Winter 2015 Version

Winter 2014 Version

Contents

0. About This Book

JavaScript for Sheridan Students is intended to be a complete course in **JavaScript** programming for **web application** design. In this book, you will learn the basics of the JavaScript language and how to embed JavaScript programs into a web page; you will learn how to create **client-side** web apps; you will learn how to effectively use the extensions provided in the widely-used **jQuery** package, including its **AJAX** capabilities; you will learn how to use the new **HTML5 canvas** to create dynamic, graphical web content; you will learn how to access the new **web storage** capabilities of HTML5; and you will learn how to implement **object-oriented** design in JavaScript.

0.1 Who Should Read This Book?

JavaScript for Sheridan Students is aimed at Students in the Applied Computing programs at **Sheridan College [http://www.sheridancollege.ca]**, though it could also be used effectively by students in other contexts or even by experienced programmers who are new to **JavaScript**.

When Sheridan students first encounter JavaScript, they have already completed a first programming course in **Java** and a first course in web design, in which they were asked to write HTML and CSS by hand (i.e. not using a **WYSIWYG** editor like DreamWeaver). These students already have a good understanding of **variables**, **data types**, **if statements**, **loops**, **methods**, **HTML elements** (tags and **attributes**), and **CSS rules** (properties and **values**). They have designed simple web pages and written computer programs that are split up into multiple methods (possibly also multiple **classes**) to process user input and produce new output.

Java Connection

If you are a Sheridan student or if you have just completed an introductory course in **Java**, **JavaScript for Sheridan Students** was written just for you!

If you fit the profile above, this text will leverage your prior knowledge of programming to skim over stuff you already know and **focus** on the differences and unique capabilities of JavaScript, **jQuery**, and **HTML5**. It will also leverage your existing web design skills to get you creating **client-side web applications** in JavaScript more quickly.

Do It Yourself

Look for DIY (Do It Yourself) boxes like this throughout this text. They are here to give you something to do, to keep you engaged and thinking, and to break up the monotony of reading. But they're not optional! You should read them and try the activities before moving on. Think of it as an opportunity to experiment, try new things, and test out what you have already learned.

0.2 What Is JavaScript?

JavaScript is a fully-functional programming language that was designed to be embedded in an HTML page to create **Dynamic HTML** (DHTML). It was first shipped with the now-defunct **Netscape** browser in 1995. DHTML pages make use of JavaScript **statements** and **functions** (similar to **Java methods**) to change their appearance and contents both as they are loading and after they are loaded. (Static HTML pages, on the other hand, always look the same and do not change once they are loaded. Static pages used to be the only thing on the web, but they are now all but extinct.)

JavaScript was standardized as **ECMAScript** a couple of years after its initial release. The name "JavaScript" is a trademark of the **Oracle** corporation, but other implementations of the ECMAScript also exist (notably Microsoft's **JScript** used in Internet Explorer and Adobe's **ActionScript** used in Flash).

Java Connection

Look for these boxes throughout the text to get a quick summary of the similarities and differences between **Java** and **JavaScript**.

Almost all **web applications** make use of JavaScript for drop down menus, pop-up windows, slide shows, image magnifiers and other dynamic **elements** that can be implemented on the client side of the **client-server** architecture. Many web sites also make use of JavaScript with **AJAX** to load new content from the server and refresh parts of the page after load time. This is how **Google [http://www.google.com]** makes suggestions as you type into the search bar, and how **Facebook [http://www.facebook.com]** loads new content when you get to the bottom of your activity feed.

By at least one measure, JavaScript might be the most frequently used programming language: If you counted up all the lines of JavaScript in the browser caches of all the computers in the world, the total would probably be higher than the number of lines of source code in any other language.

0.2.1 Popular Misconceptions About JavaScript

There are two widely-held misconceptions about JavaScript. The first is that Java and JavaScript are the same language, or two versions of the same language. In fact, JavaScript is a very different programming language from Java. The similarity in names is the result of a marketing decision made by Netscape in the 1990's. It's true that the two languages can look similar at first glance. Like Java, JavaScript derives much of its basic **syntax** from the **C programming language**. But don't be fooled by surface appearances. JavaScript's **variables**, **arrays**, **objects**, and functions are implemented very differently from their counterparts in Java.

The second misconception is that JavaScript is somehow a less powerful, simpler, or more "lightweight" language than Java. In this case, there are several kernels of truth. It is true that JavaScript is designed to run in a restricted environment (e.g. a web page, a PDF document, the Windows desktop, etc.). It is also true that historically, JavaScript has been used mostly for very simple tasks (e.g. image **rollovers** and drop-down menus). And it is also true that JavaScript runs more slowly than Java (it's **interpreted** rather than **compiled**), making it less suited to high performance applications. Finally, it is also true that JavaScript programmers do not have easy access to the same kind of standard plugin libraries that Java programmers enjoy.

But JavaScript is not "lightweight" or "simpler" in the sense of being easier to learn. Writing effective code in JavaScript can be just as difficult as any other language, and a deep understanding of how the language works is as important to being an effective JavaScript programmer as it is for a programmer in any other language.

JavaScript is also not "less powerful" in the theoretical sense than any other language. JavaScript is a fully developed, **Turing complete**, programming language. This means that there is no computational problem that can be solved in Java that could not also be solved in JavaScript.

0.2.2 The Future of JavaScript

With the widespread adoption of **HTML5**, JavaScript is poised to become the language of choice for web-based, hybrid, and possibly even standalone mobile apps, and it will soon replace Adobe's Flash as the primary way to produce graphics-intensive web content like games, interactive diagrams and fancy menu systems. With the development of **Node.js**, JavaScript can now also be used to write code on the **server side** of a **web app**.

And JavaScript is not just for web apps any more. JavaScript **interpreters** are also embedded into the Microsoft Windows Desktop, Adobe Acrobat, Open Office, and a number of other environments to allow programmers to add apps and other dynamic functionality to these products.

It may not be long before a solid working knowledge of JavaScript is a requirement for many programming jobs.

0.3 How To Use This Book

JavaScript for Sheridan Students was designed using **JavaScript**, **jQuery** and **CSS3 Media Queries** to make it look good and be usable on a tablet, in a narrow desktop browser, and on the printed page. This is to make it easier for you to try things out on your computer as you read. The book also makes use of the **HTML5 App Cache**, which means after you load it for the first time the main text will always be available, even if your device has no Internet connection. Bookmark it once, download the **example pack** once, and you will always have everything you need.

Since it is a **web app** written in HTML5, CSS3 and JavaScript, the book can also serve as a testbed to try out JavaScript **statements** as you read along.

You will see lots of links and highlighted terms in the book. In most cases these are links to **Wikipedia** entries, **w3Schools** pages or online examples . When you print the book, the links appear in boldface. If the link points to anything other than Wikipedia, w3Schools or an online example, the URL will appear beside it.

0.3.1 W3Schools and Wikipedia

W3Schools is a standard reference site for JavaScript, HTML, CSS, and other web technologies. It also has a "Try it Yourself Editor" that lets you tweak example code to see the results immediately. You should get to know this site as a good quick **reference**.

Wikipedia is a **crowdsourced** encyclopedia. For computer-related topics, the entries are usually of acceptable quality, thanks to a very large number of working computer scientists and computer programmers who police the content and keep it up to date and accurate. You should use the Wikipedia links to brush up on the meanings of terms you may have forgotten and to dig deeper into topics that interest you.

But as good as these resources are, they have some limitations. The text in w3Schools is brief and does not always work well for students without much prior programming experience, and some of the topics in Wikipedia can be difficult for a novice to read. Beginner students need detailed explanations of big ideas, lots of signposts along the way, worked examples and exercises. So we will focus on that stuff here. You are encouraged to move back and forth between this text, w3schools, Wikipedia and other sources to help build your understanding.

> ***Do It Yourself***
>
> If you have never used **w3schools** before, you should get to know it. Before moving on, pay w3schools a visit and follow one of the trails (e.g. click on the HTML link and start reading). Be sure to try out some of the "Try It Yourself" activities.

0.3.2 Critical Thinking

All sources of information (including the one you're reading now) should be treated with caution. I think w3Schools is a great resource to get quick details on HTML, CSS and JavaScript, but it is not without its critics (see the **w3fools [http://www.w3fools.com/]** site). It has gaps and sometimes it's not as up-to-date as it should be. I also think Wikipedia generally has high quality entries for computer-related topics, but it is crowdsourced so on any given day the information might be inaccurate or out of date as well.

You should always be aware of the drawbacks of any on line source and develop strategies for cross-checking information from multiple sources. If you think you have found an error in this text, please **let me know [mailto:sam.scott@sheridancollege.ca]** so I can investigate.

1. Getting Started with JavaScript

> ***Java Connection***
>
> **Syntax.** JavaScript's core **syntax** is very similar to **Java**. But don't be fooled by surface appearances. Some deep differences lurk beneath.

Because **JavaScript** and **Java** are both based on **C syntax**, and because you have some experience in Java, you should have no problem with many of the nuts and bolts of the language. **Statements**, comments, operators, comparisons, selection statements (`if` and `switch`), repetition statements (`for`, `while`, `do..while`) and even catching and throwing exceptions all work pretty much exactly how you would expect them to. JavaScript **string objects** and the special Math object also contain most of the same **methods** as their Java counterparts (e.g. `charAt`, `Math.random`, etc.)

This is great news. You already know most of the **syntax** of JavaScript and you haven't even done anything yet! But before you can leverage all that existing knowledge and start writing JavaScript programs, you need to understand two core features of JavaScript that make it very different from Java: its support for **imperative programming** and its **dynamically-typed** and **weakly-typed** approach to **variables**, **values** and **data types**.

> ***Do It Yourself***
>
> If you have some programming experience, but not with a C-like language, or if you need to brush up on your **syntax**, **w3schools** is a great place to go for a refresher. Just click on the **Learn JavaScript** link and then choose the topics from the list on the left. Even if you're feeling confident, you should probably take a quick look at the sections on **JS String** and **JS Math** because there are some slight differences from the **Java String** and Math **classes**.

1.1 Imperative Programming

> ***Java Connection***
>
> **Statements.** In **Java**, every **statement** must be part of a **class** or **method**. This is not true in **JavaScript**.

