. We will see the alert box instead of the HTML contents being changed. If we switch the arrangement of the external JavaScript, then the HTML contents of the `div` elements will be changed and we will not be able to see the alert box.

With such naming clashes, our program becomes unpredictable; the solution to this is to use unique names in your functions, classes, or events. If you have a relatively large program, it would be advisable to use namespaces, which is a common strategy in several JavaScript libraries such as YUI and jQuery.

Using JavaScript libraries

There are many JavaScript libraries available right now. Some of the most commonly-used ones are as follows:

- JQuery (`http://jquery.com`)
- YUI (Yahoo! User Interface JavaScript library) (`http://developer.yahoo.com/yui/`)
- Dojo (`http://dojotoolkit.org/`)
- Prototype (`http://www.prototypejs.org/`)
- Mootools (`http://mootools.net/`)
- Script.aculo.us (`http://script.aculo.us/`)

There are many more JavaScript libraries out there. For a complete list, feel free to visit `http://en.wikipedia.org/wiki/List_of_JavaScript_libraries`.

If you have considered using JavaScript libraries, you may be aware of the benefits of using a JavaScript library. Issues such as event handling, and the much dreaded cross-browser issues make it necessary to consider using a JavaScript library. But you might want to know what you should look out for when selecting a JavaScript library as a beginner JavaScript programmer. So here is a list of things to consider:

- The level of available support, in terms of documentation.
- Whether tutorials are available, and whether they are free or paid for. This helps you to speed up the programing process.
- The availability of plugins and add-ons.
- Does the library have a built-in testing suite? This is very important, especially for our purposes here.

Do you need to test a library that someone else has written?

Firstly, while we are learning about JavaScript testing, I would say that for a beginner learning JavaScript programing, it might not be advisable to test JavaScript libraries that someone else wrote. This is because we need to understand the code in order to perform accurate tests. People who are able to conduct objective (and accurate) tests are JavaScript experts, and although you are on your way to becoming one, you are probably not there yet.

Secondly, from a practical standpoint, many such tests have already been done for us. All you need to do is search for them on the Internet.

But for learning purposes, let us have a brief look at what tests are usually run against library code.

What sort of tests to run against library code

In general, as a user of various JavaScript libraries, we would most commonly perform performance testing and profiling testing.

Performance testing

Performance testing, as the name suggests, is about testing the performance of your code. This includes testing how fast your code runs (on various browsers) in a manual way, or by using certain tools such as Firebug or others (more such tools are covered in *Chapter 8*).

In general, in order to generate accurate results for performance testing, it is important for you to test your code (most preferably by using tools and test suites) against all popular platforms. For example, a common way to performance test JavaScript code id to install Firebug in Firefox and test your code using that. But to think of it from a practical standpoint, Firefox users only make up approximately a quarter (or a third at the most) of the total number of Internet users. You will have to test your code against other platforms such as Internet Explorer in order to make sure that your code is up to the mark. We'll cover more of this in *Chapter 8*.

Profiling testing

Profiling testing is similar to performance testing, except that it focuses on bottlenecks in your code rather than the overall performance. Bottlenecks are, in general, the main culprits for inefficient code. Fixing bottlenecks is (almost) a sure way to enhance the performance of your code.

GUI and widget add-ons to libraries and considerations on how to test them

If you have checked the list of various JavaScript libraries that I pointed you to, you may have noticed that some of the JavaScript libraries provide user interface or widget add-ons as well. These are meant to enhance your application's user interface, and most importantly will help you to save time and effort by implementing commonly-used user interface components, such as dialog boxes, color selectors, and so on.

But that's where the problem starts—how do we test such user interface and widget add-ons? There are many ways in which we can go about doing that, but the simplest way (and perhaps the most cumbersome) would be to test visually and manually. For example, if we are expecting a dialog box to appear at the top left-hand side of the screen with a certain color, width, and height, and it does not appear the way we want, then something is wrong.

Similarly, if we see something that we expect to see, then we can say that it is correct—at least in a visual sense.

However, more vigorous testing is required. Testing user interfaces can be a daunting task, and hence I would suggest that you use testing tools such as Sahi, which allows us to write automated web application UI tests in any programing language. Tools such as Sahi are out of scope for this chapter. We will cover Sahi in detail in *Chapter 8*. Meanwhile, if you are eager to check out Sahi, feel free to visit their website at `http://sahi.co.in`.

Deliberately throwing your own JavaScript errors

In this section, we will learn how to throw our own JavaScript errors and exceptions. We will briefly cover the syntax of the error functions and commands. It may be a little incomprehensible at this stage to just give you the syntax, but this is necessary. Once you understand how to make use of these commands and reserved words, you will see how you can make use of them to give yourself more specific information (and hence more control) over the types of errors that you can catch and create in the next section. So let us get started with the first reserved word—`throw`.

The throw statements

`throw` is a statement that allows you to create an exception or error. It is a bit like the `break` statement, but `throw` allows you to break out of any scope. In general, this is what we usually use to literally throw an error. The syntax is as follows:

```
throw(exception);
```

We can use `throw(exception)` in the following ways:

```
throw "This is an error";
```

or:

```
throw new Error("this is an error");
```

`Error` is a built-in object that is commonly used in conjunction with the `throw` statement; we will cover `Error` later. The important thing to understand now is the syntax, and the fact that `throw` is also often used with `try`, `catch`, and `finally`, which will help you to control the program flow and create accurate error messages. Now let us move on to `catch`.

The try, catch, and finally statement

The `try`, `catch`, and `finally` statement are JavaScript's exception handling mechanism, which, as mentioned previously, helps you control the program flow, while catching your errors. The syntax of the `try`, `catch`, and `finally` statements is as follows:

```
try {
  // exceptions are handled here
}
catch (e) {
  // code within the catch block is executed if any exceptions are
caught in the try block
}
finally {
  // code here is executed no matter what happens in the try block
}
```

Notice that `try` is followed by `catch`, and then `finally` can be used optionally. In general, the `catch` statement catches the exceptions that occur in the `try` statement. An exception is an error. The `finally` statement is executed as long as the `try` or `catch` statement terminates.

Now that we have covered the basic commands and reserved words for deliberately throwing JavaScript errors, let us take a look at an example of how `try`, `catch`, and `finally` can be used together. The following code can be found in the `source code` folder of *Chapter 6*, in the HTML document named `try-catch-finally-correct-version.html`. Check out the following code:

```html
<html>
  <head>
    <script>
      function factorial(x) {
        if(x == 0) {
          return 1;
        }
          else {
            return x * factorial(x-1);
        }
  }

  try {

      var a = prompt("Enter a positive integer", "");

      var f = factorial(a);

      alert(a + "! = " + f);
  }
  catch (error) {
      // alert user of the error
      alert(error);
      alert(error.message);
  }
  finally {
      alert("ok, all is done!");
  }

    </script>
  </head>
  <body>
  </body>
</html>
```

You can copy and paste the code above into your favorite text editor, save it, and run it in your browser. Or you can run the sample file `try-catch-finally-correct-version.html`.

You will see a prompt window asking you to enter a positive integer. Go ahead and enter a positive integer, say **3** for instance, and you will receive an alert window telling you **3! = 6**. After that, you should receive another alert window, which contains the message **ok, all is done!**, as the `finally` block will be executed after `try` or `catch` terminates.

Now, enter a negative number, say **-1**. If you are using Firefox, you will receive an alert window that says that you have too much recursion. If you are using Internet Explorer, you will receive an **[object Error]** message.

After the first pop-up window, you will receive a second pop-up window. If you are using Firefox, you will see an **InternalError: Too much recursion** message. If you are using Internet Explorer, you will receive an **Out of stack space** message.

Lastly, you should receive a final alert window, which contains the message **ok, all is done!**, as the `finally` block will be executed after `try` or `catch` terminates. While it is true that we have an error, the error message is not exactly what we need, as it does not tell us that we have entered an illegal value.

This is where `throw` comes in. `throw` can be used to control the program flow and give us the correct response for each type of error. Check out the following code, which can also be found in the `source code` folder, in the file `try-catch-finally-throw-correct-version.html`.

```
<html>
<head>
<script>
function factorial(x) {
    if(x == 0) {
        return 1;
    }
    else {
        return x * factorial(x-1);
    }

}

try {

    var a = prompt("Please enter a positive integer", "");
    if(a < 0){
        throw "negative-error";
    }
    else if(isNaN(a)){
```

```
            throw "not-a-number";
        }
        var f = factorial(a);

        alert(a + "! = " + f);
    }
    catch (error) {
        if(error == "negative-error") {
            alert("value cannot be negative");
        }
        else if(error == "not-a-number") {
            alert("value must be a number");
        }
        else
            throw error;
    }
    finally {
        alert("ok, all is done!");
    }

    </script>
    </head>
    <body>
    </body>
    </html>
```

Now go ahead and execute the program, and enter correct values, negative values, and non-alphanumeric values. You should receive the correct error messages depending on your input.

Notice the previous lines of code where we used the `throw` statement to control the types of error messages, which will be shown to the user in the `catch` block. This is one way in which `throw` statements can be used. Note that the string that is defined after `throw` is used to create program logic to decide what error messages should be called.

In case you are wondering what other capabilities this exception handling mechanism has, remove the `factorial` function from `try-catch-finally-correct-version.html`. Alternatively, you can open the file `try-catch-finally-wrong-version.html` and run the program. Then try entering any value. You should receive an alert message telling you that the `factorial` function is not defined, and after that you will receive another alert box saying **ok, all is done!**. Notice that, in this case, there is no need for us to write any form of message; `catch` is powerful enough to tell us what went wrong.

One thing to note, though, is that the JavaScript runtime may catch an exception if you do not write an exception handler.

Now that we have covered the basics of the exception handling mechanism, let us move on to the specifics—built-in objects for handling errors.

Trapping errors by using built-in objects

In this section, we'll briefly describe what each type of built-in object is, along with its syntax, before we show some examples of how each of the built-in objects work. Do take note that the alert messages, which we will be using sparingly in the examples, are based on the Firefox browser. If you try the code on Internet Explorer, you might see different error messages.

The Error object

An `Error` is a generic exception, and it accepts an optional message that provides details of the exception. We can use the `Error` object by using the following syntax:

```
new Error(message); // message can be a string or an integer
```

Here's an example that shows the `Error` object in action. The source code for this example can be found in the file `error-object.html`.

```
<html>
<head>
<script type="text/javascript">
function factorial(x) {
    if(x == 0) {
        return 1;
    }
    else {
        return x * factorial(x-1);
    }

}
try {
    var a = prompt("Please enter a positive integer", "");
    if(a < 0){
        var error = new Error(1);
        alert(error.message);
     alert(error.name);
        throw error;
    }
    else if(isNaN(a)){
```

```
        var error = new Error("it must be a number");
        alert(error.message);
        alert(error.name);
        throw error;
    }
    var f = factorial(a);

    alert(a + "! = " + f);
}
catch (error) {
    if(error.message == 1) {
        alert("value cannot be negative");
    }
    else if(error.message == "it must be a number") {
        alert("value must be a number");
    }
    else
        throw error;
}
finally {
    alert("ok, all is done!");
}
</script>
</head>
<body>
</body>
</html>
```

You may have noticed that the structure of this code is similar to the previous examples, in which we demonstrated `try`, `catch`, `finally`, and `throw`. In this example, we have made use of what we have learned, and instead of throwing the error directly, we have used the `Error` object.

I need you to focus on the code given above. Notice that we have used an integer and a string as the message argument for `var error`, namely `new Error(1)` and `new Error("it must be a number")`. Take note that we can make use of `alert()` to create a pop-up window to inform the user of the error that has occurred and the name of the error, which is **Error**, as it is an `Error` object. Similarly, we can make use of the message property to create program logic for the appropriate error message.

It is important to see how the `Error` object works, as the following built-in objects, which we are going to learn about, work similarly to how we have seen for the `Error` object. (We might be able to show how we can use these errors in the console log.)

The RangeError object

A `RangeError` occurs when a number is out of its appropriate range. The syntax is similar to what we have seen for the `Error` object. Here's the syntax for `RangeError`:

```
new RangeError(message);
```

`message` can either be a string or an integer.

We'll start with a simple example to show how this works. Check out the following code that can be found in the `source code` folder, in the file `rangeerror.html`:

```
<html>
<head>
<script type="text/javascript">
try {
    var anArray = new Array(-1);
    // an array length must be positive
}
catch (error) {
    alert(error.message);
    alert(error.name);
}
finally {
    alert("ok, all is done!");
}
</script>
</head>
<body>
</body>
</html>
```

When you run this example, you should see an alert window informing you that the array is of an invalid length. After this alert window, you should receive another alert window telling you that **The error is RangeError**, as this is a `RangeError` object. If you look at the code carefully, you will see that I have deliberately created this error by giving a negative value to the array's length (array's length must be positive).

The ReferenceError object

A `ReferenceError` occurs when a variable, object, function, or array that you have referenced does not exist. The syntax is similar to what you have seen so far and it is as follows:

```
new ReferenceError(message);
```

`message` can either be a string or an integer.

As this is pretty straightforward, I'll dive right into the next example. The code for the following example can be found in the `source code` folder, in the file `referenceerror.html`.

```html
<html>
<head>
<script type="text/javascript">
try {
    x = y;
    // notice that y is not defined
    // an array length must be positive
}
catch (error) {
    alert(error);
    alert(error.message);
    alert(error.name);
}
finally {
    alert("ok, all is done!");
}
</script>
</head>
<body>
</body>
</html>
```

Take note that `y` is not defined, and we are expecting to catch this error in the `catch` block. Now try the previous example in your Firefox browser. You should receive four alert windows regarding the errors, with each window giving you a different message. The messages are as follows:

- ReferenceError: y is not defined
- y is not defined
- ReferenceError
- ok, all is done

If you are using Internet Explorer, you will receive slightly different messages. You will see the following messages:

- [object Error] message
- y is undefined
- TypeError
- ok, all is done

The TypeError object

A `TypeError` is thrown when we try to access a value that is of the wrong type. The syntax is as follows:

```
new TypeError(message); // message can be a string or an integer and
it is optional
```

An example of `TypeError` is as follows:

```
<html>
<head>
<script type="text/javascript">
try {
    y = 1
    var test = function weird() {
        var foo = "weird string";
    }
    y = test.foo(); // foo is not a function

}
catch (error) {
    alert(error);
    alert(error.message);
    alert(error.name);
}
finally {
    alert("ok, all is done!");
}
</script>
</head>
<body>
</body>
</html>
```

If you try running this code in Firefox, you should receive an alert box stating that it is a `TypeError`. This is because `test.foo()` is not a function, and this results in a `TypeError`. JavaScript is capable of finding out what kind of error has been caught. Similarly, you can use the traditional method of throwing your own `TypeError()`, by uncommenting the code.

The following built-in objects are less used, so we'll just move through quickly with the syntax of the built-in objects.

The SyntaxError object

A `SyntaxError` occurs when there is an error in syntax. The syntax for `SyntaxError` is as follows:

```
new SyntaxError([message,[,,[,filename[, lineNumber]]]); // message
can be a string or an integer and it is optional
```

Take note that the `filename` and `lineNumber` parameters are non-standard, and they should be avoided if possible.

The URIError object

A `URIError` occurs when a malformed URI is encountered. The syntax is as follows:

```
new URIError([message,[,filename[, lineNumber]]]);
```

Similar to `SyntaxError`, take note that the `filename` and `lineNumber` parameters are non-standard, and they should be avoided if possible.

The EvalError object

An `EvalError` occurs when an `eval` statement is used incorrectly or contains an error other than a syntax error.

```
new EvalError([message,[,filename[, lineNumber]]]);// message can be a
string or an integer and it is optional
```

Similar to `SyntaxError` and `URIError`, take note that the `filename` and `lineNumber` parameters are non-standard, and they should be avoided if possible.

Using the error console log

Firefox's console log is a tool that is powerful enough for you to log your JavaScript messages. You can log error messages from the built-in objects, or you can write your own messages.

Error messages

What we see in this section are error messages generated that are logged in Firefox's error console log. Before we do that, I need you to open up your Firefox browser, go to **Tools** on the menu bar, and select **Error Console**. Make sure that you do not open any other tabs.

Now, open your code editor, and enter the following code into a new document:

```
<html>
<head>
<script type="text/javascript">

try {
    var anArray = new Array(-1););
}
catch (error) {
    throw error;
}
finally {
    alert("ok, all is done!");
}
</script>
</head>
<body>
</body>
</html>
```

Save the document as a `.html` file, and then run the file on your Firefox browser. Alternatively, you can use the source code found in the `source code` folder with the HTML document entitled: `error-message-console.html`. If you now take a look at your console, you should receive the following error message: **invalid array length**. This is because we have defined an array that is of negative length, which is shown in the code above.