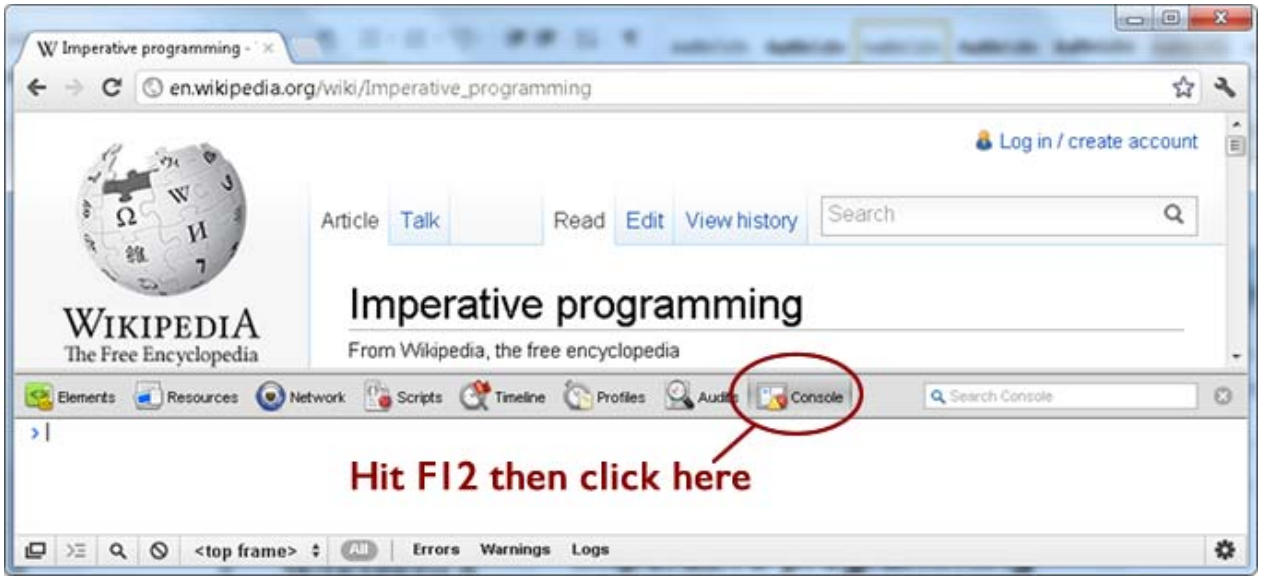
Purely **imperative** programs consist of a list of **statements** in the order they are to be executed. These statements do not need to be part of a **class**, **object**, **method**, **function** or any other structure.

In human languages, imperative sentences are commands: "Do this. Now do that." Imperative programs read just like that to the computer. There are no pleasantries, you just get straight to the instructions. This is quite different from **Java**, where you have to define both a class and a main method before you can get to the commands.

1.1.1 Imperative Programming in a Browser Console

The easiest way to execute JavaScript statements is using the JavaScript console of your favorite browser (a.k.a. **Google Chrome [https://www.google.com/intl/en_ca/chrome/browser/]**). In Chrome, hit F12 to open the developer tools, then choose "Console" from the tabs that appear at the bottom of the page.

The console will appear as a new panel beneath the web page, like this.



(What's that you say? Chrome isn't your favorite browser? Well you have two options. You can either make Chrome your new favorite, or you can search through the menus on any another browser for "Developer Tools", "JavaScript Console" or something similar.)

Do It Yourself

If you're reading this on line, open the JavaScript console now (F12 in Chrome). If you're not reading this on line, open any web page in Chrome or another browser, then go to the developer console.

Now you're ready to execute your first JavaScript statement. At the flashing cursor beside the '>', type:

```
window.alert("Hello, World!");
```

...and hit enter. See the popup message?

What you have just done in the DIY box above is execute a **JavaScript** function named `alert` that is part of the built-in `window` object. JavaScript functions are a lot like Java methods or C functions – you call them by typing their name followed by a list of **arguments** inside round brackets. If a function belongs to an object, you type the name of the object first followed by the . (dot) operator.

No matter what browser you are using, you will always have access to a `window` object containing **variables** and functions pertaining to the currently active panel of the browser. And because you are always "inside" the `window` object when executing JavaScript, you don't actually need to type the "`window.`" part. So this would have worked, too:

```
alert("Hello, World!");
```

Java Connection

Semicolons. In **JavaScript**, you usually don't need a semicolon at the end of a **statement**, but sometimes you do. Instead of trying to remember which case is which, it's better to always include one.

Java Connection

String Literals. In **Java**, double quotes are for `Strings` and single quotes are for `chars`. **JavaScript** doesn't have a `char` type, so you can use either type of quote character for a **string**. This means you don't have to use **escape characters** to put quotes inside a string (e.g. instead of `"\"Murder,\", she wrote"`, you can use single quotes like this: `'"Murder,", she wrote'`).

Do It Yourself

Here are a couple more window functions to try in a Chrome console window.

```
prompt('What is your name? ');
confirm("You're programming in JavaScript!");
```

Note the **return values** of each function that appear in the console window when you try the statements in the above DIY box. Just like Java and other languages, you can use the return value of one function as an argument to another.

Before we move on, a few more useful tips:

1. If you ever want to repeat a command in the Chrome console, you can always hit the up arrow on your keyboard to cycle through recent commands, tweak them, and try again
2. If you ever want to enter a multiple line command, use `shift-enter`.
3. If you want to complete a command using one of Chrome's pop-up suggestions, navigate to it using up and down arrows, then complete the command using the right arrow
4. If you want to clear the console screen, use `ctrl-L` (or you can use the `console.clear()` function)
5. If you want to detach the console and use it in a separate window, press the windows icon on the top right (beside the X icon). If you change your mind, press the same button again to reattach it.

Do It Yourself

In a Chrome console window, try these two separate statements that use the return values from the prompt and confirm functions, then make up some of your own.

```
alert(prompt('What do you want to say? '));

if (confirm('OK?')) alert("You're OK"); else alert("You're not OK?");
```

The `prompt`, `alert` and `confirm` functions are pretty straightforward and are discussed under the heading **JS PopupAlert** on **w3Schools**.

The Chrome console is a great tool for testing out JavaScript statements and expressions to get the **syntax** right before incorporating them into a web page. It can also help you remember method names by popping up suggestions as you type...

Do It Yourself

Type "Math.r" in the Chrome console window and look at the suggestions that pop up

... and it can evaluate JavaScript expressions for you.

Do It Yourself

Try the following in a Chrome console window. (Make sure you get the upper and lower case right in these examples or they won't work.)

```
5 + 3 * 2
5 <= 3
Math.round(Math.PI*2)
Math.round(Math.PI*2) < Math.PI*Math.PI
```

1.1.2 Imperative Programming in an HTML <script> Element

Of course the real point of JavaScript is to embed a program into a web page. You can put JavaScript statements anywhere you want in an **HTML** document, as long as they are inside a `<script>` **element** with a `type` **attribute** set to `"text/javascript"`, as in the example below.

```
<script type="text/javascript">
    alert("Hello, world!");
</script>
```

You can put any number of `<script>` elements anywhere inside the `<body>` or `<head>` elements of an HTML document. The browser will read the page from top to bottom as it loads it. When it gets to a `<script>` element, it will execute the commands inside before moving on. So if the `<script>` element appears inside the `<head>`, the page will be blank until the script has finished executing. If it appears in the middle of the `<body>`, the first part of the page will be displayed, then the `<script>` element will run, then the rest of the page will be displayed.

1.1.3 A Note on IDEs

You can write HTML, CSS, and JavaScript in a text editor and test it in a browser, but there are much more programmer-friendly environments out there for code creation. I recommend that you at least use a programming editor like **NotePad++ [http://notepad-plus-plus.org/]**. This program will use color to highlight keywords, **literals**, and other elements, and will also do some simple bracket matching for you.

Better still is to use a fully-developed Integrated Development Environment (IDE) like **NetBeans [https://netbeans.org/]** or **Aptana Studio [http://www.aptana.com/]** (which is based on **Eclipse [https://eclipse.org/]**). For web programming, I like NetBeans the best, especially now it has support for HTML5 projects and connects to Chrome using the **NetBeans Connector Plugin [https://chrome.google.com/webstore/detail/netbeans-connector/hafdlehgocfcodbgjnpecfajgkeejnaa]**. Just download the latest copy of NetBeans, then start an HTML5 Project, right click it in the project window and create an HTML document. You'll get a starting template, errors and warnings, as well as shortcuts and suggestions. You can also create CSS and JavaScript files this way, and you can "run" the files in Chrome to test them once you have installed the NetBeans plugin from the **Chrome Web Store [https://chrome.google.com/webstore/category/apps]**.

> ***Do It Yourself***
>
> Create a new HTML5 project in **NetBeans [https://netbeans.org/]** (or your favorite IDE) and inside that project create a new **HTML** page. You should get an **HTML5** template but just in case you don't, you can download **template.html** from the **example pack** and use that instead.
>
> Make sure you can run your template (i.e. display it in a web browser) then add the `<script>` **element** shown in the previous section somewhere in the page and refresh it to see what happens.
>
> Try moving the `<script>` element to different positions in the file. What do you notice about the behavior of the page when you reload it with the script element in different positions? What happens if you have multiple copies of it in the file? Is there anywhere in the file you can't put this element?

1.1.4 Debugging Output

Once we get past this opening chapter, we won't use alert boxes very much to talk to the user. They're rather inelegant and obtrusive. But you might still want to have a way to create debugging output to let you know what is going on inside your JavaScript programs. For this, you can use the `console.log` function. It's a bit like `System.out.println()` in Java. It sends output to the browser's console where nobody but us developers will see it.

Do It Yourself

Go back to the **NetBeans [https://netbeans.org/]** project from the last DIY box and change the `alert` **function** to `console.log`. Refresh the page to see the message end up in the browser's console window.

1.2 Dynamic (and Weak) Typing

Another important feature of **JavaScript** that sets it apart from **Java** is that its **variables** are **dynamically-typed**. Another way to say this is that **values** have fixed **data types** but variables do not. JavaScript supports a number of data types including `Number`, `String`, `Boolean`, `Array` and `Object` (you can look these up in **w3schools** under **JS Data Types**) but you do not have to **declare** a variable's type in order to use it and you can change the type of a variable as the program runs.

Unlike in Java where you have to declare a variable by specifying its type, variable **declaration** in JavaScript is done with the generic keyword `var`.

Do It Yourself

Try the statements below in the JavaScript console of your browser.

```
var x = 100;
alert(x);

x = "I'm a string now";
alert(x);

x = 45.3;
alert(x);

x = "$"+x;
alert(x);
```

(Here's a tip: To reset the console and make it forget all the variables you have defined in this session, just hit the refresh button on your browser.)

The examples in the DIY box above show that not only can `x` change its type from a number to a **string** and back again, the `alert` **function** can also take an **argument** of any type.

Other than the fact that variables are dynamically-typed, they work in pretty much the way you are used to from Java.

Java Connection

Data Types. Java is statically-typed, but **JavaScript** is **dynamically-typed**. This seemingly small difference has ripple effects throughout the entire language.

Do It Yourself

Here's a little **loop** for counting to 10. You can try it out by creating a simple **HTML** file with a `<script>` **element**. You could also try it it in your **JavaScript** console using `shift-enter` to separate the lines, or if you're online right now, just copy and paste it into the console.)

```
for(var i = 1; i <= 10; i++)
    console.log(i);
```

Here's another one to try.

```
var n = prompt("Enter a number");
if (n < 1000)
    alert("that was a small number");
else
    alert("that was a big number");
```

Note that in this case the **variable** n contains a **string**, but JavaScript still allows the comparison to happen. This is because JavaScript practices not only dyamic but also weak typing. More on this in the next section.

Actually, you can often use variables without declaring them at all, but this is considered very bad programming practice, for reasons we will discuss later. **Always declare your variables!**

<exercises section='1.2'>

1. Write a script on a web page that presents the user with an arithmetic problem (e.g. `What is 23 + -4?`) and asks for the answer using a `prompt`. Then give them feedback using an alert box.
2. Extend the script so that it generates the integers for the arithmetic problem randomly. Keep presenting the question until they get it right. Then give them a success message in an alert box.
3. Extend the script so that it asks the user if they would like another question using a confirm dialog and gives them another question if they press "OK". If they press "Cancel", the page should finish loading and display a goodbye message.

</exercises>

Java Connection

Random Numbers. `Math.random()` works the same way in both **Java** and **JavaScript**. But you can't cast to an integer with (int) in JavaScript. (Why not?) You can use `parseInt()` or `Math.floor()` instead. Look up the **JavaScript Math object** on **w3schools** for more info.

1.2.1 Weak Typing and Boolean Expressions

In **Boolean** expressions (i.e. the expressions that evaluate to `true` or `false` to control **if statements** and **loops**) JavaScript uses aggressive **implicit typecasting** (also known as weak typing) to compare any two values regardless of whether or not their types match. Typecasting is the act of converting one data type to another. Implicit typecasting is when the **compiler** or **interpreter** does the conversion automatically for you.

Java Connection

Implicit Typecasting. `"double x=45"` is an example of implicit type casting in **Java** (an `int` **value** is converted and then assigned to a `double` **variable**). **JavaScript** also performs **implicit typecasting**, but does it in many more cases than Java does. Because of this we say that JavaScript is **weakly-typed** while Java is strongly-typed.

Do It Yourself

To see JavaScript's aggressive implicit typecasting in action, try the following in the Chrome console:

```
var x = "hi";
x == 45.3
x >= 45.3
x == "hi"
```

The first two comparisons above are legal but `false`. The last one is `true`.

If a String contains a representation of a number, it can be compared to integers and floats...

Do It Yourself

Try the following in a JavaScript console:

```
var y = "50";
x = 50;
y == x
```

This comparison is `true`, even though `y` is a **string** and `x` is a number

Because of JavaScript's weak approach to types, there is a special Boolean operator === that returns true if two values are equal and also of the same type - this is known as being "exactly equal" or "identical". Similarly, `!==` is a "not exactly equal to" operator that respects the types of the operands. See **JS Comparison Operators** on w3schools for more info.

Do It Yourself

Try the **statements** below using the same `x` and `y` variables from the last DIY box. Try to predict the result before you hit enter.

```
y === x
y === "50"
y !== x
y !== "50"
```

Explain to yourself what is happening in each case, and why each statement is either `true` or `false`.

Because of the confusion that implicit type casting can cause, it is often considered best practice to use === and `!==` whenever possible and try to avoid using == and `!=`.

Java Connection

Boolean Operators. JavaScript adds two new **Boolean** operators, === and `!==` for "is exactly equal to" and "is not exactly equal to".

Java Connection

String Comparison. In **Java** you write `x.equals("hi")` to find out if the **String** `x` contains the word "hi". In **JavaScript** you just use `x === "hi"`.

1.2.2 Explicit Casting

Sometimes you really want to treat a String value as a Number. For example, the prompt function always returns a String, which works fine in most cases, but suppose we are asking the user to enter a number and then we want to add 1 to it. The code in the DIY box below won't do it.

Do It Yourself

Try this in a `<script>` element or a Chrome console window. What is going wrong?

```
var y=prompt("Enter a number");
y = y + 1;
alert(y);
```

Java Connection

Explicit Typecasting. In **Java** this can often be done with a casting operator. For example `(int)23.5` converts the `double` `23.5` to the `int 23`. There is no such mechanism in **JavaScript**. Instead you use the `parseInt` **function**.

The example in the DIY box above doesn't do what we want because y is a string. So JavaScript treats the + operator as concatenation (much like Java). In this case, we need to use **explicit typecasting** to force the type we want. JavaScript has built in **global** `parseInt` and `parseFloat` functions for this very purpose.

(Unlike `alert`, `prompt` and `confirm` which belong to the `window` **object**, `parseInt` and `parseFloat` are **global functions**. They do not belong to any object in the system. But this doesn't usually make a difference in practice.)

Do It Yourself

Here's the fix for the last example. Verify that it works and that you understand why it works before moving on.

```
var y=prompt("Enter a number");
y = parseInt(y)+1;
alert(y);
```

But what if `parseInt` or `parseFloat` fail (i.e. the user typed something that can't be **interpreted** as a number)? In that case, they return the special value NaN (Not a Number) which can be detected with the global Boolean function `isNaN`. So to be fully robust, you could do the following:

Do It Yourself

Try this out in a `<script>` element or a JavaScript console. Can you explain what each line is doing?

```
var n;
do {
    n = prompt("Enter a number");
} while (isNaN(n));
n = parseFloat(n) + 1;
alert(n);
```

JavaScript always tries to do the best it can to avoid crashing. So `parseInt("50.5")` will work and return `50`. So will `parseInt("50x6")` (the parser just stops at the x and returns what it has so far). This can be either a blessing or a curse depending on what the you are trying to do.

<exercises section='1.2.2'>

1. Load **whatswrong.html** file from the **example pack**, run it to see what it does, then view the page source and read the instructions in the comment header of the file. Explain exactly where the error or errors happen, and fix them.

2. Further fix the code from question 1 so that it recovers gracefully when the user types a bad number.

</exercises>

1.2.3 Dynamic Typing and Arrays

Like most languages, JavaScript contains support for **arrays**. In many cases the code for processing arrays will look very familiar. Things are only slightly different in JavaScript because of the consequences of the language being dynamically-typed.

> ***Do It Yourself***
>
> As a Java programmer, you ought to be able leverage your understanding of Java arrays and your growing understanding of JavaScript to describe the effect of this JavaScript code:
>
> ```
> var a = [3, 4, 5];
> for(var i = 0; i < a.length; i++)
>     a[i] = a[i] * 2;
> ```
>
> Take a minute to think about what this code does, then type or paste it into a browser console window to run it, and then type a and press enter to examine the contents of the **array**. Did you get it right?

The first consequence of JavaScript's dynamic typing is that you don't have to declare a type for an array. The following pieces of code show two different ways of creating a generic empty array.

```
var a = [];
var a = new Array();
```

You can also create an array with some initial contents, like this:

```
var initializedArray = [43, "hello", -2.5,
true];
```

The expression in square brackets above is called an array **literal**. In JavaScript, you can use array literals anywhere you could use an array.

> ***Java Connection***
>
> **Array Literals.** In **Java**, **array literals** are written with curly brackets and can only be used as part of a combined **declaration** and assignment, like this: `int[] x = {5,3,2};`. In contrast, **JavaScript** array literals are written with square brackets and can be used anywhere you could use an array. For example the expression `[-10,3,0][1]` evaluates to 3. Can you figure out why?

Notice that this array contains values of three different types (Number, String, and Boolean). This brings us to the second consequence of dynamic typing: you can mix values of different types in a single array. In fact, you can even have arrays stored within other arrays.

> ***Do It Yourself***
>
> Type or paste this line into a browser console and hit enter.
>
> ```
> var a = [6, [5, 3, -2], "JavaScript"];
> ```
>
> How many items does the **array** a have in it? Type `a.length` to find out.
>
> What does each item contain? Type `a[0]`, `a[1]`, and so on to find out. What if you type `a[1][1]`? What do you expect to be the result? Try it to see if you were right.

You never have to declare the size of an array in advance because you can arbitrarily extend its length after you create it.

Do It Yourself

Type or paste the following into a browser console.

```
var a = [];
a[0] = -23;
a[1] = 45;
```

Now type a and hit enter to see the contents of the **array**. This is an example of extending the length of an initially empty array. Type a.length to see the new length.

It is also possible to create an array with "holes" in it.

Do It Yourself

Complete the previous DIY box, and then in the same console window, type the following:

```
a[9] = 0;
```

You just created an **array** with 7 "holes in" it from indices 2 to 8. How long is the array now? Type `a.length` to find out, then type a and hit enter to find out what the contents of the array looks like.

The "holes" in the array have been filled with the special **global property** `undefined`.

So if you are ever not sure whether an array index has been given a value, you can test it in an if statement. Try this:

```
if (a[3] === undefined) alert("index 3 is undefined");
```

Java Connection

Arrays. JavaScript arrays are a lot like **Java** arrays, except: 1. You don't need to specify the type or size of the array; 2. You can mix **data types** in the same array; 3. You can increase the size of the array after it has been created; and 4. The array indices you fill with **values** don't have to be **contiguous**.

<exercises section='1.2.3'>

1. Create a page that prompts the user for 10 numbers and then computes the average **value** of the numbers entered and reports in an `alert` box how many of the values they entered were above average.

2. Create a page with a script that defines an **array** and puts the **String** values `"Hello"`, `"World"`, and `"!"` into **random** array locations between 1 and 10. Then prompt the user for three integers and check to see if they have found the array locations where the three String values are stored. Give them a score out of 3 depending on how many they found and show them the values they uncovered using `alert` boxes.

</exercises>

2. The `document` Object and the DOM

After the previous chapter, you should be able to write **JavaScript** programs that are integrated with a web page, communicate with the user, and solve problems. Congratulations! You still have a long way to go.

Up to this point, your interactions with the user have entirely made use of pop-up dialogs. While this style of communication is sometimes used in **web apps**, most input to a JavaScript program usually consists of mouse and keyboard **events** (i.e. moving the mouse, clicking, and typing) and most JavaScript output consists of modifications to the web page (e.g. changing colors and styles, revealing and hiding menus, changing the contents of an **element**, loading and displaying new content, etc.). We'll start with the output first in this chapter, then the next chapter will show you how to respond to keyboard and mouse input. By the end of these two chapters, you'll have the tools you need to write much more professional web apps.