The trick here is to use the `throw` statement to throw error messages. Take note that Firefox's error console does not show the `name` of the error.

Now we will take a look at how to create custom error messages.

Writing your own messages

Let us move on to creating our own error messages. The completed code is found in the `source code` folder, in the file `test-custom.html`.

Once again, open your code editor, create a new document, and enter the following code into it:

```
<html>
<head>
<script type="text/javascript">
function factorial(x) {
    if(x == 0) {
        return 1;
```

```
        }
        else {
            return x * factorial(x-1);
        }
    }
    try {
        var a = prompt("Please enter a positive integer", "");
        if(a < 0){
            throw new Error("Number must be bigger than zero");
        }
        else if(isNaN(a)){
            throw new Error("You must enter a number");
        }
        var f = factorial(a);

        alert(a + "! = " + f);
    }
    catch (error) {
        throw error;
    }
    </script>
    </head>
    <body>
    </body>
    </html>
```

What we have done here is that within the `try` block we have thrown two new `Error` objects, each with a custom message, and then in the `catch` block, we throw the `Error` object again. In the `try` block, we are creating a custom `Error` object, and in the `catch` block, we are throwing the message into the **Error Console**.

Take note of the highlighted lines. We have defined our own messages in the `Error` object. Save the file, and then open up your Firefox browser. Go to **Tools | Error Console**. In the **Error Console**, make sure you are in either the **All** tab or the **Errors** tab. Now run your code in your Firefox browser. You will receive the message **You must enter a number** in your error console if you enter a non-numeric input. If you enter a number that is less than zero, you will receive the message **Number must be bigger than zero**. The key here is to make use of the provided methods and properties to throw your own error messages.

Modifying scripts and testing

Now that we have covered the basic building blocks of throwing and catching errors using built-in objects, and using the console to throw error messages, it is time to learn how we can apply what we have learnt to a simple application.

Time for action – coding, modifying, throwing, and catching errors

I need you to focus and pay attention in this section because we will be applying all that we have learnt previously when we first created an application. After that, we will attempt to generate our own errors and throw various error messages as a part of our testing process.

What we will create is a mock movie booking system. I'm not sure about you, but I've noticed that the folks at the service counter use some form of a movie booking system that has a GUI to facilitate their booking process. Not only will we be creating that, but we will also add more features, such as purchasing food and drinks to go with the movie tickets.

Here are the details of the movie tickets booking system: as you click on each seat, you are executing a booking action. If the seat is booked, a click on it will execute a remove booking action.

Other important design rules are as follows: you cannot buy more meals than the number of tickets that you have booked. For example, if you have booked four tickets, you can only purchase up to four meals, be it a hotdog meal or a popcorn meal. Similarly, for every meal that you have purchased, you can purchase one Sky Walker. This means that if you have purchased three meals, you can only purchase up to three Sky Walkers. Next, you can only pay in hundred dollar notes. This means that you can only enter figures in hundreds for the **Please pay in $100 notes** input field.

In case you are wondering about the pricing of the various merchandise, the tickets are priced at $10 each. The hotdog meal costs $6 while the popcorn meal costs $4. Sky Walker costs $10 each.

Clear about the rules? If you are clear about the rules, we'll first start by creating this application. After that, we will apply the exception catching mechanism as the final step. By the way, the completed code for this example can be found in the folder `cinema-incomplete` of *Chapter 6.*

1. Open up code editor and create a new file. Enter the following code into your file.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>JavaScript Cinema</title>
</head>
<body>
</body>
</html>
```

This will form the backbone of our program. Right now, it will not do anything, nor will it show any design on your webpage. Therefore, we will start by creating the layout of our application.

2. Enter the following code within the `<body>` tag of your HTML document.

```
<div id="container">
    <div id="side-a">
        <h1>Welcome to JavaScript Cinema </h1>
        <div class="screen">
            <p> Screen is located here. </p>
        </div>
        <div class="wrapper" id="tickets">
           <p>You have booked 0 tickets</p>
        </div>
        <div class="wrapper">
           <p>Click on the seats above to make your booking.</p>
        </div>
    </div>
    <div id="side-b">
        <div class="menuRight">
            <h4>Meal Pricing</h4>
            <p>Hotdog Meal : $6 <br />Popcorn Meal : $4</</p>
          <form name="foodForm" onsubmit="return checkForm()">
              <!-- total number of meals cannot exceed total
number of tickets purchased -->
              # of Hotdog Meal ($6/meal): <input type="text"
name="hotdogQty" length="3" size="3px"/>
```

```
                    <br />
                    # of Popcorn Meal ($4/meal): <input type="text"
    name="popcornQty" length="3" size="3px" />
                    <p class="smalltext">Total # of meals cannot
    exceed total number of tickets purchases</p>
                    <br />
                <!-- here's some specials to go with -->
                <p>Here's the special deal of the day:</p>
                Sky Walker($10):<input type="text" name="skywalker"
    length="3" size="3px"/>
                    <p class="smalltext">You can only buy 1 Sky Walker
    for every meal you've purchased.</p>
                    <br />
                    <!-- show total price here -->
                    Please pay in $100 notes
                    <input type="text" name="hundred" length="3"
    size="3px" />
                    <br />
                    <input type="submit" value="Order Now">
            </form>
        </div>
        <div id="orderResults"> </div>
    </div>
</div>
```

This code forms the basic control of our movie ticket booking application. You may have noticed that there are various `div` elements with the class wrapper. These elements will be used to create a grid-like user interface that represents the seats in a cinema. So now we will start to create the grid that will be used to represent the seats.

3. We will first build the first row of the grid. For a start, type the following code within the first `div` element with a wrapper class:

```
<div class="left1" id="a1" name="seats" onclick="checkBooking(this
);">
<p>Available</p>
</div>
<div class="left2" id="a2" name="seats" onclick="checkBooking(this
);">
<p>Available</p>
</div>

<div class="left8" id="a8" name="seats" onclick="checkBooking(this
);">
    <p>Available</p>
```

```
</div>
<div class="left9" id="a9" name="seats" onclick="checkBooking(this
);">
    <p>Available</p>
</div>
```

Notice that each of the `<div>` elements that you have typed within the first `div` element with a wrapper class has a `class` and `id` property. In general, the first `div` will have a class of `left1`, and an ID of `a1`. The next `div` element will have a class of `left2` and an ID of `a2`, and so on. This is the way that we will be designing our grid. Now, let us proceed to the next step.

4. Similar to step 3, we will build the next row of our grid. Enter the following code into the second `div` element with a wrapper class:

```
<div class="left1" id="b1" name="seats" onclick="checkBooking(this
);">
    <p>Available</p>
</div>

<div class="left2" id="b2" name="seats" onclick="checkBooking(this
);">
    <p>Available</p>
</div>


<div class="left8" id="b8" name="seats" onclick="checkBooking(this
);">
    <p>Available</p>
</div>

<div class="left9" id="b9" name="seats" onclick="checkBooking(this
);">
    <p>Available</p>
</div>
```

Notice that the `div` elements that form the second row of the grid have an ID starting with a "b" as opposed to an "a" as is the case in the first row of the grid. This will be the way that we will name and continue to build the grid as we go along. This means that the next row will have an ID beginning with "c", the fourth row will have an ID beginning with "d", and so on.

In all, we will be creating five rows. This means that we have three more rows to go.

5. Now we will build the next three rows of the grid. Type the code given in the previous step into the remaining `div` elements, but remember to change the `id` of each element to suit the row number. At the same time, remember to include the `onclick="checkBooking(this)"`, as this will be used for executing our JavaScript functions.

Once you are done with the HTML, it's time for us to add the CSS in order to create the proper design and layout for our application.

6. For this example, we will be using an external CSS. Therefore, insert the following code after the `<title> </title>` tags.

```
<link rel="stylesheet" type="text/css" href="cinema.css" />
```

7. Now we will create a CSS file. Open up a new document and save it as `cinema.css`, as this is what we referred to in step 6. Next, enter the following code into `cinema.css`:

```css
body{
    border-width: 0px;
    padding: 0px;
    padding-left: 20px;
    margin: 0px;
    font-size: 90%;
}
#container {
  text-align: left;
  margin: 0px auto;
  padding: 0px;
  border:0;
  width: 1040px;
}

#side-a {
  float: left;
  width: 840px;
}

#side-b {
  margin: 0;
  float: left;
  margin-top:100px;
  width: 200px;
  height: 600px;
  background-color: #cccc00;
}
```

This is the code for the CSS classes and ID selectors that are used to build the scaffold of our application. You might want to refresh yourself by going back to *Chapter 1, What is JavaScript Testing,* if you have forgotten how CSS works.

Now, we will decide on the size of the *seats* on the grid, and other important properties.

8. We will define the width, height, background color, and text color of the seats. Append the following code to `cinema.css`:

```
#a1,#a2,#a3,#a4,#a5,#a6,#a7,#a8,#a9,
#b1,#b2,#b3,#b4,#b5,#b6,#b7,#b8,#b9,
#c1,#c2,#c3,#c4,#c5,#c6,#c7,#c8,#c9,
#d1,#d2,#d3,#d4,#d5,#d6,#d7,#d8,#d9,
#e1,#e2,#e3,#e4,#e5,#e6,#e7,#e8,#e9
{
    background:#e5791e;
    color:#000000;
    width: 71px;
    height: 71px;
}
```

The previous code defines the size, color, and background for all of the "seats" in our cinema. Now we are down to the final step in creating the layout and design of our application.

9. We will now define the layout and colors of our grid, which contains our seats. The completed CSS code can be found in the `source code` folder `cinema-incomplete`, in the file `cinema.css`. Append the following code to `cinema.css`:

```
.wrapper{
    position: relative;
    float: left;
    left: 0px;
    width: 840px;
    margin-bottom: 20px;
    background-color: #cccccc
}
…
.left1{
    position: relative;
    float: left;
    left: 10px;
    z-index:0;
}
.left2{
```

```
      position: relative;
      float: left;
      left: 30px;
      width: 71px;
      height: 71px;
}

... ...

.left8{
      position: relative;
      float: left;
      left: 150px;

}
.left9{
      position: relative;
      float: left;
      left: 170px;
}
```

This CSS code basically defines each column of the grid. Once you are done with this, save it as `cinema.css` and `cinema.html`. Make sure that these files are in the same folder. Open up `cinema.html` in your web browser, and you should see something similar to the following screenshot:
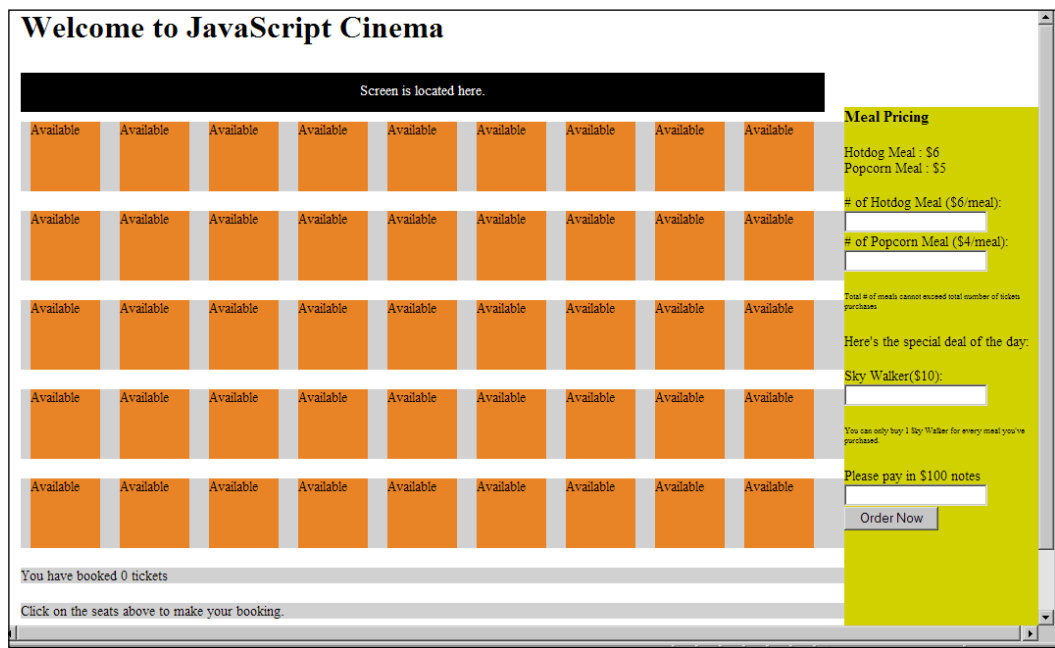
If you see something amiss, you might want to compare your code to the example source code found in the folder `cinema-incomplete`.

Now that we are done with the design and layout of our application, it is time for us to add in the behaviors of the application. The completed code example for the following section can be found in the folder `cinema-complete` of *Chapter 6*.

10. We will be using an external JavaScript file. So let us add the following code snippet before the `</head>` tag:

```
<script type="text/javascript" src="cinema.js"></script>
```

11. Now let us create a new file, and name it `cinema.js`. We will focus on creating the ticket booking mechanism. Because we will be booking tickets by clicking on the seats, we need some mechanism to handle the click event. Because we have already included the `onclick="checkBooking(this)"` in the HTML code, what we need to do now is create a function that handles the click event. Add the following code into `cinema.js`:

```
function checkBooking(element) {
    var id = element.id;
    var status = document.getElementById(id).innerHTML;

    // "<P>Available</P>"  is for an IE quirks
    if(status === "<p>Available</p>" || status === "<P>Available</
P>" )
        addBooking(id);
    else
        removeBooking(id);
    //alert(id);
    return true;
}
```

Notice that the previous code checks for the `innerHTML` of the `div` element and checks to see if it is `<p>Available</p>`. If it is, this means that the seat is available and we can proceed with booking the seat. If not, the seat is booked and a click on the `div` element will result in removing the booking of the seat.

With that in mind, we need two more functions that will help us with the booking and removal of the booking of the seats.

12. We will now create two more functions, to book or to remove booking of the seats. Prepend the following code to `cinema.js`:

```
var counterNumReservations = 0;
function addBooking(id) {
    // add 1 to counterNumReservations when a user clicks on the
seating
    // alert("addBooking");
    document.getElementById(id).style.backgroundColor = "#000000";
    document.getElementById(id).style.color = "#ffffff";
    document.getElementById(id).innerHTML = "<p>Booked!</p>";
    counterNumReservations = counterNumReservations + 1;
    document.getElementById("tickets").innerHTML = "<p>You have
booked " + counterNumReservations + " tickets</p>">";
    return true;
}
function removeBooking(id) {
    // minus 1 from counterNumReservations when a user clicks on a
seating that is already booked
    // alert("removeBooking");
    document.getElementById(id).style.backgroundColor = "#e5791e";
    document.getElementById(id).style.color = "#000000";

    document.getElementById(id).innerHTML = "<p>Available</p>";
    counterNumReservations = counterNumReservations - 1;
        document.getElementById("tickets").innerHTML = "<p>You
have booked " + counterNumReservations + " tickets</p>">";
    return true;
}
```

We have used a global variable to keep track of the number of tickets or seats booked. What the previous functions are doing is that they will increase or decrease (as appropriate) `counterNumReservations` and, at the same time, change the HTML contents of the `div` elements to reflect the status of the booking process. In this case, the seat that is booked will be black in color.

Now, save your file and click on the seats. You should be able to receive visual feedback on the booking process.

We will move on to the form handling mechanism.

13. The form handling mechanism basically handles the following: calculating total spending, the total meal quantity, the amount of money that the user has paid, the change (if any), and also other possible errors or conditions, such as whether enough money is paid, if the money has been paid in hundreds, and so on. With that in mind, we will create the following function:

```
function checkForm(){
    var mealPrice;
    var special;
    var hundred;
    var change;
    var ticketPrice
    if(calculateMealQty() == 1 && checkHundred() == 1 &&
checkSpecial() == 1 && checkMoney() == 1) {
        alert("passed! for checkForm");
        mealPrice = calculateMealPrice();
        special = specialOffer();
        ticketPrice = calculateTicketPrice();
        change = parseInt(amountReceived()) - parseInt((mealPrice
+ special + ticketPrice));
        alert(change);
        success(change);
    }
    else
        alert("there was something wrong with your order.");

    return false;
}
```

In order to create code that is modular, we have split the functionality down into separate functions. For instance, `success()` and `failure()` are used to create the HTML contents, which will show the status of the booking process.

Similarly, notice that we will need to create other functions for calculating meal quantity, checking total money spent, and so on. These functions are created based on what we have learnt from *Chapter 1* to *Chapter 5*, so I'll go on quickly. So now, let us create these functions.

14. We will now create various functions for calculating the meal quantity, the total meal price, the total ticket price, and so on. We'll start with calculating the meal quantity:

```
function calculateMealQty() {
    var total = parseInt(document.foodForm.hotdogQty.value) +
parseInt(document.foodForm.popcornQty.value);
    alert("you have ordered " + total + " meals");
    if(total > counterNumReservations) {
        alert("you have ordered too many meals!");
        failure("you have ordered too many meals!");
        return 0;
    }
    else {
        alert("ok proceed!");
        return 1;
    }
}
```

Now, we'll write the function for calculating the meal price:

```
function calculateMealPrice() {
    // add up total price
    var price = 6*parseInt(document.foodForm.hotdogQty.value) +
(4*parseInt(document.foodForm.popcornQty.value));
    alert("meal price is " + price);
    return price;
}
```

Next is the function for calculating the ticket price:

```
function calculateTicketPrice() {
    var price = counterNumReservations * 10;
    alert("ticket price is " + price);
    return price;
}
```

We'll now write the function for calculating how much was spent on Sky Walker by the user:

```
function specialOffer() {
    // for more ordering offers
    var skywalker = 10 * parseInt(document.foodForm.skywalker.
value);
    alert("skywalker price is " + skywalker);
    return skywalker;
}
```

Once this has been done, we'll write a small function that checks how much money has been received:

```
function amountReceived() {
    var amount = parseInt(document.foodForm.hundred.value);
    alert("I received "+ amount);
    return amount;
}
```

Now that we are done with the functions that do the bulk of the calculations, it's time to write functions to check if the user has ordered too much Sky Walker:

```
function checkSpecial() {
    if(parseInt(document.foodForm.skywalker.value) >
(parseInt(document.foodForm.hotdogQty.value) + parseInt(document.
foodForm.popcornQty.value))){
        alert("you have ordered too many sky walker");
        failure("you have ordered too many sky walker");
        return 0;
    }
    else {
        return 1;
    }
}
```

Once we are done with the previous step, it's time to check if the user paid too little money:

```
function checkMoney() {
    var mealPrice = calculateMealPrice();
    var special = specialOffer();
    var ticketPrice = calculateTicketPrice();
    var change = amountReceived() - (mealPrice + special +
ticketPrice);

    alert("checkMoney :" + change);
    if(change < 0) {
        alert("you have paid too little money!");
        failure("you have paid too little money!");
        return 0;
    }
    else
        return 1;
}
```

As stipulated at the beginning, we will also need to check to see if the user paid in hundred dollar notes. This is done as follows:

```
function checkHundred() {
    // see if notes are in hundreds
    var figure = parseInt(document.foodForm.hundred.value);
    if((figure%100) != 0) {
        alert("You did not pay in hundreds!");
        failure("You did not pay in hundreds!");
        return 0;
    }
    // can use error checking here as well
    else {
        alert("checkHundred proceed");
        return 1;
    }
}
```

Finally, the functions for creating the HTML content that reflects the booking status are as follows:

```
function failure(errorMessage) {

    document.getElementById("orderResults").innerHTML =
errorMessage;
}


function success(change) {
    document.getElementById("orderResults").innerHTML = "Your
order was successful.";
    document.getElementById("orderResults").innerHTML +=
"Your change is " + change + " and you have purchased " +
counterNumReservations + " tickets.";
}
```

Phew! That was quite a bit of coding! You might want to save your files and test your application in your browser. You should have a full working application, assuming that you have entered the code correctly. The completed code up to this stage can be found in the `cinema-complete` folder.

Although we have just been through a tedious process, it was a necessary process. You might ask why you are coding first instead of testing immediately. My answer is that firstly, in the real business world, it is very likely that we need to write code and then test the code that we have written. Secondly, if I were to create a tutorial and ask you to test the code without knowing what the code is, it might leave you hanging on the cliff, as you might not know what to test for. Most importantly, the approach that we have taken allows you to practice your coding skills and understand what the code is about.

This will help you to understand how to apply the `try`, `catch`, and other built-in exceptions object in your code; we will be doing this right now.

15. We will now create a function that will be used to throw and catch our errors by using built-in objects. Now, open `cinema.js` and prepend the following code at the top of the document:

```
function catchError(elementObj) {
    try {
        // some code here
    }
    catch (error) {
        if(error instanceof TypeError){
            alert(error.name);
            alert(error.message);
            return 0;
        }
        else if(error instanceof ReferenceError){
            alert(error.name);
            alert(error.message);
            return 0;
        }
… …
        else if(error instanceof EvalError){
            alert(error.name);
            alert(error.message);
            return 0;
        }
        else {
            alert(error);
            return 0;
        }
    }
    finally {
        alert("ok, all is done!");
    }
}
```

The previous code will form the scaffold of our `catchError()` function. Basically, what this function does is to catch the error (or potential error) and test to see what type of error it is. We will be seeing two sample usages of this function in this example.

The first example is a simple example to show how we can use `catchError()` in other functions so that we can catch any real or potential errors. In the second example, we will throw and catch a `TypeError` by using `catchError()`.

The completed code for this stage can be found in the folder `cinema-error-catching`. Take note that the bulk of the code did not change, except for the addition of the `catchError()` and some minor additions to the `addBooking()` function.

16. We will now try to catch a `ReferenceError` (or `TypeError`, if you are using Internet Explorer) by adding the following code snippet within the `try` block:

```
x = elementObj;
```

Next, add the following code at the top of the function `addBooking()`:

```
var test = catchError((counterNumReservations);
if(test == 0)
    return 0; // stop execution if an error is catched;
```

What we are trying to do here is to stop execution of our JavaScript code if we find any errors. In the above code snippet, we pass a variable, `counterNumReservations`, into `catchError()` as an example.

Now, save the file and test the program. The program should be working normally. However, if you now change the code in the `try` block to:

```
var x =  testing;
```

where testing is not defined, you will receive a `ReferenceError` (if you are using Firefox browser) or `TypeError` (if you are using Internet Explorer) when you execute your application.

The previous simple example shows that you can pass variables into the `catchError()` function to check if it's what you want.

Now, let us move on to something more difficult.

17. We will now try to `throw` and `catch` a `TypeError`. Let us first remove the changes that we made in the previous example. Now what we are doing here is checking to see if the object passed into the `addBooking()` function is the `nodeType` that we want. We can achieve this by adding the following code at the top of the `addBooking()` function:

```
var test = document.getElementById(id);
// alert(test.nodeName); // this returns a DIV -> we use
nodeName as it has more functionality as compared to tagName
var test = catchError(test.nodeType);
// nodeType should return a 1
if(test == 0)
    return 0; // stop execution if an error is catched;
```

Take note of the above lines in the code. What we have done is that we are getting the `nodeType` of the `id` element. The result of this will be used as an argument for the `catchError()` function. For some basic details about `nodeType`, please visit http://www.w3schools.com/htmldom/dom_nodes_info.asp.

Now, remove whatever changes you have done to `catchError()`, and add the following code to the `try` block:

```
    var y = elementObj;
    // var correct is the type of element we need.
    var correct = document.getElementById("a1").nodeType;
alert("Correct nodeType is: " + correct);

var wrong = 9; // 9 represents type Document

 if(y != correct){
     throw new TypeError("This is wrong!");