2.1 The Document Object Model

Every **JavaScript** program running in a web browser has access to a `document` **object**. This object holds the browser's internal representation of the **Document Object Model** (**DOM**), and contains all the information the browser constructs using the **HTML tags** and **attributes**, **CSS style rules**, images, and other components that make up the source code of the page. Understanding the DOM is key to becoming an effective JavaScript programmer.

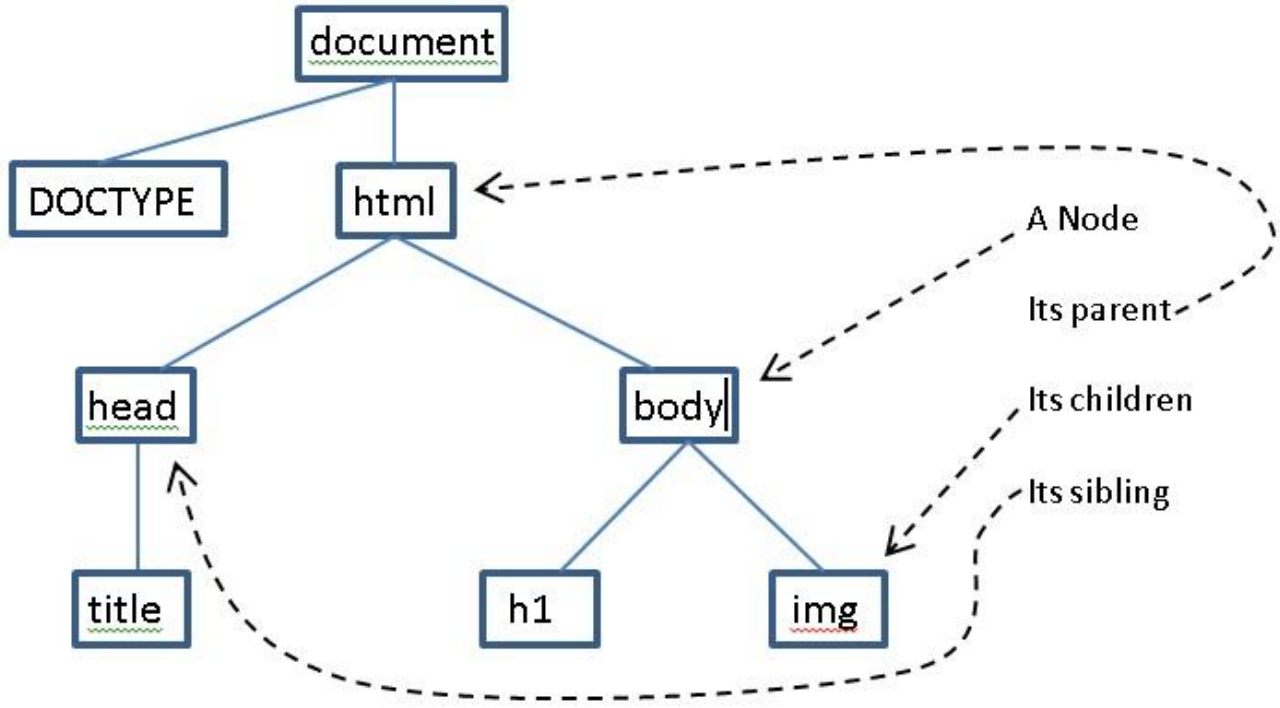
Like any bracketed structure, an HTML page can be viewed as a hierarchical family **tree** of **elements** containing other elements. When the browser reads an HTML source page it constructs an object for each element, links it to the elements it contains (the "children") and also links it to the element that contains it (the "parent").

Do It Yourself

Load **helloworld.html** from the **example pack**, and look at the page source:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <h1 id='message' class='heading'>Hello, World!</h1>
        <img src='images/smiley.jpg' alt='smile image'>
    </body>
</html>
```

In the example above, the document consists of a `<!DOCTYPE>` element and an `<html>` element. The `<html>` element contains a `<head>` and a `<body>`. The `<head>` contains a `<title>` element and the `<body>` contains an `<h1>` element and an `<img>` element. This set of relationships can be displayed in a tree diagram like the one shown below.



The items in the boxes are referred to as **nodes**. Each node is a JavaScript object that has been constructed by the browser to represent the corresponding **HTML element**. Each node object contains information about the attributes, CSS style and contents of the element it represents.

A node can have one parent above it, any number of children directly below it, and any number of siblings (nodes with the same parent). The root node is at the top and the leaf nodes are the ones at the bottom with no children, which makes this a curious sort of upside down "tree". When you are using JavaScript in a web page, the `window` **property**, `document`, holds the root node of this tree. Like all properties and **functions** belonging to the `window` object, you can access `document` on its own or by typing `window.document`.

Do It Yourself

Open **helloworld.html** from the **example pack**. Hit F12 and instead of the Console tab, go to the **Elements** tab. This is a representation of the browser's **DOM**.

Now right click the `<img>` element, select "add **attribute**", and give this image a style attribute (e.g. `style="width:100px"`). You should see the change reflected on the page immediately.

Now check the page source, and you will see that it has not changed. Why not?

Now go to the Console, type `document`, and hit enter. Click the response to open the DOM and explore it. This `document` **object** gives you access to the browser's DOM from within a **JavaScript** program.

There is a lot more to say about the structure of the DOM than this, but the information in this section is good enough to get you started.

2.2 What is a JavaScript Object? (An Important Aside)

The last section introduced the notion of an **Object**, so we better say a few words about that before we move on.

Just like in **Java**, a **JavaScript** object is a package of **variables** (or "**fields**" or "properties") and code (or "**methods**") stored together under a single variable name. In your experience with Java you have probably used

objects before, and you may even have created your own objects.

For example, in JavaScript every **String** object has a field named `length` and a method named `charAt` (just like the Java `String` object). The code in the DIY box below creates a string and displays its length and first character. Note the use of the variable name, `s`, with the dot operator to access the variables and methods that belong to the object referenced by `s`.

Java Connection

Objects. JavaScript makes extensive use of **Objects**, and supports the creation of custom objects and an **object-oriented** style of programming. But JavaScript also supports other programming styles (e.g. **imperative**), and as you will see later, JavaScript objects are different in many ways from their **Java** counterparts. But for now you can treat JavaScript objects just like Java objects.

Do It Yourself

Try this in a console window or a `<script>` element of a web page. Predict the result of each line before you run it.

```
var s = "JavaScript is Cool. ";
alert(s.length);
alert(s.charAt(0));
```

Java Connection

Char versus String. Unlike **Java**, there is no `char` **data type** in **JavaScript**.

The second line in the code above accesses the `length` field associated with the object `s`. The third line calls the `charAt` method with the parameter 0 to get the first character from the object `s`. The `charAt` method returns a string of length 1.

Java Connection

Object Syntax. There are two ways of accessing an object's **fields** and **methods** in **JavaScript**. Like **Java**, you can write `objectName.fieldName`. Unlike Java, you can also write `objectName["fieldname"]`.

One thing that is a little different about JavaScript is that you can use either dot notation or square bracket notation to access an object's properties and methods. For example, instead of `s.length` and `s.charAt`, you can write `s["length"]` and `s["charAt"]`, as shown in the DIY box below.

This square bracket object notation looks and behaves exactly like a data structure called an **associative array** (or sometimes a "map" or a "dictionary"). Indeed, there is really no difference between an object and an associative array in JavaScript, so both styles of referencing are included for the convenience of the programmer.

Do It Yourself

Try this in a console window or a `<script>` element of a web page.

```
var s = "JavaScript is Cool. ";
alert( s["length"] );
alert( s["charAt"](0) );
```

This square bracket style of accessing an object's fields and methods might seem strange, but it has its uses. For example, try the following (after typing the lines above). Make sure you type something legal like "length" when prompted.

```
var fieldName = prompt("type a field name");
alert( s[fieldName] );
```

2.3 Retrieving and Manipulating a DOM Node

There are lots of ways to access and dig through the **DOM tree** from within a **JavaScript** program, but by far the easiest way to retrieve a **node** is by using its `id` **attribute**. To do this, use the `getElementById` **method** of the `document` **object** to reach into the DOM and grab the node you want (note the lower-case d in that method name!). Once you have it, you can access and/or change the contents of the **element**, its style information, or any of its attributes.

Any element you retrieve using `document.getElementById` will be an object of type `Element` (and also of type Node). JavaScript **Element objects** contain a number of **fields**, each containing information associated with the corresponding **HTML element**. These fields can be used to make changes to the DOM long after a page has been loaded.

Here are some of the more useful `Element` fields...

`innerHTML`:	a String representing the contents of the node as HTML
`style`:	the CSS style information associated with the node
`className`:	the HTML class attribute
plus...	there will be one field for every attribute specified in the corresponding tag in the original HTML source code (e.g. `id`, `src`, `href`, `value`, `type`, etc.)

Do It Yourself

Open **helloworld.html** example from the **example pack**, go to the console and try the following commands, noting the return values in each case.

```
var e = document.getElementById("message");
e;
e.innerHTML;
e.style;
e.className;
```

Now add an `id` **attribute** to the `<img>` **element**, either by changing the source code, or by changing the **DOM** through the Elements view (see a previous DIY box).

Now use **JavaScript statements** similar to the ones above to retrieve the `src` and `alt` attributes of the image.

An alternative to the `getElementById` method is `querySelector`. You can use `querySelector` with any **CSS selector** and it will return the first matching element (or `null` if no elements match).

For example, `document.querySelector("h1")` returns the first `<h1>` element in the DOM, `document.querySelector("h1.main")` returns the first `<h1>` element with a **class** of `main`, and so on. And of course, `document.querySelector("#myId")` is equivalent to `document.getElementById("myID")`.

Do It Yourself

Open the browser console for this page (the one you're reading now) and try the following to get the first DIY box and the first Java Connection box.

```
document.querySelector("div.DIY")
document.querySelector("div.JC")
```

The `querySelector` method can sometimes be a useful tool to have in your toolbox, but to keep things simple, I will continue to just use `getElementById` in the examples that follow.

2.3.1 Changing an Element's `innerHTML`

An `Element`'s `innerHTML` **property** contains a **String** representation of the content of the corresponding DOM node. The content is everything that appears between the opening and closing tags that defined that node in the original HTML file. Changing this string is probably the easiest way to rewrite part of the DOM.

The DIY example below shows you how to change the text contents of an element.

(Be careful with case here. If you type `innerHtml` instead of `innerHTML`, the command may fail without even giving you an error message!)

Do It Yourself

Load **innerHTMLExample1.html** from the **example pack**, and let it do its thing. After you press OK, you will see the text of the heading change.

The relevant code is shown below:

```
<h1 id="heading">A Simple Example</h1>
<p>Here is some text in a paragraph.</p>
<script type="text/javascript">
    alert("Press OK to see me change my own heading.");
    document.getElementById("heading").innerHTML = "Done";
</script>
```

See if you can rewrite this code so that instead of just rewriting the heading, it also rewrites the `<p>` **element**.

You can also place **HTML tags** into the `innerHTML` field of a node. When you do that, the browser will read the tags, create node objects for the corresponding HTML elements, and add them to the DOM, as demonstrated in the DIY box below. If the `innerHTML` of the element you rewrite already contained other HTML elements, the corresponding nodes will be discarded from the DOM and replaced with new ones.

Do It Yourself

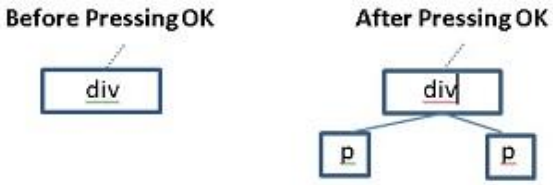
Load the file **innerHTMLExample2.html** from the **example pack** into a browser, and let it do its thing. After you press OK, you will see the div change to have two new paragraphs added as contents. Hit F12 and go to **Elements** to verify that two new **nodes** have been added to the **DOM**.

The relevant code is shown below:

```
<div id='target'>Here is some text in a div.</div>
<script type='text/javascript'>
    alert("Press OK to see me change the contents of my div.");
    document.getElementById("target").innerHTML =
    "<p>First paragraph</p><p>Second paragraph</p>";
</script>
```

Rewrite the code so that it asks the user for some HTML and then inserts whatever the user types into the `<div>`.

Here is a diagram of the relevant part of the DOM before and after executing the script in the DIY example above.



Java Connection

Bad Field Names. In **Java**, if you mistype an object's **field** name, you will get a **syntax error** every time. In **JavaScript** if you try to read a non-existent field, you will get a runtime error in the console. But if you try to assign to a non-existent field, JavaScript will create a new field for you without any errors or warnings. This makes JavaScript **objects** very flexible, but it also makes some errors very hard to find.

Note: Adding Fields to Objects

Unlike **Java**, JavaScript allows you to create a new field in any object at any time simply by assigning something to a new field name. This is very convenient, but it also means that some errors are hard to catch. If you misspell a field or get the case wrong JavaScript will modify the object by adding a new field without warning you about what has happened.

For example if you type `node.innerHtml = "new content"`, the `innerHTML` field of the node object will not change because you got the case wrong on its name. But you will not get any error or warning messages either since JavaScript just adds a new `innerHtml` field to the node object.

<exercises section='2.3.1'>

1. **From the first DIY box above:** Modify the code for **innerHTMLExample1.html** so that it rewrites the <p> element as well as the <h1> element.

2. **From the second DIY box above:**

   a. Modify **innerHTMLExample2.html** so that it prompts the user for some text, and then changes the contents of the to match what they typed.

   b. Test your code and try entering HTML at the prompt and investigate what happens to the DOM.

   c. Why do you have to create the script after the element? What happens if you place it earlier in the file?

   d. Modify the script again so that it asks the user to enter a positive integer. Then alter the contents of the element to display the integers from 1 up to the number the user entered. Each integer should be contained in a separate <p> element. (Note this will require some string processing with a loop.)

3. Get **addition.html** from the **example pack**. Read the comment header in the page source and follow the instructions to complete the program.

4. Create a new page using **template.html** from the example pack (or the default **NetBeans [https://netbeans.org/]** template). Place a <script> **element** after the to put a **JavaScript** program. This program should ask the user how many paragraph elements they want, then ask them to type a sentence to use as the content of those paragraphs. Then change the contents of the element according to what the user entered. Right click the page and choose "inspect element" to make sure that the paragraphs got created correctly.

</exercises>

2.3.2 Changing an Element's `style`

Changing an element's CSS information can be done most easily by accessing the `style` field of the corresponding DOM node. This `style` object contains all the in-line CSS information for the Node, where each field of the object corresponds to a CSS property.

Do It Yourself

Load **innerHTMLExample1.html** from the **example pack**, and press OK when prompted. Now hit F12 to open the console and type the following:

```
var e = document.getElementById("heading");
e.style.color = "red";
```

Now try changing other CSS styles in the same way. Add a border, change the font, position the header below the paragraph, etc.

Now go to the **Elements** view and look at the <h1> element. You will see style information in the element's `style` **attribute**. Note that the page source does not contain this information.

In addition to assigning or changing style information, you can also read style information from the fields of the `style` object, but you cannot read any CSS properties that were defined as part of a **style sheet** in this way.

Do It Yourself

Load **innerHTMLExample2.html** from the **example pack**, and press OK when prompted. Note that the `<div>` element is red with a border.

Now hit F12 to open the console and type the following:

```
var e = document.getElementById("target");
e.style.color;
e.style.border;
```

Why does one of the above lines give you a **value** but not the other? View the original page source and see if you can figure it out.

The only wrinkle in accessing the style object above is that you can get into problems with the dot notation for hyphenated style properties like `background-color`. The problem is that the JavaScript expression `e.style.background-color` looks like we are subtracting the **variable** color from the field `e.style.background`. The solution implemented in most browsers is to convert hyphenated-property-names to **camelCasePropertyNames** (e.g. `border-radius-top-left` becomes `borderRadiusTopLeft`).

Do It Yourself

Load **innerHTMLExample1.html** from the **example pack**, and press OK when prompted. Now hit F12 to open the console and type the following:

```
var e = document.getElementById("heading");
e.style.backgroundColor = "red";
```

Now change some other hyphenated properties, like `font-size`, `border-radius`, etc.

Note: Use of Square Brackets

Don't forget that you can access an objects' fields using the dot notation or **associative array** notation (see the section on JavaScript objects above). Many JavaScript programmers choose to use this notation for style properties. So in the most recent DIY box above, instead of `e.style.backgroundColor` many JavaScript programmers would prefer `e.style["backgroundColor"]`. It's up to you which method you use to access the `style` object's fields (or any object's fields for that matter) but you need to know about the two alternatives in order to read other people's code effectively.

Do It Yourself

Go back and redo all the other DIY boxes in this section, changing the dot notation to **associative array** notation for all style properties. Every example should behave in exactly the same way as before.

<exercises section='2.3.2'>

1. Create a web page with a single "hello world" paragraph. Create a script on the page after this **element** that prompts the user for a color, and then sets the color of the "hello world" element according to what they typed. Why do you have to create the script after the <p> element? What happens if you place it earlier in the file?

2. Modify the script from question 1 so that the user can specify both the style **property** to change and the **value** (note that you will have to use square bracket notation to access the style property the user chooses).

3. Go back to the **addition.html** exercise from the previous section. Change the file so that the result **field** changes to Green if the user gets your question right, otherwise it should be Red.

4. Modify **addition.html** again so that if the user fails to enter a valid number, the <h1> element turns into a solid red box with a thick black border around it.

5. Get the **moveme.html** file from the **example pack**. When you run it you will see a circle (actually a element) inside a box (another element). Both of these elements have their CSS `position` property set to `absolute`. If you are not familiar with how **absolute positioning** works, you can read about it in the comments of the moveme.html file or in the **W3Schools CSS Positioning** page.

    a. Your first task is to add a <script> element that will repeatedly use a `confirm` dialog to ask the user if they want to move the box. If they click OK, move the circle to a **random** position within the box and then ask again. if they click CANCEL, exit the program.

    b. Your second task is to change the program so it uses a `prompt` dialog to ask the user to enter U, D, L, R or Q (for up, down, left, right and quit) and then move the circle 20 pixels in the desired direction.

    c. Finally, modify your solution so that the user cannot move the circle out of bounds.

</exercises>

2.3.3 Changing an Element's `class` Name

Sometimes you may want to change a large number of property/**value** pairs at once. In this case, a good option might be to define styles for two different classes, and then change the `className` field of the element. This will automatically update the style of the element to match its new class, effectively changing many styles at once.

Do It Yourself

Load **classNameChangeExample.html** from the **example pack**. Now hit F12 to open the console and type the following:

```
document.getElementById("maindiv").className="classtwo"
```

You should see a big change. Take a look at the source code for this file to see how this change happened. Can you use a single command to change it back to the way it was originally?

(Advanced Programming Note: If you need an object to have multiple classes associated with it, you can use the `classList` **api** for that object. You won't find this on w3schools, so here's a link to the **Mozilla Developer Network ClassList [https://developer.mozilla.org/en-US/docs/Web/API/Element.classList]** page.)

<exercises section='2.3.3'>

1. Go back to the **addition.html** exercises from the previous sections and create **CSS rules** for two **classes** named `correct` and `incorrect`. The `correct` class should make an **element** big, centered in the middle of the screen, with a border and a happy background color. The `incorrect` class should make an element small, positioned in the lower right, with sad colors. Then modify the script to change the class of the result element appropriately depending on whether the user gets your question right or wrong.

</exercises>

2.3.4 Changing Other Attributes

You can also access, change or add any other attributes of an element by accessing the fields for those attributes. For example, if you want to change an image, you can modify its `src` attribute by accessing the `src` field of the corresponding node. Or if you want to change a link, you can modify the `href` attribute of the corresponding node.

Do It Yourself

Open the chrome console for this page and execute the following command to retrieve the node corresponding to this DIY box:

```
node = document.getElementById("testDIY");
```

Now add a `title` attribute:

```
node.title="Read Me!";
```

Now hover the mouse over this DIY box to see the effect of this **statement**. Then right-click and choose "Inspect **Element**" to see the **attribute** you added to the **DOM**.

This link goes to Google. It has an `id` of "testLink". Can you enter commands in the browser console to redirect the link to **facebook [http://www.facebook.com]** instead?

<exercises section='2.3.4'>

1. Get the file **mood.html** from the **example pack** and follow the instructions in the comments at the top. Here are the image files you will need (right click and save them to an appropriate folder):

</exercises>

2.4 Retrieving and Manipulating a List of Nodes

If you are comfortable with indexed structures such as **arrays**, you can use several other powerful `document` **methods**, including `querySelectorAll`. This method behaves just like `getElementById`, except that instead of passing it an `id` **attribute value**, you pass it a **CSS Selector** and it returns a **NodeList object** (an array-like structure) containing every **element** that matched the selector.

Do It Yourself

Go to the Sheridan College Wikipedia page, open the JavaScript console of your browser, and execute the following:

```
document.querySelectorAll("p");
```

You should get a **NodeList object** returned that contains all the <p> elements in the document. To get a particular one, use square brackets like this:

```
document.querySelectorAll("p")[2];
```

Or like this:

```
var paragraphs = document.querySelectorAll("p");
paragraphs[2];
```

Now try to get all the elements with the class name "body":

```
document.querySelectorAll("p.body");
```

These methods can be handy if you want to make a change to a bunch of elements at once. Once you have them in a NodeList you can use a **loop** to change them all.

Do It Yourself

If you are reading on line, open the JavaScript console. Or you could go to a Wikipedia page and open the console there. Execute the following code:

```
var a = document.querySelectorAll("p");
for (var i=0; i<a.length; i++)
    a[i].innerHTML="JavaScript Rules!";
```

Did the code do what you expected? Can you figure out how to select all the Do It Yourself boxes on this page and change their background color to red?

<exercises section='2.4'>

1. Go back to your solution to the multiple paragraphs exercise from section 2.3.1 and modify it so that after the paragraphs have been created, the user is asked for a CSS **property** and a **value** in two separate prompts, and then the given property and value get set for all the paragraphs. For example if the user types "color" at the first prompt and "red" at the second, all the paragraphs should turn red. Repeat this until the user enters "quit" instead of a property name.

   If you don't yet have a working solution the the multiple paragraphs exercise, you can get **paragraphs.html** from the **example pack** - then you can at least complete the work in this question for a fixed number of paragraphs.

</exercises>

3. Functions and Events

In the previous chapter you learned how to manipulate the **DOM** to produce different kinds of output from a **JavaScript** program. But you are still very limited in the kinds of input you can get from the user. This chapter will begin to change that.

3.1 The Basics

Up to this point, you have only been using **JavaScript** on your web pages in an **imperative programming** style, running scripts within a `<script>` **element** as the page is loading. This is pretty limiting. Usually you want to make changes or interact with the user after the document is loaded, not while it's loading. To do this, you need to **declare functions** while the page loads and then execute them later in response to specific **events**.

3.1.1 Declaring a Function

JavaScript functions are a lot like the methods, procedures, and functions found in most other languages. Here's an example of a JavaScript function declaration.

```
function hello() {
    alert("Hello, world!");
}
```

> ***Java Connection***
>
> **Methods. JavaScript functions** are very similar to **Java methods** although, as we shall see, there are also some very important differences.

The above function has no parameters and no **return value**.
Note that the **syntax** is almost identical to **Java method declaration** except that you use the keyword `function`, and you don't have to specify a return type. You also don't need to specify an **access level** (public, private, etc.) or whether the function is **static** or not – none of these things apply in JavaScript.

Functions can be defined anywhere on the page, as long as they are inside a `<script>` element, but it's good practice to define your functions in a `<script>` element at the top of the document, inside the `<head>` element. It's even better practice to define your functions (and the rest of your code) in a separate file or multiple separate files with a `.js` extension. Then you can load these files using the `src` **attribute** of an empty `<script>` element, like this:

```
<script type="text/javascript" src="js/hello.js"></script>
```

Note that a `<script>` element can be used in two ways - either you can put JavaScript code between the opening and closing tags, or you can leave the element empty and use a `src` attribute to load JavaScript code from an external file. But you cannot do both in the same `<script>` element. If you want to load some code from a file and put some directly on the page you will need two `<script>` elements.

3.1.2 Responding to an Event

No matter where you define a function, it won't execute unless you call it. Most JavaScript function calls are triggered in response to browser events. An event is an important milestone in the life of a page. Events are triggered whenever an element has finished loading, the user has resized the browser, an element has been clicked, the mouse has entered or left the page, a key has been pressed and so on.

Any **HTML element** can be the source of an event. By default most events are ignored, but for any particular event that might occur to any particular element, you can give the browser a block of JavaScript **statements** to execute when the event occurs. These statements can include a call to the function you defined.

Here's an example:

```
<p onclick="alert('You clicked me!');"> Click Me </p>
```

> ***Java Connection***
>
> **Events.** In **Java** GUI programming, you define listener **classes** which contain **event** handlers (i.e. **methods**). In **JavaScript**, you just define the handlers (i.e. **functions**).

The HTML code above creates a paragraph element with the text "Click Me". The `onclick` attribute contains a single JavaScript statement that should be executed when a click event happens (i.e. when the user clicks on the paragraph).

Note that the JavaScript statement has to be enclosed in quotation marks, like any attribute **value**, and that because of this, the alert command has to use single quotes around the text in order to be **syntactically** correct. Alternatively, since HTML and JavaScript both treat double and single quotes interchangeably, you could have written this:

```
<p onclick='alert("You clicked me!");'> Click Me </p>
```

You can put as many JavaScript statements as you want into the `onclick` attribute, as long as they're separated with semicolons. But if you need a lot of code there, you should probably define a function and put a call to it into the `onclick` attribute instead. The example below executes the `hello` function defined in the previous section when the heading is clicked.

```
<h1 onclick="hello();">Click Me</h1>
```

Other JavaScript mouse events are listed below.

`ondblclick`:	Triggered when element is double-clicked
`onmousedown`, `onmouseup`:	Triggered when a mouse button is pressed or released over the element
`onmouseover`, `onmouseout`:	Triggered when mouse is moved into / out of the bounding box of an element
`onmousemove`:	Triggered every time the mouse moves (even one pixel) while it is over the bounding box of an element. Watch out, this generates a lot of events!

> ***Do It Yourself***
>
> Load **functions1.html** and **functions2.html** from the **example pack** to see the above examples in action. The two pages have the same behavior but differ in where the `hello` **function** is defined. One has it in a `<script>` tag, while the other has the function loaded from a separate file.

<exercises section='3.1.2'>

1. Edit **functions1.html** (or **functions2.html**) and alter it so that instead of popping up an alert when the heading is clicked, the user is asked for their name and then the message "Hello name" is displayed in the `<p>` **element**. Make it so that each time a new name is entered, a new message appears without erasing the old one.
2. Try making the element respond to a different **event**, such as `onmouseover` or `ondblclick`.

</exercises>

3.1.3 Clicking on Buttons

Clicking on text is all very well, but we should really be clicking on buttons. Clicking on buttons can be more intuitive to users, and the default style of buttons (color, text, size) changes on on mouse-overs or mouse-clicks to make them appear more active.

You can define a button like this, where the `value` attribute holds the button text:

```
<input type="button" value="Press me">
```

To make it respond to clicks, just add an `onclick` attribute, like this:

```
<input type="button" value="Press me" onclick="hello();">
```

> ***Do It Yourself***
>
> Load **functions3.html** from the **example pack** to see the above example in action. Can you change the code so that the button text changes when it is clicked?

<exercises section='3.1.3'>

1. Create a web page that asks the user a true or false question and give them a button to click for each option. Display a success or fail message somewhere on the page in response to their click.

</exercises>

3.1.4 Clicking on Text

Another option to give the user a better `onclick` experience is to use a `<span>` element with the CSS `cursor` **property** defined (and also possibly some `:hover` styles). This will style the click-able area like a link and make it more obvious that it can be clicked.

> ***Do It Yourself***
>
> Load **functions4.html** from the **example pack** and click where it says "click me". Here is the relevant HTML code:
>
> ```
> <h1 id="heading">
> You can
>     
>         click here
>     
>     if you want.
> </h1>
> ```
>
> The `clickable` **class** has **CSS style rules** that change the cursor and add other hover effects.
>
> See the **W3Schools CSS cursor Property** page and the **W3Schools CSS :hover Selector** pages for more info.

3.2 Parameters, Global Variables and Return Values

Just like Java methods, functions can have parameters and return values. But in JavaScript, there is no need to declare any types for anything. Here's an example:

```
function foo(x, y, z) {
    return x+y+z;
```

```
}
```

The **function** above takes three parameters and returns the result of "adding" them together. What gets returned depends on the type of what was passed in.

If a function finishes without encoutering a return **statement**, it will return the special **JavaScript value** `undefined` by default.

> ***Java Connection***
>
> **Parameters and Return Values. JavaScript functions** can have parameters and **return values**, just like **Java methods**. But because the language is weakly typed, you don't have to define parameter or return types ahead of time. A function can also be inconsistent about what type it returns for different calls, or even whether it returns something or not.

> ***Do It Yourself***
>
> Open a browser and go to the console. Enter the foo function above (use shift-enter for multiple-line commands like this). Now try the following calls to the function foo.
>
> ```
> foo(1,2,3)
> foo(1,"2",3)
> foo(1,2,"3")
> ```
>
> Can you explain why the **function** returns different results for each call?

3.2.1 Variable Scope

> ***Java Connection***
>
> **Scope. Java** has block-level **scope**, meaning that any **variable** is **local** to the code block in which it was declared. If you **declare** a variable in a while **loop**, that variable is local to the while loop.
>
> **JavaScript** has **global** and **function** level scope, but not block-level scope. This means that if you declare a variable in a while loop inside a function, that variable can still be accessed anywhere within the function.

Variable scope refers to the parts of a program that can see a given **variable**. If a variable can only be accessed within a give code block, the variable is **local** to that code block. If it can be accessed anywhere in the program it is **global**.

Variables declared inside functions or listed as parameters are local to the function. These variables are created when the function is called and destroyed when it finishes. Variables declared outside of functions are always global, which means they are accessible to all functions on the page.

If a function contains a variable name that is the same as a **global variable** name, **references** to that name within the function will always be to the **local variable**.

> ***Java Connection***
>
> **Hoisting.** JavaScript also has a strange process called **hoisting [http://www.sitepoint.com/back-to-basics-javascript-hoisting/]** which is unlike anything in **Java**. In a nutshell, it doesn't matter where in the **function** you **declare** a **variable**, **JavaScript** always acts as if the **declaration** was at the top (i.e. it "hoists" the declaration to the top of the function). This means that you can use a **local variable** even before you've declared it.

> ***Do It Yourself***
>
> Load **functions5.html** from the **example pack** into a browser and press the buttons to see what they do. Take a look at the code. The buttons are all linked to a **JavaScript function** in their `onclick` **attribute**, but they each pass a different parameter to the function. The function accesses a **global variable** to keep track of the current count.

Do It Yourself

Here is an example from **functions5.html** showing global and local variables.