 }
```

Notice that we are testing for the `nodeType` by checking the resulting integer. Anything that is not correct (the `correct` variable is 1) will result in an error, as shown in the `if` statement block.

Save the file, and then run your example. You should first receive an alert box telling you that the **Correct nodeType is 1**, followed by the message **TypeError**. Next, you will see the message **This is wrong** (which is a personalized message) and finally the message **ok, all is done** indicating the end of the `catchError()` function.

What we have done is that we have thrown our own errors in response to different error types. In our case here, we wanted to make sure that we are passing the correct `nodeType`. If not, it is an error and we can throw our own error.

With that, we'll end this example.

Have a go hero – using catchError function to check input

Now that you have covered quite a bit of code and gained new knowledge, you might want to try this out: use the `catchError()` function to check the user's input for correctness. How would you go about doing that? Here are some ideas to help you get going:

- You might want to make sure that the input values go through `catchError()` before passing them to some other function.
- Will you implement `catchError()` within other functions? Or are the values passed to `catchError()` immediately upon input and then passed to other functions?

Summary

We have covered quite a few concepts in this chapter. The most important is using JavaScript's exception handling mechanisms through the built-in objects, and using these objects together with `try`, `catch`, and `finally` statements. We then tried to apply these concepts into the cinema ticket booking application that we created.

We also learnt the following topics:

◆ Issues that occur when using scripts together, such as name clashing and combining event handlers to make the code more compact

◆ Why we need to use JavaScript libraries, and the issues to consider, such as the availability of documentation, tutorials, plugins, and a testing suite

◆ How we can make use of tools such as Selenium to test GUI and widgets add-ons for libraries (these will be covered in more detail in *Chapter 8*)

◆ How we can write error messages, or our own messages, to the console log

◆ How to perform exception handling by using JavaScript built-in objects and using these together with the `try`, `catch`, and `finally` statements

◆ How to use JavaScript's exception handling mechanisms in a sample application

Up to this chapter, we have been using manual ways to test our code, albeit now using more advanced testing methods. In the next chapter, we will learn how to use different debugging tools to make debugging, which is a part of testing, easier. This will include using tools such as the IE8 Developer Tools, the Firebug extension for Firefox, the Google Chrome Web Browser Inspector, and the JavaScript debugger.

What makes such tools powerful is that they allow us to test in a less obtrusive manner; for instance, there's no need for us to use `alert()`, as we can, in general, write error messages to the built-in consoles of these tools. This is a real time-saver and will make our testing process a lot smoother. We will learn about these different debugging tools in the next chapter. See you there!

7
Debugging Tools

In this chapter, we shall learn about debugging tools that can make our lives easier. We will be using debugging tools provided by major browsers in the market such as Internet Explorer, Firefox, Google Chrome, and Safari.

I understand that there is informative documentation on the Internet, therefore what you can expect in this chapter is that I'll very briefly talk about the features, and then walk through a simple example as to how you can make use of the debugging features to make your life easier.

In general, you will learn about each of the following topics for the above-mentioned debugging tools for each browser:

◆ Where and how to get the debugging tools

◆ How to use the tools to debug HTML, CSS, and JavaScript

◆ Advanced debugging, such as setting breakpoints and watching variables

◆ How to perform profiling by using the debugging tools

So let's get started.

IE 8 Developer Tools (and the developer toolbar plugin for IE6 and 7)

In this section we will focus on Internet Explorer 8's developer toolbar.

> In case you are using Internet Explorer 6 or 7, here's how you can install the developer toolbar for Internet Explorer 6 or 7.
>
> You will need to visit `http://www.microsoft.com/downloads/details.aspx?familyid=e59c3964-672d-4511-bb3e-2d5e1db91038&displaylang=en` and download the developer toolbar. In case you are reading a paper version of this book and cannot copy and paste the above URL, Google "developer toolbar for IE6 or IE7", and you should land on the download page you need.
>
> Note that the toolbar from the above webpage is not compatible with Internet Explorer 8.

If you do not wish to install the developer tool separately, I'd recommend that you install Internet Explorer 8; IE8 comes pre-packaged with their developer tool and it is more handy when compared to installing developer tools for IE6 or IE7 separately.

From this point onwards, I'll be covering the developer tool using the built-in tool in Internet Explorer 8.

Using IE developer tools

Now that we have obtained the plugin, it's time to go through an example to get an idea of how it works. I have prepared a sample code in the `source code` folder of this chapter; go to the folder and open the document called `IE-sample.html` in you browser. Basically what this example does is that it requires you to enter two numbers, and then it will perform addition, subtraction, multiplication, and division on the two numbers. The result will be shown on a box which is found on the right-hand side of the form.

Now give it a test, and once you are done with it, we will start to learn how we can debug this web page using IE8's debugging tool.

Open

I assume that the file is still open in your browser. If not, open `IE-sample.html` in your browser (using Internet Explorer, of course). Once the example is opened, you will need to open the debugging tool. You can navigate to **Tools**, and then click on **Developer Tools**. Alternatively, you can access the debugging tool by pressing *Shift + F12* on your keyboard.

A brief introduction to the user interface

Before we move into the actual debugging process, I'll briefly focus on the key features of the IE debugging tool.
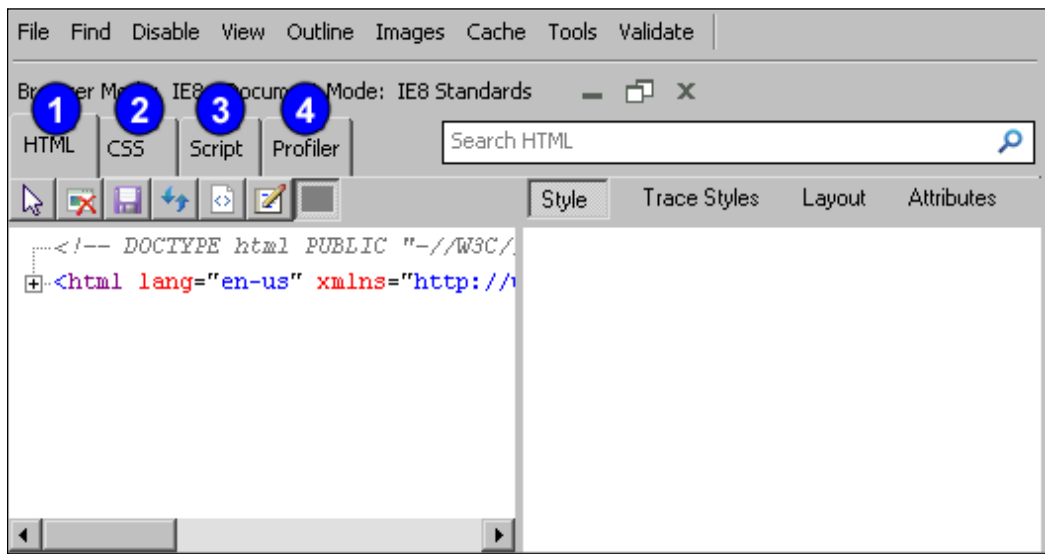
1. HTML: The **HTML** tab shows the source code for the script or web page that you are currently viewing. When you click on this tab, you will get the related tabs on the right-hand side, as shown in the previous screenshot.

2. CSS: The **CSS** tab shows you the CSS stylesheet used by the current webpage that you are viewing.

3. Script: The **Script** tab is where you will be performing your JavaScript debugging tasks. When you click on this tab, you will get a list of features related to the debugging tasks, such as **Console**, **Breakpoints**, **Locals**, and **Watch**.

4. Profiler: The **Profiler** tab shows the profiling data for the web page, should you choose to perform profiling.

Debugging basics of the IE debugging tool

In general, we can use IE's debugging tool in two ways:

- ◆ In a separate window
- ◆ Docking it Docked within the browser

You can dock the debugging tool within the browser by going to the **upper right-hand corner** of the debugging window and clicking on the pin icon. **In my case, I prefer to dock it in my** browser so that I have more viewing space on my screen. Moreover, because the example code is fairly small, docking it on your browser should suffice.

In general, the left-hand side of **the debugging panel is what the IE team calls the Primary Content** pane. This panel displays the web page's Document Object Model; this is the panel that gives us a programmatic overview of the source code of the web page.

Here are some of the basics of debugging when using IE**'s debugging tool.**

Time for action – debugging HTML by using the IE8 developer tool

1. To inspect HTML **elements of the webpage, click on the HTML** tab found in the **Primary Content Panel**. We can click on the **+** icon located on **the first line of the Primary Content Panel**.

2. Once you have clicked on the **+** icon, you should see `<head>` and `<body>` appearing as soon as the `<html>` tag is expanded; clicking on them again will show the other elements contained within the `<head>` and `<body>` tags. For example, let us click on the `div` element with the `id wrap`.

3. On **clicking the** `div` element, **you can immediately see the various properties** associated with `wrap`, such as its parent element, its inherited HTML and CSS, and the CSS **properties that belong to** `wrap`.

   We can perform further inspection by clicking on the various commands found on the **Properties** pane:

   ❑ Style: **The Style** command improves CSS debugging by providing a list of all of the rules that apply to the selected element. The rules are displayed in precedence order; so those that apply last appear at the bottom, and any property overridden by another is struck through, allowing you to quickly understand how CSS rules affect the current element without manually matching selectors. You may quickly turn a CSS rule on or off by toggling the checkbox next to the rule, and the action will take effect immediately on your page. **In our case, you will** see **two inheritances for our** `#wrap` element: body and HTML. You can change the color property to, say, `#eee`, by clicking on the property value **and typing** `#eee`. Once you are done, press *Enter* **and you can** see changes immediately.

❑ Trace **Styles: This command contains the same information as Style** except it groups styles by property. If you are looking for information about a specific property, switch to the **Trace Styles** command. Simply find the property that interests you, click the plus (*+*) icon, and see a list of all rules that set that property—again in precedence order.

❑ Layout: The **Layout** command provides box model information, such as the element's offset, height, and padding. Use this command when debugging an element's positioning.

❑ Attributes: The **Attributes** command allows you to inspect all of the defined attributes of the selected element. This command also allows you to edit, add, or remove the selected element's attributes.

Time for action – debugging CSS by using the IE8 developer tool

Now let use shift our attention back to the **Primary Content Panel**.

1. Click on the **CSS** tab so that we have access to all of the CSS (external or internal) files. Once you have done that, you will see an identical CSS that is used by our webpage.

2. Now I want you to click on a style property, say **color**, found in **BODY**, and change it to **#ccc**. You will immediately see changes to the color of the text in our web page.

What just happened?

We have just performed the basics of debugging, which has provided us with the required knowledge before we move into debugging JavaScript by using IE's debugging tool.

The simple examples that we have carried out above are what we call editing sources on-the-fly; we can edit any HTML or CSS properties without going back to our source code, changing it, saving it, and than reloading the file in our browser. In my opinion, such features are some of the key reasons why we should use debugging tools.

However, take note that the changes that you have made only exist in Internet Explorer's internal representation of the site. This means that refreshing the page or navigating away brings back the original site.

However, there will be cases where you may want to save the changes, and in order to do that, you can click the **Save** button to save the current HTML or CSS to a file. This is done in order to prevent the accidental overwriting of your original source code.

Let us move on to JavaScript.

Debugging JavaScript

Now its time to learn how we can debug JavaScript by using IE's developer tool.

Time for action – more Debugging JavaScript by using the IE8 developer tool

Here are the steps to start debugging:

1. Click on the **Script** tab found in the **Primary Content Panel**.

2. Next, click on the button that says **Start Debugging**.

3. After clicking on **Start Debugging**, you will have all of the functionality of a proper debugger.

If you wish to stop debugging at any point in the debugging process, click on **Stop debugging**.

Now let us see what we can do with the various functionalities of the debugging tools. Let us start with the first one: setting breakpoints.

We usually set breakpoints in order to control execution. In the previous chapters, we have typically relied on `alert()` or other functions in order to control program execution.

However, by using IE's debugging tool, you can control program execution by simply setting breakpoints; you can save quite a lot of `alert()`, or other self-defined functions, along the way.

Now, let us control execution by using breakpoints.
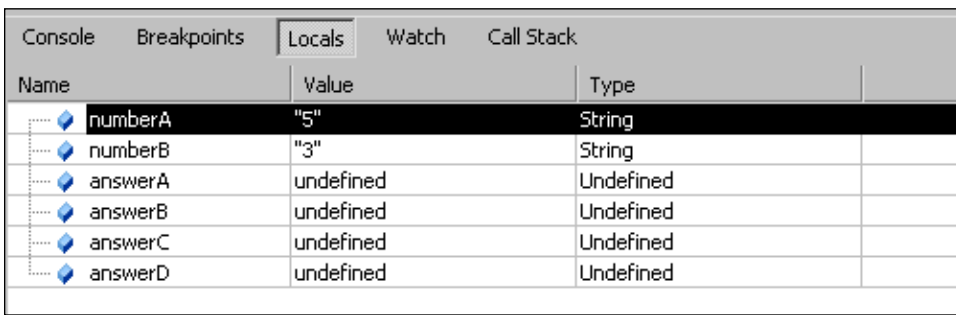
4. You can set a breakpoint by right-clicking on a line number and selecting **Insert Breakpoint**. In our case, let us go to the line that contains `buildContent(answerB, "minus");` right-click on it, and then select **Insert Breakpoint**.

5. Now try running the example by entering some values into the input fields in your browser. You will see that the dynamic content will not be created on the **black square** on the right-hand side. This is because the code execution stops at `buildContent(answerB, "minus");`.

We usually use breakpoints to inspect variables; we need to know if our code is executing the way in which we want it to, in order to make sure that it is correct. So now, let us see how we can set breakpoints and inspect variables.

We inspect variables by using the watch functionality. Continuing from the previous example, we can use the watch functionality by clicking on the **Watch** pane. Alternatively, you can click on **Locals**, which provides a similar functionality and allows us to see a set of variables. This can be done to monitor a custom list of variables, and also to inspect the current state of variables.

To do what we have just described, we need to perform the following steps:

6. Click on **Start Debugging** and set breakpoints for the lines that contain `var answerA = add(numberA, number);` and `buildContent(answerA, "add");`

7. Now, run the example, and type in **5** and **3** respectively for the input fields. Then click on **Submit**.

8. Now go to your **Debugger** panel, and click on **Locals**. You will see the output as shown in the following screenshot:

Name	Value	Type	
numberA	"5"	String	
numberB	"3"	String	
answerA	undefined	Undefined	
answerB	undefined	Undefined	
answerC	undefined	Undefined	
answerD	undefined	Undefined	

What this panel shows is a list of local variables that are local to the function where breakpoints are set

Notice that **answerA**, **answerB**, **answerC**, and **answerD** are currently undefined as we have not performed any calculation for them, because we have set the breakpoint at `var answerA = add(numberA, number);`.

9. Next, click on **Watch**. You can now add the variables that you want to inspect. You can achieve this by typing in the name of the variables. Type in **answerA** and **numberB**, and then press *Enter*. You will see a screen similar to the example shown in the following screenshot:
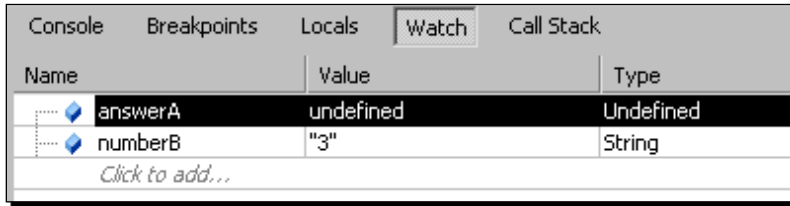
As explained previously, **answerA** is not defined yet as it has not been calculated by our program. Also, because we enter the values for **numberA** and **numberB**, **numberB** is naturally defined.

> Did you notice that we have the incorrect types for our input? This is because we have used the `.value` method to access the values of the input fields. As a good JavaScript programmer, we should be converting the values to floating-point numbers by using `parseFloat()`.

We can continue to execute the code (in debugging mode) by performing Continue, Step In, Step Over, and Step Out operations in the debugging window.

We will move quickly into the example to see how Continue, Step In, Step Over and Step Out work. Continuing from the above example:

10. Click on the **Continue** button, which is green and looks like a "play" button. Immediately, you will see that the code will execute until the next breakpoint. This means that the variables that were previously undefined will now be defined. If you click on **Locals**, you will see output similar to the example shown in the next screenshot:
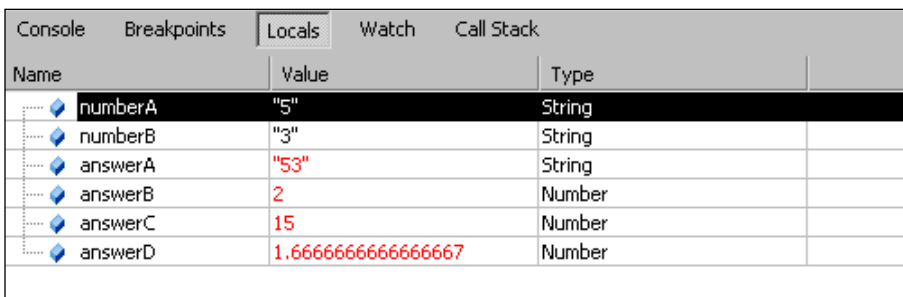
11. Click on **Watch**, and you will see a screen similar to the example displayed in the next screenshot:

This means that the effect of Continue is that it will execute the code from one breakpoint to the next breakpoint. If there is no second breakpoint, the code will execute up to the end.

You might want to experiment with **Step In, Step Over, and Step Out.**

In general, this is what they do:

❑ Step In: This traces the code as the code executes. For instance, you can perform the steps shown in the above example except that you click on **Step In** instead of **Continue**. You will notice that you are effectively tracing the code. Next, you can check on the **Locals** and **Watch** window and you will notice that the previously-undefined variables will be defined as the code progresses.

❑ Step Over: This simply moves to the next line of code without jumping into other functions as with what happened in Step In.

❑ Step Out: This simply "steps out" of the current breakpoint until the next breakpoint. It is similar to **Continue**. If you use Step Out after Step In, it will continue to the next breakpoint (if any).

Now let us move on to the next useful feature, stopping your code when an error is encountered.

To enable this feature, you will need to click on the **Break on Error** button, or you can simply press *Cntrl + Shift + E*. This feature should be automatically enabled once you start debugging.

What this feature does is stop executing the code should any error be discovered. For example, uncomment the line that says: `buildContent(noSuchThing, "add");` and run the code in debugging mode. You will see the following screenshot in the Console, in your debugging window:
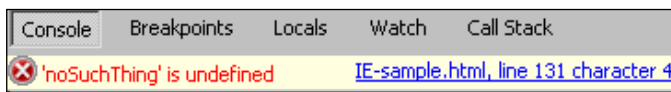
This is one of the cool things about using a debugger; it helps you to spot errors during run time, so that you can quickly identify the errors that you have made.

Now that we have a basic knowledge and understanding of some of the more advanced features of IE's debugging tool, it's time to be concerned about the performance of our JavaScript program.

The Internet Explorer debugging tool comes with a built-in profiler called the JavaScript Profiler, which helps to take your site to the next level by improving its performance.

In general, the profiler gives you data on the amount of time spent in each of your site's JavaScript methods and even built-in JavaScript functions. Here's how you can use this feature.

12. Using the sample example source code in your browser, open the **Develop** tool and click on the **Profile** tab. Then click on **Start Profiling**, to begin a session.

13. Go to your browser, and enter some sample values. For instance, I entered **5** and **3**. Once you have entered the sample values, go to your debugging window and click on **Stop Profiling**. A screen similar to the one shown in the following screenshot will be displayed:

Notice that the Jscript Profiler includes the time spent on each of the functions (the name of each function is also given). The number of times that each function is being used is also given, as shown in the **Count** column. You may have noticed that the time taken for each of our functions is **0.00**; this is because our example program is relatively small, so the time required is close to zero.

What just happened?

We have just covered Internet Explorer's developer tool, which helps us to perform debugging tasks in a much streamlined manner.

In case want to know what the difference between debugging manually and using a debugging tool is, I can safely tell you from experience that the amount of time saved by using a debugging tool alone is a good enough reason for us to use debugging tools.

You may understand that there are various quirks involved when developing for Internet Explorer; using its built-in debugging tools will help you to figure out these quirks in a more efficient manner.

With that in mind, let us move on to the next tool.

Safari or Google Chrome Web Inspector and JavaScript Debugger

In this section, we will learn about the JavaScript debugger used in Safari and Google Chrome. Both browsers have similar code base, but have subtle differences, so let us start by learning about the differences between Safari and Google Chrome.

Differences between Safari and Google Chrome

If you are an Apple fan, you will no doubt feel that Safari is perhaps the best browser on planet Earth. Nonetheless, both Google Chrome and Safari have their roots in an open source project called WebKit.

Safari and Google Chrome use a different JavaScript Engine. Since Safari 4.0, Safari has used a new JavaScript engine called SquirrelFish. Google Chrome uses the V8 JavaScript Engine.

However, in terms of JavaScript debugging, the two are almost identical when we are using the built-in debugger provided by Google Chrome and Safari; even the interface is similar.

In the following sections, I'll be using Chrome to explain the examples.

Debugging using Chrome

For Google Chrome, there is no need to download any external tools in order for us to perform debugging tasks. The debugging tools are delivered right out the box with the browser itself. So now, we will see how we can start our debugging session, using `sample.html`.

Opening and Enabling: We'll start by opening and enabling debugging in Chrome. There are basically two tools in Google Chrome that you can use to help you to perform debugging tasks for your web applications: the web inspector and the javascript debugger.

Web Inspector: Google Chrome's Web Inspector's predominant use is for inspecting your HTML and CSS elements. To use Web Inspector, right-click on any component on a web page to launch the Web Inspector. You'll be able to see the elements and resources associated with the component on which you clicked, including a hierarchy view of the DOM and a JavaScript console. To use the Web Inspector, open `example.html` in Google Chrome. Move your mouse to the side bar column that says **Column 2**. Right-click on **Column 2** and you will see a pop-up menu. Select **Inspect Element**. A new window is opened. This is the Web Inspector.

Now we'll move on to the JavaScript debugger.

JavaScript Debugger: To use Chrome's JavaScript Debugger, select the **Page menu** icon, which can be found on the right-hand side of the URL input field, and then go to **Developer | Debug JavaScript Console**. You can also press *Ctrl + Shift + J* to launch JavaScript Debugger. If you are using Safari, you will have to first enable the developer menu by clicking on the **Display Settings** icon that is found on the right-hand side of the **Page** icon, select **Preference**, and then go to **Advanced**. On this screen, enable the option **Show Develop menu in menu bar**. Then you can access this menu bar by clicking on the **Page** icon and going to **Develop** and selecting **Start Debugging JavaScript**. The interface is almost identical as to what we see in Google Chrome.

Notice that by opening the JavaScript Debugger, you will be opening up the same window that you saw in the Web Inspector. However, the default tab is now Scripts. In this tab, you can to view the source code of our example mentioned in the previous subsection.

This is the main screen that we will be using to perform our debugging tasks. In the following sessions, we will start to get our hands a little dirty by doing some basic debugging.

Most of the tasks and actions that you are going in order to perform should be conceptually similar if you have gone through our debugging session on using the Internet Explorer developer tools.

We have just explored the basic actions of opening and starting the Web Inspector and the JavaScript Debugger. Let us now go through a brief introduction to the user interface, in order to get you up to speed.

A brief introduction to the user interface

Here's a brief explanation of where you can find the key features in Google Chrome's debugging tool as shown in the following screenshot:
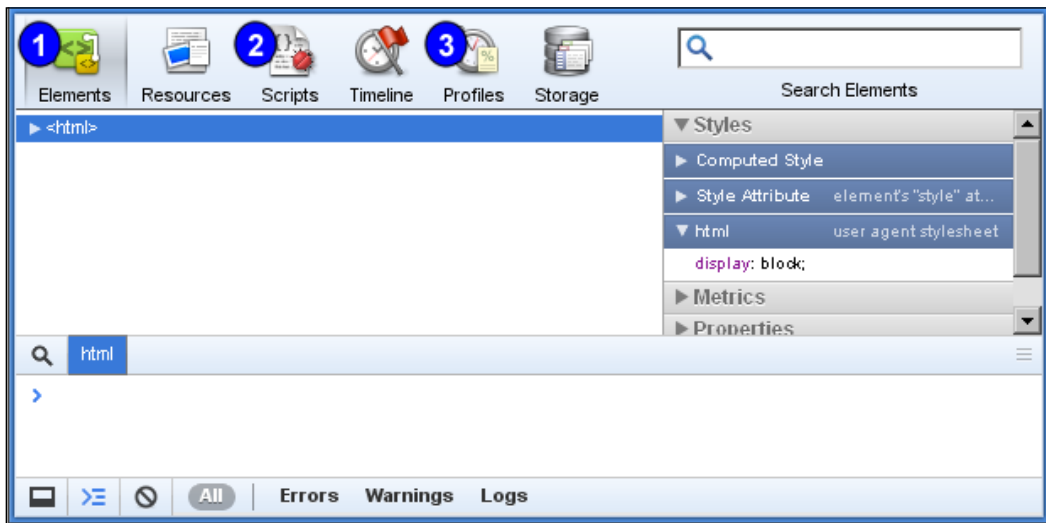
1. Elements: The **Elements** tab shows you the source code for the script or web page that you are currently displaying. When you click on the **Elements** icon, you will get the related tabs (as shown on the right-hand side of the previous screenshot), such as **Computed Style**.

2. Scripts: The **Scripts** tab is where you will perform your JavaScript debugging tasks. When you click on the **Scripts** icon, you will get a list of related features for debugging, such as **Watch Expressions**, **Call Stack**, **Scope Variables** and **Break**.

3. Profiles: The **Profiles** tab shows the profiling data of your web page, should you choose to perform profiling.

Time for action – debugging with Chrome

1. We'll now learn how to use the console and make use of breakpoints in order to simplify our debugging session. We'll start with the console.

2. The console basically shows what you have done within a debugging session. We first see how we can access the console.

3. Start off by opening the file `sample.html` in your Google Chrome browser, if you have not done so already. Once you have done that, perform the following steps in order to show the console:

4. Open your JavaScript debugger by selecting the **Page menu** icon which can be found **on the right-hand side of the URL** input field, and then go to **Developer | Debug JavaScript**. You can also press *Ctrl + Shift + J* to launch JavaScript Debugger.

5. Once you have completed step 4, click on the console icon, which can be found at the bottom of the JavaScript debugger. Once you are done, you will see a screen similar to the example shown in the following screenshot:
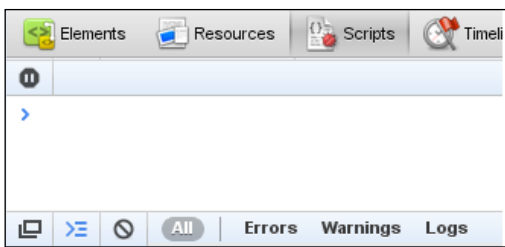
Now that we have opened the console, we move on to the most commonly-used features of the debugger. Along the way, you will also see how the console logs our actions.

We'll now move on to breakpoints by learning how to set them.

As noted earlier, setting breakpoints is an important part of the debugging process. So we will start off our actual debugging process by setting a breakpoint.

6. With `sample.html` opened in Google Chrome, start your debugger and make sure that you are in the **Scripts** tab. You can set a breakpoint by clicking on the line number at which we want to set our breakpoint. Let use try going to the line that contains `buildContent(answerB, "minus");` and click on the line number. You will see a screen similar to the example shown in the following screenshot:
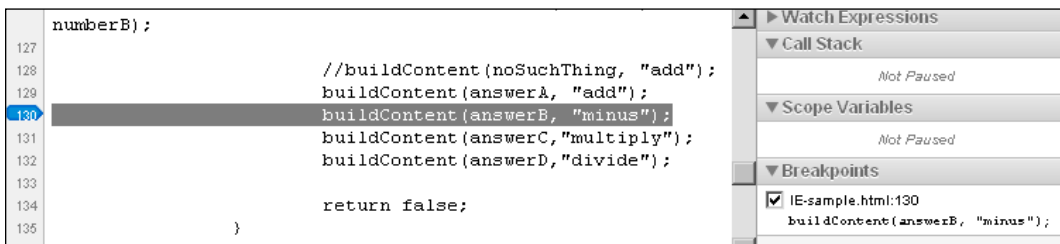
Notice that **line 130** now has a blue arrow (highlighted line), and over to the right of the source code panel, you will see the **Breakpoint** panel. This now contains the breakpoint, which we have just set, within it.

7. Run the example and enter some values into the input fields in your browser. I want you to enter **4** in the first input field and **3** in the second input field. Then click on **Submit**. You will see that the dynamic content will not be created in the black square on the right. This is because the code has stopped at `buildContent(answerB, "minus");`.

8. Now go back to your debugger, and you will see the next screenshot on the right-hand side of your source code, similar to the example shown below:
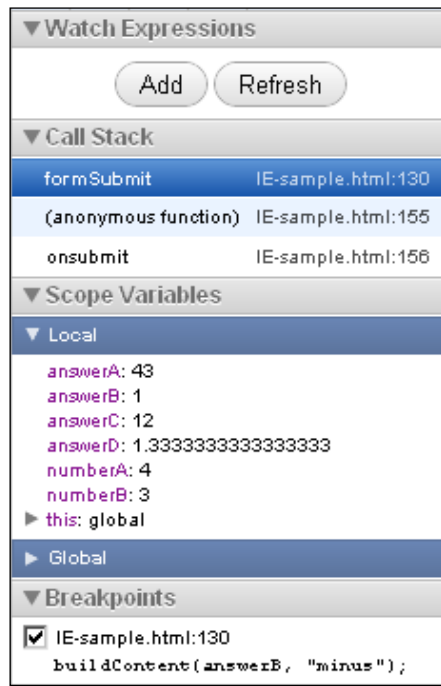
You will see that **Call Stack**, and **Scope variables** are now being populated with values, while **Watch Expressions** is not. We will cover these in detail in the next few paragraphs. But for now, we first start off with **Call Stack** and **Scope Variables**.

As shown in the previous screenshot, **Call Stack** and **Scope Variables** are now populated with values when we execute the program. In general, **Call Stack** contains the sequence of functions that are being executed, and **Scope Variables** shows the values of the variables that are available until a breakpoint or end of execution.

Here's what happens when we click on the **Submit** button: the first function that is executed is `formSubmit()`, and within this function, var `answerA`, `var answerB`, `var answerC`, and `var answerD` are calculated. This is how the **Scope Variables** get populated with our values.

In general, that is how **Call Stack** and **Scope Variables** work in Google Chrome. Now, let us focus on a feature that has been lingering in our minds, **Watch Expression**.

Before we explain what **Watch Expression** is, it is best that we see it in action, first. Going back to the previous screenshot, you will notice that **Watch Expression** is not populated at this point of time. We'll now try to populate **Watch Expression** by performing the following steps:

9. Refresh your browser and go back to your debugger.

10. In the **Watch Expression** panel, click on **Add**, and enter the following: `document.sampleform.firstnumber.value` and `document.getElementById("dynamic")`.

11. Go back to your browser and enter **4** and **3** for the input values. Click on **Submit**. Assuming that you have not removed the breakpoint that we set in the previous subsection, you will see the information shown in the next screenshot in the **Watch Expression** panel:
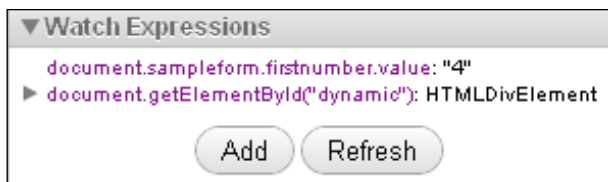
Watch Expression is now populated. `document.sampleform.firstnumber.value` and `document.getElementById("dynamic")` are lines of code copied from our JavaScript program. If you were to trace the code, you would notice that `document.sampleform.firstnumber.value` is used to derive the value of the first input field, and `document.getElementById("dynamic")` is used to refer to the `div` element.

Up to this point, you will have understood that **Watch Expression** is useful for checking out expressions. All you have to do is add the expression that you want to see, and, after executing the program, you will see what that expression means, refers to, or what current values it has. This allows you to watch the expressions update as the program executes. You do not have to complete the program to see the value of the variables.

Now it's time to move on to the Continue, Step In, Step Over, and Step Out operations in the debugging window.

The concepts here are pretty similar to what we have seen in Internet Explorer developer tools. In case you are wondering where the buttons are for executing these operations, you can find them above the **Watch Expression** panel. Here are the related concepts for each of the operation:

- Step In: This traces the code as the code executes. Assuming that you are still at our example, you can click on the icon with an arrow pointing downwards. You will see that you are effectively tracing the code. As you continue to click on **Step In**, you will see the values in **Scope Variables** and **Call Stack** change. This is because at different points of the code there will be different values for various variables or expressions.

- Step Out: This simply moves to the next line of code without jumping into other functions, similarly to how Step In works.

- Step Over: This simply moves to the next line of code.

In this last subsection, we will focus on how we can pause on exceptions. In general what this means is that the program will halt at the line where a problem is encountered. Here's what we will do to see it in action:

12. Open `sample.html` in your editor. Search for the line that says `buildContent (noSuchThing, "add");` and uncomment it. Save the file and open it in Google Chrome.

13. Open the debugger. Click on the button with a Pause sign ⏸, which can be found to the right of the **Show Console** button. This will cause the debugger to halt execution when errors are encountered.

14. As usual, enter some values for the input fields. Click on **Submit**. Once you have done so, go back to your debugger, and you will see the information shown in the following screenshot:


```
buildContent(noSuchThing, "add");
```
❌ [Exception] ReferenceError: noSuchThing is not defined
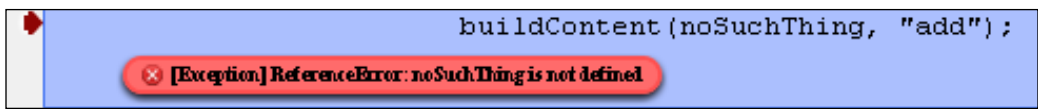
In general, this is the kind of visual message that you can get if you enable the pause on exception feature.

What just happened?

We have covered the basics of using Google Chrome. If you have followed the **previous** tutorial, you will have learned how to use the **Console, setting, stepping in, stepping out** and over **a breakpoint, pausing on exceptions, and watching the variables.**

By using a mix of **the above features, you will be able to quickly sniff out and spot unintended** JavaScript errors. You can even trace how your JavaScript code is working as it executes.

Over the next few sections, you will begin to notice that most of the tools have very similar features, although some may have different terms for the same feature.

Now it**'s time to move on to the other tool, the Opera JavaScript Debugger.**

Opera JavaScript Debugger (Dragonfly)

Opera**'s** JavaScript **Debugger is called Dragonfly. In order to use it, all you need to do** is download the latest version of Opera; Dragon**fly is included in the latest version of** Opera already.

Now that you have installed the **necessary software, it is time for us to perform** debugging tasks.

Using Dragonfly

We**'ll** first start with our `example.html` file. Open this file in Opera **browser. Now we will** see how we can start Dragon**fly.**

Starting Dragonfly

To access Dragonfly, go to **menu** option **Tools**. Select **Advanced**, and then click on **Developer Tools**. Once you have done that, Dragon**fly will appear. As usual, we'll start with a brief** introduction to the user interface of the tool.

Brief Introduction to the User Interface

Here**'s a** brief overview of the most important functions that we will be using, **as shown in** the next screenshot:
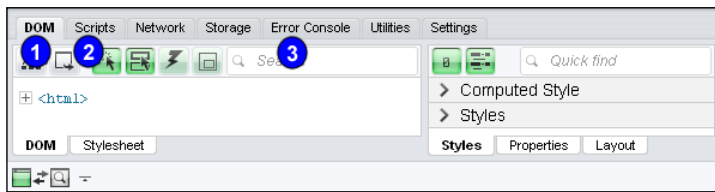
1. **DOM**: This tab is used for checking the HTML and CSS elements

2. **Scripts**: This tab is used when we are debugging JavaScript

3. **Error Console**: This tab shows the various error messages when we are debugging JavaScript.

We'll now start with debugging `example.html`.

Time for action – debugging with Opera Dragonfly

1. In this section, we'll learn how to use the debugging facilities of the Dragonfly. We'll start by setting breakpoints.

Here's how we can set a breakpoint in Dragonfly:

2. With `sample.html` opened in Opera, start Dragonfly and click on the Scripts tabbed page. You can set a breakpoint by clicking on the line number at which we want to set our breakpoint. Let us try going to the line that contains `buildContent(answerB, "minus");` and then clicking on the line number.

3. Go to your browser and execute `example.html`. Enter **5** and **3** as the inputs. Click on **Submit**. As usual, you will not see any content being created dynamically. The program's breakpoint is at contains `buildContent(answerB, "minus");`.

4. Now go back to Dragonfly, and you will notice that the panels for **Call Stack** and **Inspection** are now populated. You should see similar values to those shown in the next screenshot if you enter the same values as I did:
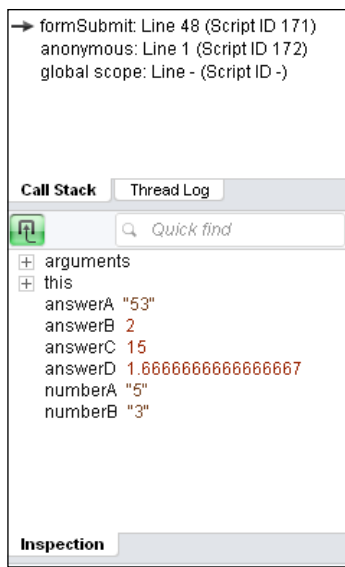
The values shown in **Inspection** and **Call Stack** are the values and functions that have been calculated and executed up to the breakpoint.

What just happened?

We have just used Dragonfly to set a breakpoint, and as we executed our JavaScript program, we have seen how Dragonfly's various fields get populated. We'll now go into detail with regards to each field.

Inspection and Call Stack

As shown in the previous screenshot, **Call Stack** and **Inspection** are populated with values when we execute the program. In general, **Call Stack** shows the nature of the runtime environment at the time of a specific function call–what has been called, and in what order. The inspection panel lists all of the property values and others for the current call. Stack frames are specific parts of the **Call Stack**. **Inspection** is conceptually similar to the **Scope Variables** seen in Google Chrome.

Thread Log

The **Thread Log** panel shows the details of the different threads running through the script that you are currently debugging.
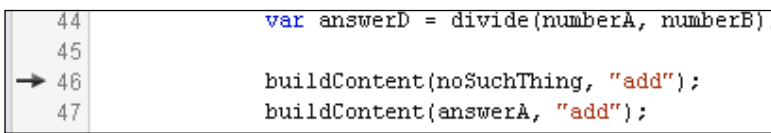
We'll now move on into greater details about the functionalities of Dragonfly.

Continue, Step Into, Step Over, Step Out, and Stop at Error

We can also perform the usual Continue, Step Into, Step Over, and Step Out tasks while debugging our code. Here's a screenshot that shows us where to find the previously-mentioned functions:

1. Continue: This continues the currently-selected script after it has stopped at a breakpoint. This will continue to the next breakpoint, if any, or it will continue to the end of the script.

2. Step Into: This allows you to step into the next function in the stack, after the current function within which the breakpoint is contained. It effectively traces the code as the code executes. Assuming that you are still at our example, you can click on the **Step Into** icon with an arrow pointing downwards. You will see that you are effectively tracing the code. As you continue to click on **Step In**, you will see the values in **Inspection** and **Call Stack** change. This is because at different points of the code there will be different values for various variables or expressions.

3. Step over: This allows you to step to the next line after the line on which the breakpoint is set—you can use this multiple times to follow the execution path of the script.

4. Step out: This causes you to step out of the function.

5. Stop at error: This allows you to stop executing your script at the point where an error is encountered. To see this in action, open the file `example.html` in your editor, and search for the line that says `buildContent(noSuchThing, "add");` and uncomment it. Save the file and then open it again, using Opera. Go to Dragonfly and click on the icon. Now execute your program in Opera and enter some sample values. Once you are done, you will see the following screenshot in Dragonfly:

Notice that at line **46** there is a black arrow pointing to the right. This means that there is an error in this line of code.

Before we end of the section on DragonFly, we'll take a look at one more important feature the settings feature.

Settings

Opera's Dragonfly has a nifty feature that allows us to create different settings for our debugging tasks. There is a whole list of these settings, so I will not go through all of them. But I will focus on those that are useful for your debugging sessions.

◆ Scripts: **In this panel, enabling reload documents automatically when selecting** window is a huge time saver when you have multiple JavaScript files to debug, because it will help **you to automatically reload the documents.**

◆ Console: **This panel allows you to control what information you wish to see during** your debugging session. From XML to HTML, **you can enable or disable messages in** order to see the most important information.

With that, we'll end the section on Dragonfly and move on to Firefox and the Venkman Extension.

Firefox and the Venkman extension

We know that Firefox has many plugins and tools, some of which are made for web development purposes. In this section, we will learn about the Venkman extension, which is Mozilla's JavaScript Debugger.

Using Firefox's Venkman extension

We'll start off by obtaining the extension; we will assume that you have Firefox installed. In my case, I am using Firefox 3.6.3.

Obtaining the Venkman JavaScript Debugger extension

To obtain the Venkman JavaScript Debugger extension, go to `https://addons.mozilla.org/en-US/Firefox/addon/216/` **and click on Add To Firefox**. Once it is installed, Firefox will prompt you to restart Firefox for **the changes to take effect.**

Opening Venkman

To start debugging, let us open the file `example.html` in Firefox. Here, we can now start Venkman. **Go to Tools** and select **JavaScript Debugger**. If you are using older versions of Firefox, you can access it by going to **Tools | Web Development | JavaScript Debugger menu**.

Now we'll start with a brief introduction **to Venkman's user interface.**

A brief introduction to the user interface

The next screenshot shows the user interface of the Venkman extension:
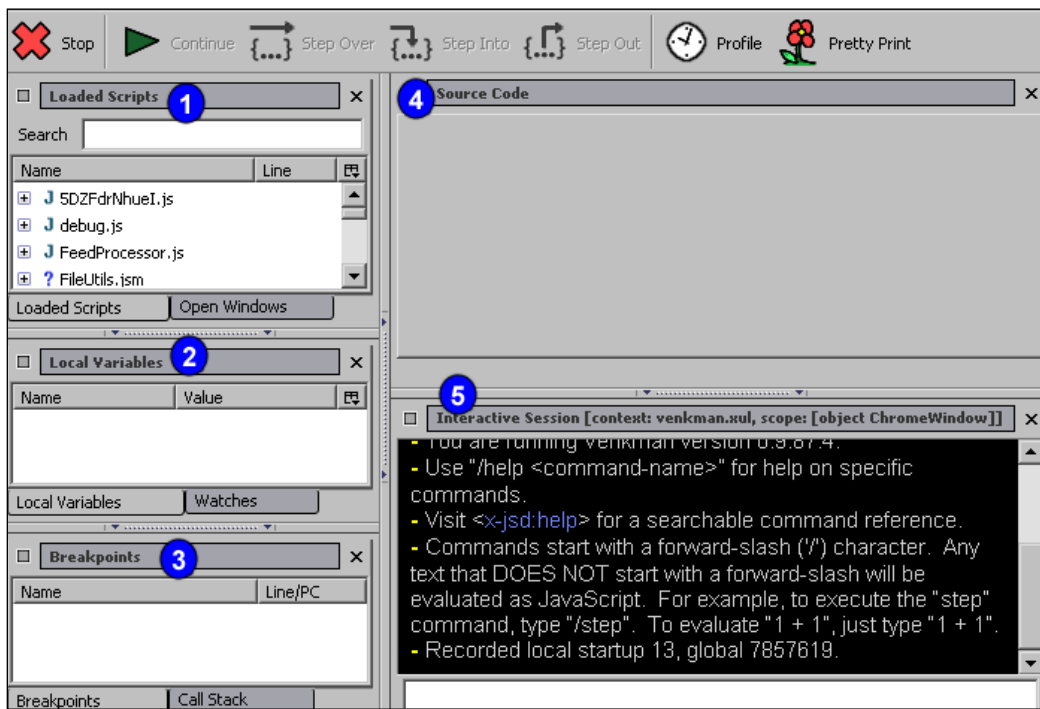
1. Loaded Scripts: **The Loaded Scripts** panel shows a list of scripts that you may load for debugging. After you have loaded a script, you will see it in the **Source Code** panel.

2. Local Variables and Watches: **The Local Variables** panel shows the local variables that are available when you are performing debugging tasks. If you click on the **Watches** tab, you will see the **Watches** panel. You will be using this to enter the expressions that you want to watch.

3. Breakpoint and Call Stack: **The Breakpoint** panel allows you to add a list of breakpoints, and the **Call Stack** panel shows a list of functions or variables that are executed, in order.

4. Source Code: **The Source Code** panel shows the source code that you are currently debugging.

5. Interactive Session: **The Interactive Session** panel is the console for this debugger.

We'll now start debugging using the Venkman extension:

Time for action – debugging using Firefox's Venkman extension

We'll start off by setting breakpoints, before we go into greater details:

As with all debuggers, we can set a breakpoint by performing the following steps:

1. Start by opening the file `example.html`, in Firefox.

2. Open the JavaScript debugger, and the debugger window will be shown.

3. When your see the debugger window, go to the **Load Scripts** panel and you will see the file `example.html` in it. Click on it and you will see the code being loaded in the **Source Code** panel.

4. To set a breakpoint, click on the line at which you want the breakpoint to be set. For instance, I have set it on line 130, which contains the code: `buildContent(answer, "minus");`. You should see something like the following screenshot:
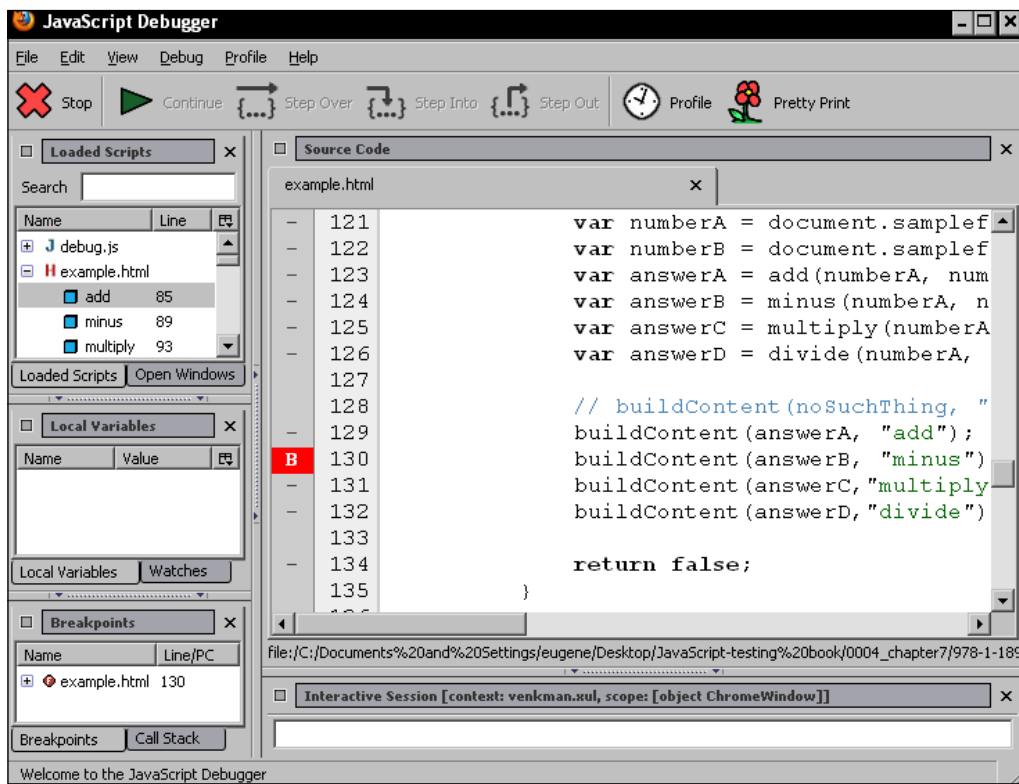
What just happened?

The first thing to note is that there is a **white B** within a red **rectangle, as shown in the** previous **screenshot. This indicates that a breakpoint has been set.**

In Venkman, there are times where you will see a **white F** within a **yellow box; this means** that Venkman could only set a Future Breakpoint. This happens when the line you select has no source code, or if the line of code has already been unloaded by the JavaScript engine (top level code is sometimes unloaded shortly after it completes execution).

A Future Breakpoint means that Venkman was unable to set a hard breakpoint now, but if the file is loaded later, and it has executable code at the selected line number, Venkman will automatically set a hard breakpoint.

The second thing to note is the **Breakpoints** panel. This contains a list of all of **the** breakpoints that we have set in this **debugging session.**

Now, before we move on to the following **subsections, I need you to enter some input for** our example application by going to your browser. In my case I have entered **5** and **3** for **the** first and second input fields respectively. Once you have done that, click on **Submit**.

Again, you will notice that the panels that were originally empty are now populated with values. We will cover this in the following subsections.

Breakpoints or Call Stack

We have **briefly covered breakpoints in the previous subsection. If you look at the** **Breakpoints** panel, you will notice that in that panel, there is another tab, to the right-**hand side of the Breakpoint** panel, called **Call Stack**.

Click on **Call Stack** and you should see some data in this new panel. Assuming that you have entered the same input and the same breakpoint, you **will see a screen similar to the** example shown in the **next screenshot:**
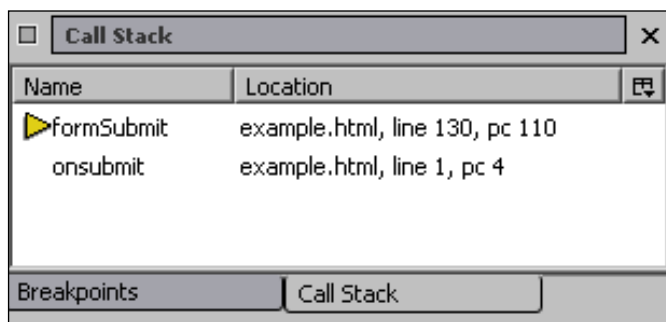
In general, **Call Stack** shows the nature of the runtime environment at the time of a specific function call–what has been called, and in what order. In Venkman, it shows the name of the function, filename, line number and pc (program counter).

Local Variables and Watches

Let us now focus on **Local Variables** and **Watches**. The panels for **Local Variables** and **Watches** are located above the **Breakpoints** and **Call Stack** panels. And if you have been following my instructions up to this point with the exact same input, you should see the following in the **Local Variables** panel:
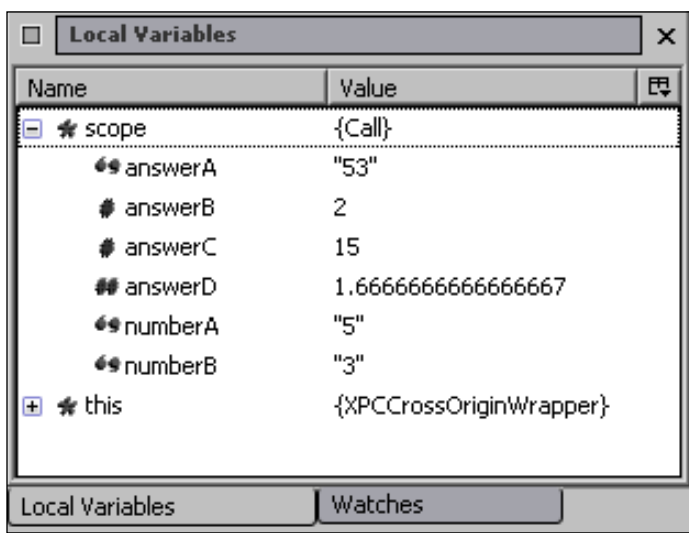
The **Local Variables** panel simply shows the values of the variables that have values (due to code execution) up to a breakpoint, or to the end of the program, according to the order in which they are created or calculated.

The next panel that we want to talk about is the **Watches** panel. The **Watches** panel does the same thing as watch expressions, as we have done previously for other browsers. However, because we have not added anything for the **Watches** panel yet, let us take some action to see how the **Watches** panel works:

Time for action – more debugging with the Venkman extension

In this section, we'll cover more debugging features such as the Watch, Stop, Continue, Step Into, Step Over, Step Out, edge triggers and throw triggers. But first, let us perform the following steps, in order to see the Watch panel in action:

1. Click on the **Watches** tab.

2. Right-click within the **Watches** panel, right-click and select **Add Watch**.

3. Enter `document.sampleform.firstnumber.value`.

4. Repeat steps 2 and 3, and this time enter `document.getElementById("dynamic")`.

    Once you are done, you will see the output shown in the following screenshot:
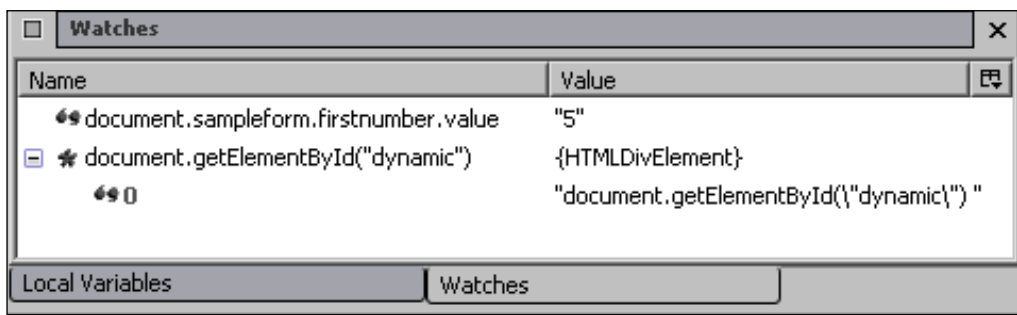
    What the **Watches** panel does is allow us to add a list of expressions that we want to keep track of, and also shows us the value of the expression.

    Now let's move on to the Stop and Continue features.

    Venkman provides some useful functionality, which includes Stop and Continue. Stop basically stops at the next JavaScript statement, and Continue continues the code execution.

    You can make Venkman stop at the next JavaScript statement.

5. Click on the large red **X** on the toolbar, or you can go to the menu and select **Debug** and then choose **Stop**.

    There are times when no JavaScript is being executed. If this is the case, you will see an ellipsis (...) appear over the **X** in the toolbar, and the menu item will be checked. When the next line of JavaScript is encountered, the debugger will stop. You can cancel this by clicking on **X** or selecting **Stop** again.

In addition to Stop and Continue, Venkman also provides the standard Step In, Step Over, and Step Out features.

❑ Step In: This executes a single line of JavaScript, and then stops. You can try this by clicking on the icon that says **Step Into**. If you click on it multiple times, you will notice that the local variables change and you will get to see that the code is being executed as if you are tracing the code.

❑ Step Over: This is used to step over an impending function call, and return control to the debugger when the call returns. If you click on **Step Over**, you will see that new content is being created in your browser. For the file `example.html`, assuming that you click on **Step Over** from the breakpoint, you will see content being created from `buildContent(answer, "minus");`.

❑ Step Out: This executes until the current function call exits.

We'll now see how we can make use of Error triggers and Throw triggers.

Error triggers is used to make Venkman stop at the next error, and Throw Triggers is used to make Venkman stop when the next exception is thrown.

To see it in action, we'll perform the following actions:

6. Open the file `example.html` in your editor and, once again, search for the line that says `buildContent(noSuchThing, "add");` and uncomment it. Save the file and open it again, using Firefox.

7. After you have opened the file in Firefox, open Venkman.

8. Once you have opened up Venkman, go to **Debug | Error Trigger** and select **Stop for Errors**. Then, once again, go back to **Debug | Throw Trigger** and select **Stop for Errors**.

9. Go to your browser and enter any two numbers for the input fields—say **5** and **3** respectively. Click on **Submit**.

10. Return to Venkman and you will see that the line with `buildContent(noSuchThing, "add");` is highlighted, and within the Interactive Session (or console) panel, you will see an error message that says **X Error. noSuchThing not defined**.

Now that we have seen how Venkman can be used to stop our program when errors are encountered, let us move on to its profiling feature.

As we have mentioned in the previous chapters, profiling is used to measure execution times for your scripts. To enable profiling:

11. Click on the **Profile** button in the toolbar. When profiling is enabled, you will see a green check mark on the toolbar button.

12. Once you have Profiling enabled, go to your browser and enter some sample values. I'll stick to **5** and **3** again. Then click on **Submit**.

13. Go back to Venkman, go to **File**, and select **Save Profile Data As**. I have included an example as to what we have just done, and saved it to `data.txt` file. You can open the file and see the contents of the profiling session. You can find the profiling data for the file `sample.html` by searching for `example.html` in the file `data.txt`.

14. When you are done, click on **Profile** again to stop collecting the data.

While profiling is enabled, Venkman will collect call count, maximum call duration, minimum call duration, and total call duration, for every function called.

You can also clear the profile data for the selected scripts by using the **Clear Profile Data** menu item.

What just happened?

We have gone through the various features of the Venkman extensions. Features like Stop, Continue, Step In, Step Out and Over of breakpoints shouldn't be unfamiliar to you by this stage, as they are conceptually similar to the tools that we introduced earlier.

So let us now move to the last and final tool, the Firebug extension.

Firefox and the Firebug extension

I personally think that the Firebug extension needs no further introduction. It is probably one of the most (if not most) popular debugging tools for Firefox in the market right now. Firebug is free and open source.

It has the following features:

- Inspection and editing HTML by pointing and clicking on your web page
- Debugging and profiling JavaScript
- Quickly spotting JavaScript errors
- Logging JavaScript
- Executing JavaScript on the fly

Firebug is perhaps one of the best documented debugging tools on the Internet. So we'll have a look at the URLs that you can visit in order to take advantage of this free, open source, and powerful debugging tool:

- To install Firebug, visit: `http://getFirebug.com`

- To see a complete list of FAQ, visit: `http://getFirebug.com/wiki/index.php/FAQ`

- To see a full list of tutorials, visit: `http://getFirebug.com/wiki/index.php/Main_Page`. If you wish to learn more about each specific feature, look for **Panel** on the left-hand side of the web page.

Summary

We have finally reached the end of this chapter. We have covered specific tools for various browsers that can be used for our debugging tasks.

Specifically, we have covered the following topics:

- The Developer tool for Internet Explorer
- JavaScript Debugger and Web Inspector for Google Chrome and Safari
- Dragonfly for Opera
- The Venkman extension for Firefox
- Resources for Firebug

In case you need more information about each specific tool, you can Google it by appending the keyword "tutorial" to each of the tools and features' mentioned in this chapter.

We have covered the most important features of the tools that can help you get started with debugging your JavaScript application. In our final chapter, we will focus on the various testing tools that you can use when your testing requirements cannot be met manually.

8
Testing Tools

In the final chapter, we will cover some advanced tools that you can use for testing your JavaScript. We will be covering tools that can help you further to automate your testing and debugging tasks and, at the same time, show you how you can test your user interface.

I understand that you are spoilt for choice as there are many tools out there for you to choose from when carrying out testing tasks. But what I will focus on are tools that are generally free, cross-browser and cross-platform; whether you are a fan of Safari, IE, Chrome or other browsers doesn't really matter. Based on `http://w3schools.com/browsers/browsers_stats.asp`*, approximately 30% of web browsers use Internet Explorer, 46% use the Firefox browser, and the remainder of them use Chrome, Safari, or Opera. This means that the tools that you use will cater to these statistics. Although there are applications that were developed specifically for only one browser, it is a good practice and learning experience for us to learn how to write code for use in different browsers.*

More importantly, the tools that I am going to cover in great detail are those that I personally feel are easier to get started with; and this will help you to get a feel of the testing tools in general.

The following tools will be covered in detail:

◆ Sahi, a cross-browser automated testing tool. We'll use this to perform UI testing.

◆ QUnit, a JavaScript testing suite, which can be used to test just about any JavaScript code. We'll use this to perform automated testing of JavaScript code.

◆ JSLitmus, a lightweight tool for creating ad hoc JavaScript benchmark tests. We'll use this to perform some benchmarking tests.

Apart from the previously-mentioned tools, I'll also cover a list of important testing tools, that I believe are useful for your daily debugging and testing tasks. So, be sure to check out this section.

Sahi

We briefly discussed about the issue of testing user interface widgets provided by JavaScript libraries. In this section, we'll get started with testing a user interface that was built by using the JavaScript libraries widget. The same technique can be used for testing custom user interfaces.

Sahi is a browser-independent, automated testing tool that uses Java and JavaScript. We will focus on this as it is browser-independent, and we cannot always ignore IE users.

Sahi can be used to perform various testing tasks, but the one feature that I would like to emphasize is its ability to record the testing process and play it back in the browser.

You will see how useful it is to use Sahi to perform user interface testing in this section.

Time for action – user Interface testing using Sahi

We will demonstrate to you the recording and play back feature of Sahi, and see how it can be used to test user interface widgets provided by JavaScript libraries such as jQuery.

1. We'll start by installing Sahi. Go to `http://sahi.co.in` and download the latest version. The latest version at this point of writing is V3 2010-04-30. Once you have downloaded it, extract it to the `C:` drive.

2. Open Internet Explorer (I am using IE8 for this demonstration) and go to `http://jqueryui.com/themeroller/`. We will be using the user interface for our demonstration purposes.

3. In order to use Sahi, we need to first navigate to `C:\sahi_20100430\sahi\bin` and look for `sahi.bat`. Click on it so that we can start Sahi.

4. Now, it's time to set up your browser so that it can be used with Sahi. Open your browser, and go to **Tools | Internet Options | Connections** and click on **LAN Settings**. Click on **Proxy Server** and enter the information that you see in the following screenshot:
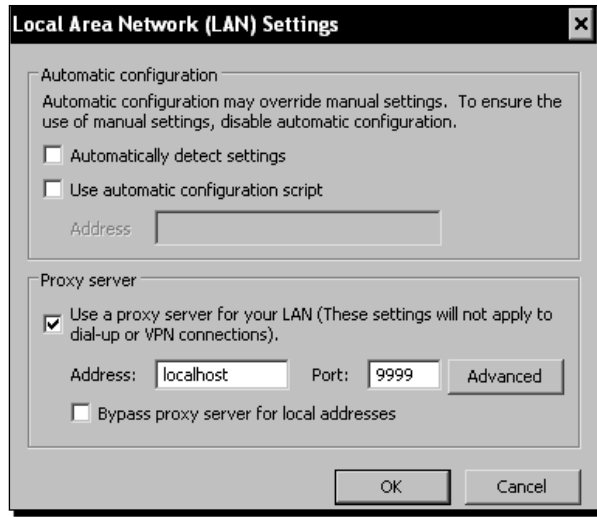
Once you are done, close this window and all other windows related to **Tools**.

5. After you have completed the previous step, let us return to our browser. In order to use Sahi within the browser, you need to press *Ctrl + Alt* and, at the same time, double-click on any element on the web page (`http://jqueryui.com/ themeroller/`). You should see a new window that appears as shown in the next screenshot:
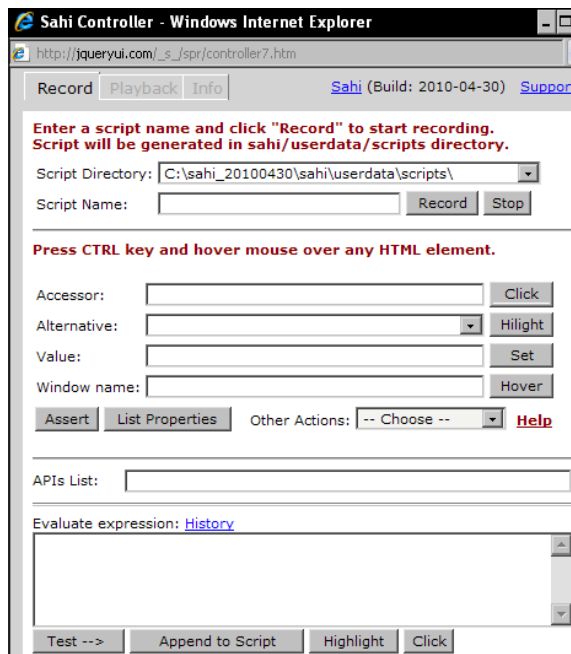
6. If you see the window shown above, then you have set up and started Sahi correctly. Now, let us check out its automated testing feature, recording, and playback capabilities.

7. Enter **jquery_testing** in the **Script Name** input field, and the click on **Record** in the window that is shown in the previous screenshot. This starts the recording process.

8. Now, let us click on a few of the user interface elements. In my case, I clicked on **Section 2**, **Section 3**, **Open Dialog**, and **Font Settings.** This can be found at the menu on the left-hand side.

9. Navigate to `C:\sahi_20100430\sahi\userdata\scripts` and you will see a file named `jquery_testing.sah`. Open this file in WordPad and you will see the list of actions that we have just created, recorded in this file.

10. Go to the Sahi window and click on **Stop**. Now, we have stopped the recording process.

11. Open `jquery_testing.sah` in WordPad and change the code so that it appears like this:

```
function jquery_testing() {
_click(_link("Section 2"));
_click(_link("Section 2"));
_click(_link("Section 3"));
_click(_link("Section 3"));
_click(_link("Open Dialog"));
_click(_link("Font Settings"));
}
jquery_testing();
```

I have defined a function called `jquery_testing()` to contain the list of actions that we have created. Then, I appended `jquery_testing()` to the end of the file. This line is to call the function when we activate the playback feature.

12. Now let us go to the Sahi window and click on **Playback**. Then, enter the information as shown in the next screenshot:
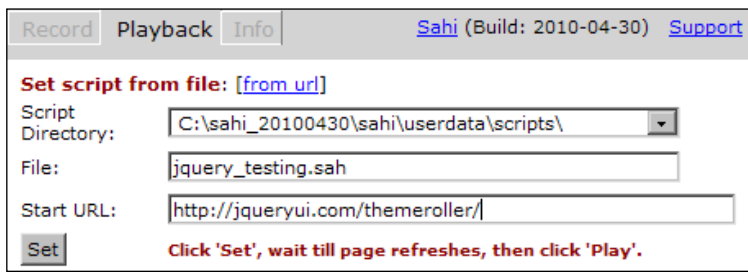
Click on **Set** and wait for the page to refresh.

13. Once the page has been refreshed, click on **Play**. Within the browser, we will see that the actions that we have performed are being repeated as per the steps mentioned previously. You will also receive a **SUCCESS** message in the **Statements** panel, which means that our testing process was successful.

What just happened?

We have just performed a simple user interface testing process by using Sahi. Sahi's playback process and recording features make it easy for us to perform testing on user interfaces.

Notice that Sahi allows us to perform testing in a visual manner. Apart from defining a function for the playback feature, there isn't much coding involved as compared to the other manual testing methods that we have seen in the previous chapters.

Now, let us focus on other important and relevant topics related to Sahi.

More complex testing with Sahi

As mentioned previously at the start of this section, Sahi can be used with any browser to perform a wide variety of tasks. It can even be used to perform assertion tests.

Check out `http://sahi.co.in/static/sahi_tutorial.html` to see how assertion can be used in your testing processes.

> After you are done with this section, make sure that you go back to Tools | Internet Options | Connections, click on LAN settings and uncheck **Proxy Server**, so that your browser can work as usual.

QUnit

Qunit is a jQuery testing suite, but it can be used to test the JavaScript code that we have written. This means that the code does not have to depend on jQuery. In general, QUnit can be used to perform assertion tests and asynchronous testing. Also, assertion testing helps in predicting the returning result of your code. If the prediction is false, it is likely that something in your code is wrong. Asynchronous testing simply refers to testing Ajax calls or functions that are happening at the same time.

Let us act immediately to see how it works.

Time for action – testing JavaScript with QUnit

In this section, we'll learn more about QUnit, by writing a bit of code, and the also learn about various tests that QUnit supports. We will write tests that are correct and tests that are and wrong, in order to see how it works. The source code for this section can be found in the `source code` folder `qunit`.

1. Open your editor and save the file as `example.html`. Enter the following code in it:

```
<!DOCTYPE html>
<html>
<head>
  <title>QUnit Example</title>
  <link rel="stylesheet" href="http://github.com/jquery/qunit/raw/
master/qunit/qunit.css" type="text/css" media="screen">
  <script type="text/javascript" src="http://github.com/jquery/
qunit/raw/master/qunit/qunit.js"></script>
  <script type="text/javascript" src="codeToBeTested.js"></script>
  <script type="text/javascript" src="testCases.js"></script>
</head>
<body>
  <h1 id="qunit-header">QUnit Test Results</h1>
  <h2 id="qunit-banner"></h2>
  <div id="qunit-testrunner-toolbar"></div>
  <h2 id="qunit-userAgent"></h2>
  <ol id="qunit-tests"></ol>
</body>
</html>
```

What the previous code does is that it simply sets up the code for testing. Take note of the highlighted lines. The first two highlighted lines simply point to the hosted version of the QUnit testing suite (both CSS and JavaScript), and the last two lines are where your JavaScript code and test cases reside.

`codeToBeTested.js` simply refers to the JavaScript code that you have written, while `testCases.js` is the place where you write your test cases. In the following steps, you will see how these two JavaScript files work together.

2. We'll start by writing code in `codeToBeTested.js`. Create a JavaScript file and name it as `codeToBeTested.js`. For a start, we'll write a simple function that tests whether a number entered is odd or not. With that in mind, enter the following code into:

`codeToBeTest.js`:

```
function isOdd(value){
  return value % 2 != 0;
}
```

`isOdd()` takes in an argument value and checks if it is odd. If it is, this function will return 1.

Let us now write a piece of code for out test case.

3. Create a new JavaScript file and name it `testCases.js`. Now, enter the following code into it:

```
test('isOdd()', function() {
  ok(isOdd(1), 'One is an odd number');
  ok(isOdd(7), 'Seven is an odd number');
  ok(isOdd(-7), 'Negative seven is an odd number');
})
```

Take note of the way that we write the test case using QUnit's provided methods. First, we define a function call `test()`, which constructs the test case. Because we are going to test the function `isOdd()`, the first parameter is a string that will be displayed in the result. The second parameter is a call-back function that contains our assertions.

We use the assertion statement by using the `ok()` function. This is a Boolean assertion, and it expects its first parameter to be true. If it is true, the test passes, if not, it fails.

4. Now save all of your files and run `example.html` in any browser you like. You will receive a screenshot similar to the following example, depending on your machine:

QUnit Example

Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; InfoPath.2; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)

1. **isOdd() (0, 3, 3)**

1. One is an odd number

2. Seven is an odd number

3. Negative seven is an odd number

Tests completed in 20 milliseconds.
3 tests of 3 passed, 0 failed.
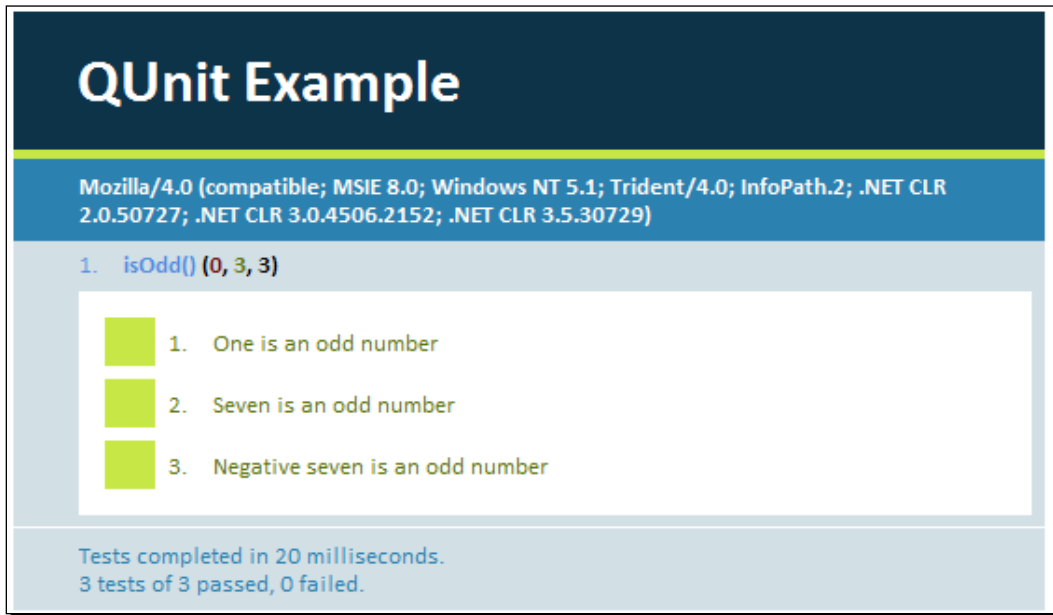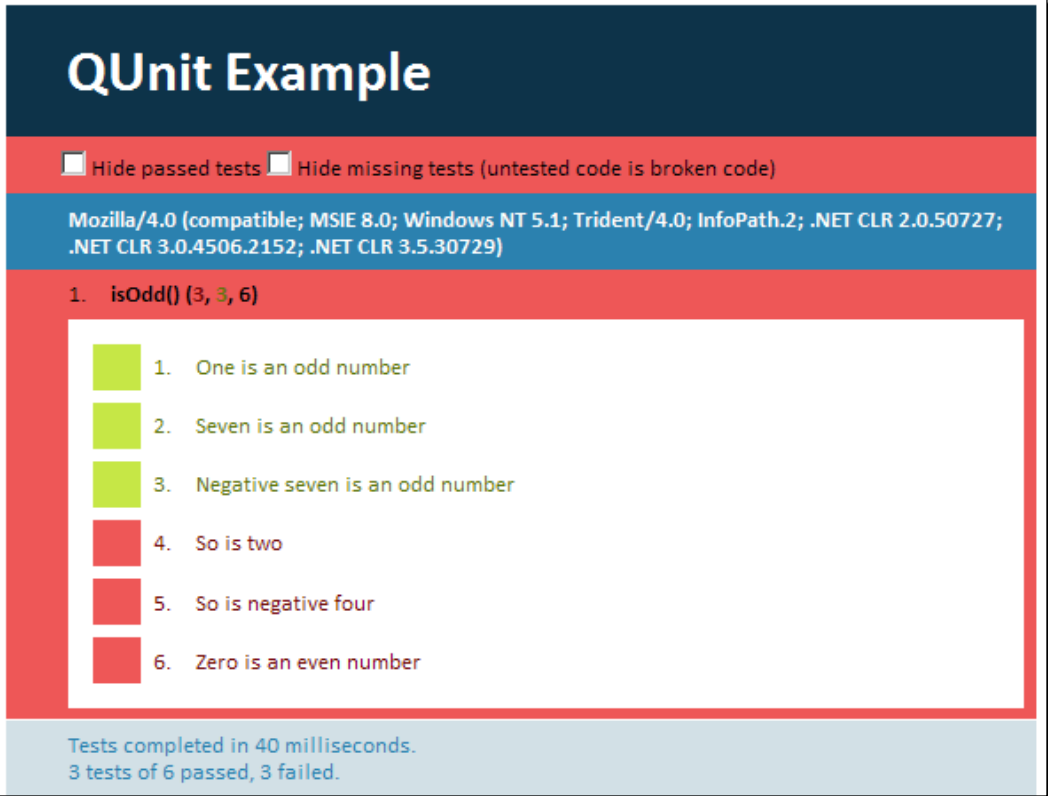
You can see the details of the test by clicking on `isOdd()` and will also see the results of it. The output is as shown in the previous screenshot.

Now let us simulate some fail tests.

5. Go back to `testCases.js`, and append the following code to the last line of `test()`:

```
// tests that fail
  ok(isOdd(2), 'So is two');
  ok(isOdd(-4), 'So is negative four');
  ok(isOdd(0), 'Zero is an even number');
```

Save the file and refresh your browser. You **will now see a screenshot** similar to the following example in your browser:

QUnit Example

☐ Hide passed tests ☐ Hide missing tests (untested code is broken code)

Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; InfoPath.2; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)

1. **isOdd() (3, 3, 6)**

   1. One is an odd number
   2. Seven is an odd number
   3. Negative seven is an odd number
   4. So is two
   5. So is negative four
   6. Zero is an even number

Tests completed in 40 milliseconds.
3 tests of 6 passed, 3 failed.

Now you can see that tests **4**, **5**, **and 6** have failed and **they are in red.**

At this point you should see that the good thing about QUnit is that it largely automates the testing process for us without us having to perform testing by clicking on buttons, submitting forms, or using `alert()`. It will certainly save us a tremendous amount of time and effort when using such automated tests.

What just happened?

We have just employed QUnit in performing automated testing of self-defined JavaScript functions. It was a simple example, but enough to get you started.

Applying QUnit in real-life situations

You might wonder how you will make use of such tests on your code in real-life situations. I would say that it is very likely that you will use `ok()` to test your code. For instance, you can test for the truth values, if the user input is alphanumeric, or if the user has entered invalid values.

More assertion tests for various situations

Another thing that you can take note of is that `ok()` is not the only assertion test that you can perform. You can also perform other tests, such as comparison assertion and identical assertion. Let us see another short example on comparison.

We'll learn to use another assertion statement, `equals()`, in this section.

1. Open your editor and open `testCases.js`. Comment out the code that you have written previously, and enter the following code into the file:

    ```
    test('assertions', function(){
      equals(5,5, 'five equals to five');
      equals(3,5, 'three is not equals to five');
    })
    ```

    This code takes the same structure as the code that you have commented out. But notice that we have used the `equals()` function instead of `ok()`. The parameters of `equals()` are as follows:

    -   The first parameter is the actual value

    -   The second parameter is the expected value

    -   The third parameter is a self-defined message

    We have used two `equals()` functions, of which the first will pass the test, but the second will not as three and five are not equal.

2. Save the file and open `example.html` in your browser. You will see the following screenshot:

JSLitmus

According to JSLitmus's homepage, JSLitmus is a lightweight tool for creating ad hoc JavaScript benchmark tests. In my opinion, it is definitely true. Using JSLitmus is quite a breeze, especially when it supports all popular browsers, such as Internet Explorer, Firefox, Google Chrome, Safari, and others. At the same time, it is entirely free with the products that we mentioned here.

In this section, we will focus on a quick example of how we are going to create ad hoc JavaScript benchmark tests.

Time for action – creating ad hoc JavaScript benchmark tests

Now we will see how easy it is to create ad hoc JavaScript benchmark tests by using JSLitmus. But first, let us install JSLitmus. By the way, all of the source code for this section can be found in the `source code` folder for this chapter, under the `jslitmus` folder.

1. Visit `http://www.broofa.com/Tools/JSLitmus/` and download `JSlitmus.js`.

2. Open your editor, create a new HTML file within the same directory as `JSLitmus.js` and name it `jslitmus_test.html`.

3. Enter the following code into `jslitmus_test.html`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en">
  <head>
    <meta http-equiv="Content-Type"
     content="text/html;charset=utf-8" />
    <title>JsLitmus Testing Example</title>

    <script type="text/javascript" src="JSLitmus.js"></script>
    <script type="text/javascript">
      function testingLoop(){
        var i = 0;
        while(i<100)
        ++i;

        return 0;
      }

      JSLitmus.test('testing testingLoop()',testingLoop);

    </script>
  </head>

  <body>
    <p>Doing a simple test using JsLitmus.</p>
    <div id="test_element" style="overflow:hidden; width: 1px;
     height:1px;"></div>
  </body>
</html>
```

I've actually taken this code from the official example found on the JSLitmus website. I will conduct the test in a slightly different manner to the official example, but nonetheless, it still demonstrates the syntax of how we can use JSLitmus.

The previous code snippet contains the user-defined function `testingLoop()`, while the `JSLItmus.test('testing testingLoop()', testingLoop);` is the line of JavaScript code written to test `testingLoop()` by using JSlitmus's syntax.

Let me explain the syntax. Generally, this is how we use JSLitmus:

```
JSlitmus.test('some string in here', nameOfFunctionTested);
```

The first argument is some string that you can type in, and the second argument is the name of the function that you intend to test. Just make sure that this line of code is located in a place after your function is defined.

4. Now that we have set up our test, it's time to run it and see how it goes. Save `jslitmus_test.html` and open this file in your browser. This is what you should see in your browser:
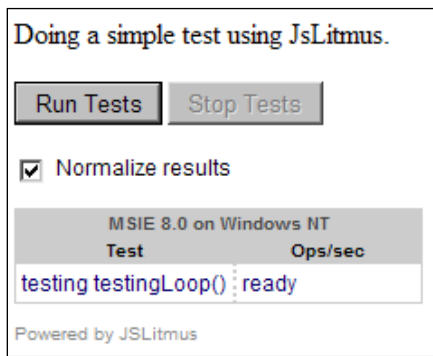
Notice that under the **Test** column, it shows the text that we have typed in as our first argument for `JSLItmus.test()`.

5. Click on the button **Run Tests**. You should receive the following result in your browser:
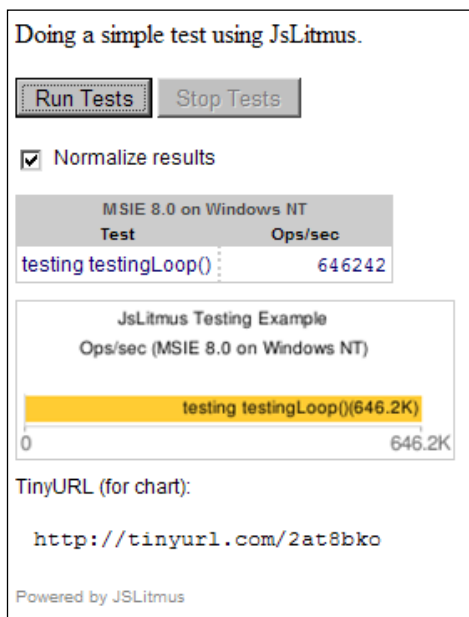
It basically shows the amount of time taken to execute the code, and other relevant information. You can even check out the performance in chart form by visiting the URL that is created dynamically. If you received something similar to the previous screenshot, then you have just done an ad hoc benchmarking test.

> If you are running this test on Internet Explorer and you happen to receive the following (or similar) message: **Script is taking too long to execute**, then you will need to tweak your Windows registry, in order to allow the test to run. Visit `http://support.microsoft.com/default.aspx?scid=kb;en-us;175500` for details on how to tweak your windows registry settings.

What just happened?

We just used JSLitmus to create an ad hoc benchmarking test. Notice how easy it is for you to perform ad hoc benchmarking test using JSLitmus. The cool thing about JSLitmus is the simplicity of it; no additional tools, no windows to open, and so on. All you need to do is to write `JSLItmus.test()` and type in the message and name of the function you want to test.

More complex testing with JSLitmus

The previous example is a really simple example to help you to get started. If you are interested in performing more complex tests, feel free to check out `http://www.broofa.com/Tools/JSLitmus/demo_test.html` and view its source code. You will see different style of writing test by using JSLitmus in its well-commented source code.

Now that we have covered the tools that are browser-independent, it is time to quickly cover other similar testing tools that can help you to debug JavaScript.

More testing tools that you should check out

Now that we are approaching the end of the chapter, I'll leave you with a simple list of testing tools that you can check out for testing purposes:

- ◆ Selenium: **Selenium** is an automated testing tool that can record only on Firefox and may time out when trying to playback in other browsers. There are also other versions of Selenium that can help you to conduct tests in multiple browsers and platforms. Selenium uses Java and Ruby. To get more information, visit `http://seleniumhq.org`. To see a simple introduction, visit `http://seleniumhq.org/movies/intro.mov`.

◆ Selenium Server: Also known as Selenium Remote Control, **Selenium Server** is a test tool that allows you to write automated web-application UI tests in any programing language, against any HTTP website, using any mainstream JavaScript-enabled browser. You can visit `http://seleniumhq.org/projects/remote-control/`.

◆ Watir: **Watir** is an automated testing tool available as a Ruby gem. There is detailed documentation on Watir, which can be found at `http://wiki.openqa.org/display/WTR/Project+Home`.

◆ Assertion Unit Framework: The **Assertion Unit Framework** is a unit testing framework based on assertions. At the point of writing, documentation appears to be limited. But you can learn how to use it at `http://jsassertunit.sourceforge.net/docs/tutorial.html`. You can visit `http://jsassertunit.sourceforge.net/docs/index.html` for other relevant information.

◆ JsUnit: **JsUnit** is a unit testing framework ported from the most popular Java unit testing framework known as JUnit. JsUnit includes a platform for automating the execution of tests on multiple browsers and multiple machines using different operating systems. You can get JsUnit at `http://www.jsunit.net/`.

◆ FireUnit: **FireUnit** is a unit testing framework designed to run in Firebug on Firefox. It is also a popular debugging tool for Firefox and there are numerous tutorials for it and documentation on it, on the Internet. You can get FireUnit at `http://fireunit.org/`.

◆ JSpec: **JSpec** is a JavaScript testing framework that utilizes its own custom grammar and pre-processor. It can also be used in variety of ways, such as via a terminal, via browsers using DOM or Console formatters, and so on. You can get JSpec at `http://visionmedia.github.com/jspec/`.

◆ TestSwarm: **TestSwarm** provides distributed, continuous integration testing for JavaScript. It was originally developed by John Resig to support the jQuery project and, has now become an official Mozilla Labs project. Take note that it is still under heavy testing. You can get more information at `http://testswarm.com/`.

Summary

We have finally reached the end of this chapter. We have covered specific tools for various browsers that can be used for our debugging tasks.

Specifically, we covered the following topics:

- Sahi: **A browser-independent automated testing tool that uses Java and JavaScript**
- QUnit: **A jQuery testing suite that can be used to test JavaScript code**
- JsLitmus: **A lightweight tool for creating ad hoc JavaScript benchmark tests**
- A list of tools that you can check out

Finally, we have reached the end of the book. I hope that you have learnt a lot from this book about JavaScript testing. I want to thank you for your time and effort in reading this book, and would also like to thank Packt Publishing for it's support.

Index

H

HTML
  about 7, 8
  debugging, IE8 developer tool used 204, 205
  elements 8
HTML document
  creating 9-11
  JavaScript, applying 20-23
  styling, CSS used 14-16
  styling, stylistic attributes used 18
HTML DOM availability
  checking 77
HTML elements
  <a> 8
  <body> </body> 8
  <h1> </h1> 8
  <head> </head> 8
  <p> </p> 8
  <title> </title> 8
  class name, specifying 12
  id, specifying 12
  styling, attributes used 11, 12
HTML event handlers
  avoiding 103-106
Hyper Text Markup Language. *See* **HTML**

I

id attribute 12
id selectors 19
IE 8 developer tools 202
IE debugging tool
  accessing 202
  CSS, debugging 205
  debugging basics 203, 204
  features 203
  HTML, debugging 204
  JavaScript, debugging 206-210
IE developer toolbar
  installing 202
IE developer tools
  about 202
  IE debugging tool, accessing 202
  using 202
illegal values phase, test plan
  test cases 144, 145

  using 144
innerHTML() method 29
insertContent() function 28
integrated testing 127, 128
invalidated code
  consequences 85

J

JavaScript
  about 7, 20
  and server side languages, differences 29
  applying, to HTML document 20-23
  debugging, IE8 developer tool used 206-210
  elements, searching in document 26, 27
  error, encountering by browser 44
  exception handling mechanisms, applying
        184-199
  features 41
  interacting, with HTML elements 28
  testing, QUnit used 236-239
  usability, enhancing 163
JavaScript code
  testing 82
  validating 82
JavaScript errors
  about 32, 172
  catch statement 172-175
  finally statement 172-175
  loading errors, types 33
  logic errors, types 37
  runtime errors, types 36
  throw statement 172
  trapping, built-in objects used 176
  try statement 172-175
  types 32
JavaScript events 26
JavaScript libraries
  about 169
  considerations 170
  Dojo 169
  GUI 171
  JQuery 169
  link 170
  Mootools 169
  performance testing 170
  profiling testing 171

Thank you for buying
JavaScript Testing Beginner's Guide

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] PUBLISHING
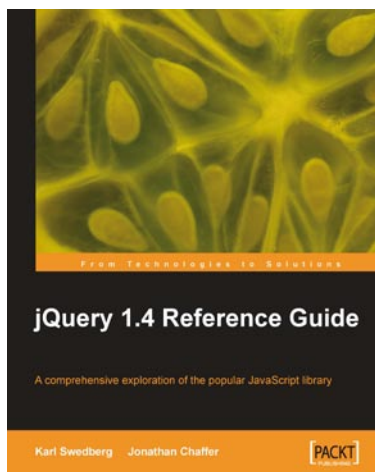
jQuery 1.4 Reference Guide

ISBN: 978-1-849510-04-2   Paperback: 336 pages

A comprehensive exploration of the popular JavaScript library

1. Quickly look up features of the jQuery library

2. Step through each function, method, and selector expression in the jQuery library with an easy-to-follow approach

3. Understand the anatomy of a jQuery script

4. Write your own plug-ins using jQuery's powerful plug-in architecture

Learning Ext JS

ISBN: 978-1-847195-14-2   Paperback: 324 pages

Build dynamic, desktop-style user interfaces for your data-driven web applications

1. Learn to build consistent, attractive web interfaces with the framework components

2. Integrate your existing data and web services with Ext JS data support

3. Enhance your JavaScript skills by using Ext's DOM and AJAX helpers

4. Extend Ext JS through custom components

Please check **www.PacktPub.com** for information on our titles

ICEfaces 1.8: Next Generation Enterprise Web Development

ISBN: 978-1-847197-24-5   Paperback: 292 pages

Build Web 2.0 Applications using AJAX Push, JSF, Facelets, Spring and JPA

1. Develop a full-blown Web application using ICEfaces

2. Design and use self-developed components using Facelets technology

3. Integrate AJAX into a JEE stack for Web 2.0 developers using JSF, Facelets, Spring, JPA
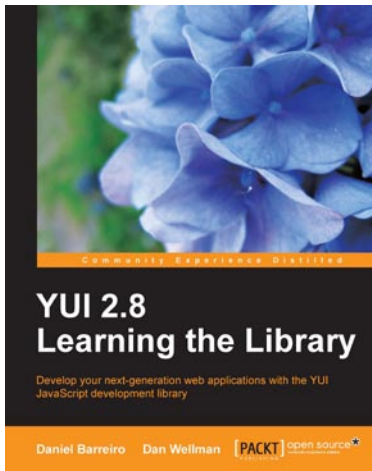
YUI 2.8: Learning the Library

ISBN: 978-1-849510-70-7   Paperback: 404 pages

Develop your next-generation web applications with the YUI JavaScript development library

1. Improve your coding and productivity with the YUI Library

2. Gain a thorough understanding of the YUI tools

3. Learn from detailed examples for common tasks

Please check **www.PacktPub.com** for information on our titles