```
var current = 0;                    ← current is global
function add(inc) {                 ← inc is local
    var newVal = current + inc;     ← newVal local
    current = newVal;
    alert(current);
}
```

Load **functions5.html** from the **example pack** and press a button. Now open a console window (F12) and type the following expressions followed by enter:

```
current
newVal
inc
```

Explain the result returned from each expression.

3.2.2 Watch Out!!!

There is a big gotcha with variable **declaration** which we have been able to ignore until now, but it's time to point it out. The issue concerns the fact that you don't actually have to **declare** variables at all in JavaScript. Variables will automatically be declared for you the first time you assign something to them. But it is very bad practice to not declare a variable because if you don't, the variable will be created with global **scope**.

Do It Yourself

Change the code for **functions5.html** so that the two occurrences of the keyword `var` are removed. Now load the page in a browser and press a button. Then open a console window (F12) and type the following expressions followed by enter (try to predict the result in each case before you do):

```
current
newVal
inc
```

Explain why the results are different from the results in the last DIY box.

Automatic variable declaration can cause confusion, not to mention hard-to-find bugs (see the Java connection box below). But some of the problems go away if you make it standard practice to **always declare all variables using the `var` keyword**.

Java Connection

Undeclared Variables. Java requires that every variable must be declared before it is used. JavaScript, on the other hand, will automatically declare a new global variable for you if you forget to declare it yourself.

Because of this, some things that would be syntax errors in Java are legal in JavaScript:

```
int myvar = 0; // Java variable declaration.
myvsr = 10; // Error stops compilation.

var myvar = 0; // JavaScript variable declaration
myvsr = 10; // Creates new global variable. No error.
```

Comments. As shown above, you can add comments to **JavaScript** in the same way as you would in **Java** (`// ...` for single line comments, or `/* ... */` for multi-line comments.)

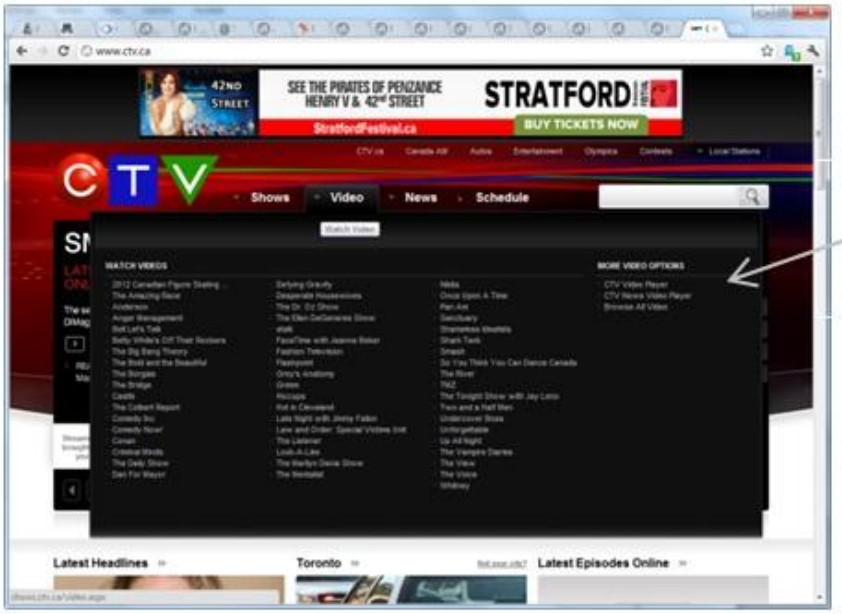
<exercises section='3.2.2'>

1. Open **messaging.html** from the **example pack**. Read the instructions in the comments that appear at the top of the page source and finish the **web app** according to these instructions.
2. Create a page with a single `<div>` **element** with a fixed width and height and a border around it. Include a `<p>` element underneath it with an `id` so you can use it as output. Use the `onmousemove` **event** on the `<div>` element to count the number of times the event is triggered and display the count in the `<p>` element.
3. Open **catchTheRabbit.html** from the example pack. Read the instructions in the comments that appear at the top of the page source and finish the web app according to these instructions. (Note that this web app has an accompanying image and CSS file which you must also download - you can get to these files by clicking on them in through "page source" on most browsers.)
4. Go back to the **moveme.html** file from the sample pack. In previous exercises, you used `prompt` dialogues to get the user to move the ball around in response to U, D, L and R. Now you should add buttons to the page for up, down, left and right and move the ball in response.

</exercises>

3.3 Application: Drop-down Menus

You see drop-down menus all over the web, on sites designed for both desktop and mobile viewing. Modern desktop sites often contain "mega-menus" that are rich with structured information and links, as shown below:



These mega-menus sometimes pop up when you click a button or tab, and sometimes when you mouse over a tab. To see how this is done, go to a site with a menu like this (e.g. **The Boston Globe [http://www.bostonglobe.com]** or **CTV [http://www.ctv.ca]**), right click the menu once it's popped up and "inspect **element**". You'll find it's just a `<div>` element (or similar) positioned cleverly, and then shown and hidden by a **JavaScript function** using either the CSS `visibility` or `display` **property**. Sometimes the JavaScript function also changes the style of the button to match the drop down menu style and indicate which menu is being displayed.

Do It Yourself

The file **dropDownMenu.html** in the **example pack** shows an example of drop down menus in action, but on a smaller page with source code that should be easier to read. Note that the "catch the rabbit" menu won't actually play the game unless you include a working `catchTheRabbit.js` file (the solution to a previous exercise).

Explore this file to see how the design works. Take note of the CSS styling used on the menus and buttons (in particular, the `cursor` **property**).

<exercises section='3.3'>

1. The file **sideMenus.html** in the **example pack** contains some starter code. When you view this file you will see what the page is supposed to look like when the user clicks menu 1. See if you can add **JavaScript** and make other changes so that when the page starts, the menus are not highlighted, but menu 1 pops up when the user clicks the button and hides itself again when the user clicks again. Now add the other two menus to **sideMenus.html**. The content of these menus is not important.

2. Now try to get the menus in **sideMenus.html** to respond to mousing over and mousing away from the buttons and/or menus. This is actually a little trickier to get right than the first version.

3. The file **dropdownMobileExercise.html** in the example pack (with its companion CSS file) contains some starter code for a simple mobile web site (view it in a narrow browser to see what it would look like on a phone). Add code to this site so that when it is loaded, the paragraphs and the list of links do not appear at first. When the user clicks on a heading, it should open the corresponding content. As a bonus see if you can get it to close the other paragraphs before opening the new one.

</exercises>

3.4 Forms and Functions

A **web app**, whatever job it is designed to do, needs to be able to have a dialog with the user. In previous chapters you learned several simple ways of interacting with the user on the client side using **JavaScript** popups, HTML buttons, and **events** like `onclick`. **HTML forms** provide a richer set of tools for getting user input.

You were introduced to the basic HTML form **elements** in your previous web design courses, but you may need a review. Try the activities below, and if you need any help brushing up on this stuff, read the **w3Schools HTML Forms** section. You might also want to look at the Forms section of the **w3schools HTML Events** page.

> ***Java Connection***
>
> **Graphical User Interface (GUI). HTML Forms** provide a set of GUI input components simlar to any other system you may have used in **Java**. For example, many `javax.swing` input components (check boxes, radio buttons, drop down lists, text input **elements**, etc.) have counterparts in HTML/**JavaScript**. And Java Android apps use **XML**, a **Markup Language** similar to like HTML, to configure their GUI interfaces.

<exercises section='3.4'>

1. Create an HTML page to display the "registration form" shown at right. A solution is given in **basicForm.html** from the **example pack**.

2. Create a simple form that (when scripted with **JavaScript**) would allow the user to convert Canadian dollars to another currency of your choice. Try to make good use of the number type and `<output>` **element** from **HTML5**.
3. Create a shopping cart confirmation page, according to the specifications given below. (In a later exercise, you will add JavaScript code to implement the "recalculate" button.)
    a. Present 2 items along with their prices, with text input elements to show the current quantities.
    b. Allow users to enter a promotional code in a text input element.
    c. Allow users to choose standard delivery ($4.99) or expedited delivery ($19.99)
    d. Allow users to separately select gift wrapping ($9.99), insurance (10% of the purchase price before tax and discounts), and rewards membership (free).
    e. Present the tax (13%) and total at the bottom.
    f. Give them 3 buttons: reset, recalculate, and submit

</exercises>

3.4.1 Responding to Form Events

Just like any other **HTML element**, a form element can be configured to execute JavaScript **statements** in response to various events.

Here is a partial list of events that form elements can respond to.

`onclick` :	Triggered when the user clicks the element
`onchange`:	Triggered when the value of the element changes (e.g. when the user presses enter or tabs away from the element)
`oninput`:	Triggered on every act of input (e.g. every key stroke)
`onmouseover`:	Triggered when the mouse pointer moves over the element
`onmouseout`:	Triggered when the mouse pointer leaves the element
`onfocus`:	Triggered when the user clicks or tabs to an element. When an element is selected, that element is said to have focus. Typically only form elements and hyperlinks can have the focus, and only one element at a time can have it.
`onblur`:	Triggered when the element loses focus.

You can process a form element event in much the same way as you process any other event, by including some JavaScript code in the attribute for that event, as shown below.

```
<input type="radio" onclick="alert('clicked!');">
```

The JavaScript code in bold above will be executed when the radio button is clicked.

Notice that the **string** passed to the alert **function** is written with single quotes instead of double quotes. This is so it can be placed within the double quotes required by the `onclick` **attribute** without causing a **syntax error**.

Aside: The Submit Event

One final event you may need to know about is `onsubmit`. This event applies to `<form>` elements, rather individual `<input>` elements, and it is triggered whenever a form is submitted. A form is submitted either when the user presses a `submit` button (if you have one) or when they press `enter` inside a text input element.

If the form's purpose is to send data to a server, you can use this event to do some JavaScript processing to make sure the data they entered is ok, then return `true` if you want to allow the user to submit the form, or `false` if you want to prevent it.

For client side web apps, you usually don't want the form to be submitted anywhere, which is why you don't specify an `action` attribute on the `<form>` element, and why you don't usually have a submit button. But with no `action` attribute, if the form gets submitted somehow, it will trigger a page reload which will reset the page and mess up the running of your app. To prevent this, you could just not bother with the `<form>` tags, or you could add `return false;` to the `onsubmit` event:

```
<form onsubmit="return false;">
```

3.4.2 Retrieving Form Elements

In order to properly respond to many form events, you need to be able to get the form elements from the **DOM**. You can do this by assigning an `id` attribute to each one and using `document.getElementById`. This is the best choice if you have elected not to use `<form>` tags (see previous section). But the `document` **object** also contains a `forms` **field** that lets you access forms and their elements using their `name` attributes. This is the standard **method** of form access in JavaScript.

Do It Yourself

Load **basicForm.html** from the **example pack** into Chrome and take a look at the code. Notice that the `<form>` **element** has the name **attribute** set to `form1`. Try the following in the Chrome JavaScript console:

```
document.forms
document.forms["form1"]
document.forms["form1"]["name"]
document.forms["form1"][0]
```

Make sure you understand the results before you move on. What is the difference between the result of the first and second expressions? What is the difference between the third and fourth expressions?

The third and fourth expressions above get you a particular **node** from the **DOM** (it's as if you had used `getElementById`). So you can manipulate it like this:

```
document.forms["form1"]["name"].style.backgroundColor="red"
```

Or like this

```
var node = document.forms["form1"]["name"];
node.style.backgroundColor="red";
```

Java Connection

Object Syntax. You can access the **fields** of a **JavaScript object** using dot notation just like in **Java**, but it's often considered best practice to use **associative array** notation.

Recall that when you access an object's fields, you can use either the dot operator or square brackets (i.e. `object.field` or `object["field"]`). It is considered good practice by many JavaScript programmers to use the square bracket (**associative array**) notation to access named elements from the DOM. This is because sometimes the names can contain characters that would cause problems when using the dot operator.

So to access a named input element from a named form, use the following formula, where `formName` and `elementName` match the `name` attributes of the form and element respectively:

```
document.forms["formName"]["elementName"]
```

To access an input element that is not part of a form, just give it an id and retrieve it using `document.getElementById`.

<exercises section='3.4.2'>

1. Create a simple form with 3 text input **elements**. Use the `onmouseover` and `onmouseout` **events** to temporarily change the background color of each text input element as the mouse glides over them.
2. Create a text input element with a light gray background. When the user clicks in the text input element to enter text, the background should change to white. When they leave the text input element, it should revert to light gray again.
3. Add to exercise 2 so that when the user changes the text input element, a thank you message appears either in a dialog box or somewhere in the HTML document.
4. Add a night/day button to **basicForm.html**. When clicked it should toggle between two different color schemes for the page – one dark and one light. Be sure to change form colors as well as.

</exercises>

3.4.3 Accessing Form Contents

To create a **client-side web application**, you need to be able to access and process the user input that is stored in the form. For almost all input elements (text boxes, numbers, dates, etc.) you can access the `value` attribute to find out what content the user has entered. This attribute always contains a string representing the data the user entered.

> ***Do It Yourself***
>
> Load **basicForm.html** from the **example pack** into Chrome, type something into the password field, then open a Javascript console and try this:
>
> ```
> document.forms["form1"]["password"].value
> ```
>
> Congratulations, you are now accessing user input. You can also change form contents like this:
>
> ```
> document.forms["form1"]["name"].value = "Type Here"
> ```

If the form element is of type `radio` or `checkbox`, it's a little more complicated because what you get back from `document.forms["formID"]["elementName"]` is an **array** of elements. You have to specify the index number of the particular element in the group. The first radio button in the group will be index 0, then 1, etc. You can access the `checked` field to get a **Boolean value** (`true` if the button is checked, `false` otherwise).

> ***Do It Yourself***
>
> Load **basicForm.html** from the **example pack** into Chrome. Then try this in the JavaScript console:
>
> ```
> document.forms["form1"]["interests"]
> ```
>
> You should get an array of checkboxes back. Now try this to get the second checkbox.
>
> ```
> document.forms["form1"]["interests"][1]
> ```
>
> Finally, try this:
>
> ```
> if (document.forms["form1"]["interests"][0].checked)
>     alert("I knew you liked gardening!");
> ```
>
> What happened? Probably nothing. Why? What do you have to do to make the alert box pop up?

<exercises section='3.4.3'>

1. Change **basicForm.html** so that when the form is first loaded, the text "Choose a User Name" appears in the text field. When the user clicks in the text field, this text should disappear. When they leave the text field, if they have typed something, it should stay. If they have not typed anything, the text"Choose a User Name" should be put back in the box. (Note that this is a good exercise for a beginning **JavaScript** programmer, but if you really want to implement this behavior in a text box, you're better off using the new HTML5 `placeholder` attribute.)

2. Change **basicForm.html** so that if the user tries to uncheck the "skiing" box, it pops up a confirmation dialog that asks "are you sure you don't like skiing?". If the user presses OK, the box should stay unchecked. If the user presses "cancel" to box should remain checked. Note that if the user unchecks the box, then re-checks it, no confirmation dialog should appear.

3. Change **basicForm.html** to add "check all" and "uncheck all" buttons to the interests section.

4. Replace the "check all" and "uncheck all" buttons from the last exercise with a new "all" checkbox. When selected, this checkbox should check or uncheck all the other boxes. When all interests are manually selected, the "all" checkbox should automatically check itself. If even one interest is deselected, the "all" checkbox should automatically uncheck itself.

5. One of the main uses for JavaScript used to be validating forms before they were submitted to the server (i.e. making sure all **fields** were filled in, checking that numeric fields contained numbers, etc.) These days most of this validation can be done by using the new **HTML5 input types** and **attributes**. But there is still a role for JavaScript. For example, suppose that the user of **basicForm.html** is only allowed to check up to 2 interests. Write a JavaScript **function** to check this condition and call it from the `onsubmit` **event** of the `<form>` **element** (`onsubmit="return yourFunction()"`). The function should pop up an error alert and return `false` if more than 2 interests are checked.

6. Now fix the form so that the user is alerted and prevented from checking more than 2 boxes.

</exercises>

3.4.4 Form-based Client-side Web Apps

When you harness the power of forms, you can create client-side web apps with almost the same input capabilities as any desktop GUI app.

An example of a very simple client-side web app is shown in the **additionQuiz.html** file from the **example pack**. This program contains two JavaScript functions. The first function creates and presents a new question to the user and clears out any old answers or feedback, while the second checks the user's answer and gives feedback. Each of these functions is called by a different button press.

Notice the use of the `<span>` elements with unique `id`s. They allow the values for the two operands to be set in a JavaScript function without rewriting the entire paragraph element that contains them.

Notice also the use of the `onload` event in the `<body>` element. This event is triggered after the `<body>` is finished loading, and calls the function to set the first question.

<exercises section='3.4.4'>

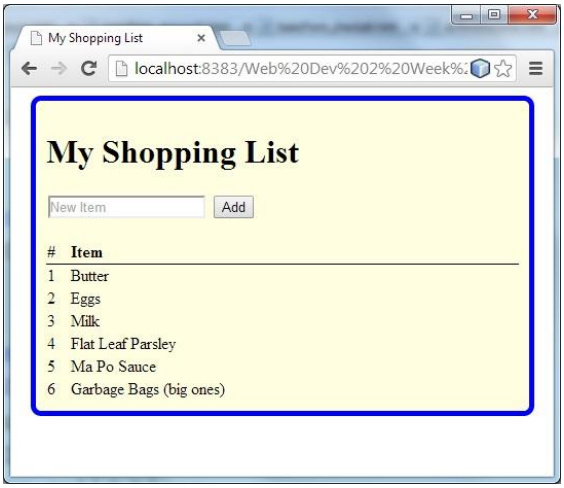
1. Get the **arithmetic.html** file from the **example pack**. This file is the interface for a very simple calculator with five **functions**. Add **JavaScript** code (as well as any necessary **HTML tags** or **attributes**) so that when the user presses the "Do It" button, the selected arithmetic is performed and the result is placed in the read only text box to the right of the equals sign.

2. Modify your solution from the last question so that any time the use changes any of the form **elements**, the result updates. You will need to use the `onchange` and/or `oninput` **events** for this. Now that the answer is automatically refreshing, you don't need the "Do It" button any more, so you can remove it.

3. Get the currency converter from the Section 3.4 Exercises working. When the user clicks "convert" it should display on the page the result of the conversion (you can look up the rate on line or just make something up if you like).

4. Get the shopping cart from the Section 3.4 Exercises working. When the user clicks "reset", they should be warned that the form is about to clear and get a chance to stop it. When the user clicks "recalculate", a function should be called that checks the quantities of the products and the various options the user has chosen, then displays the new total price with tax. Don't forget to use Math **methods** to round to two decimal places.

5. Make a simple shopping list creation app. It should give the user a text box and a button to press. When they press the button, it should add the text they entered to a `<table>` element somewhere on the page. It should be a numbered list with numbers added automatically. The first column of the **table** should contain item numbers and the second column should contain the items the user entered. Try to make it so that the user can also hit Enter to create a new list item.



6. Now make the currency converter from Exercise 3 better: Use fieldsets, styling, and images to improve the look of the currency converter; Add drop down lists to allow the user to pick the "from" and "to" currency, then perform different calculations depending on what they choose; Make the converter robust so that if the user enter a non-numeric **value** it recovers gracefully, and make sure it always shows two digits for the cents **field** (e.g. it shows "01" instead of "1").

7. Make the shopping cart from Exercise 4 better: Use fieldsets, labels, images and/or drop down menus to improve the look and feel of the shopping cart confirmation page; Add a text area for shipping instructions; Make it robust so that if they enter non-numeric values in number fields it recovers gracefully.

</exercises>

4. Graphics on the HTML5 Canvas

HTML5 brings with it a number of new **JavaScript APIs** that greatly increase the capabilities of JavaScript **web apps**. For example it is now possible to access a user's location, store data locally and respond to drag-and-drop **events**. One of the most important new JavaScript APIs is the new drawing API for the HTML5 `<canvas>` **element**. This API allows JavaScript programmers to create interactive graphics and games that can execute on any browser (functionality that previously required a bulky plugin like Flash).

The goal of this chapter is to give you all the raw materials you would need to implement a simple 2D game such as **Pong**. The chapter **focuses** on the basics, not the details. For full details, see the **w3schools Canvas Reference**.

4.1 Canvas Basics

To access the 2D drawing **API**, you need a `<canvas>` **element** somewhere on your HTML page. This element should have an `id` **attribute** as well as `height` and `width` attributes to specify its size in pixels, like this:

```
<canvas id='myCanvas' height='500' width='250'></canvas>
```

Note that `<canvas>` is not an empty element. You do need a closing tag. There is no need to include any content between the tags, though some developers put a message there which will display on older browsers, saying something like, "Sorry, your browser does not support the **HTML5 canvas**."

By default a canvas just appears as a blank space on your page. But if you style it with CSS to give it a border and background color, you will be able to see it.

There are two steps to drawing on a `<canvas>` element. First you have to retrieve the element from the DOM:

```
var c=document.getElementById("testCanvas");
```

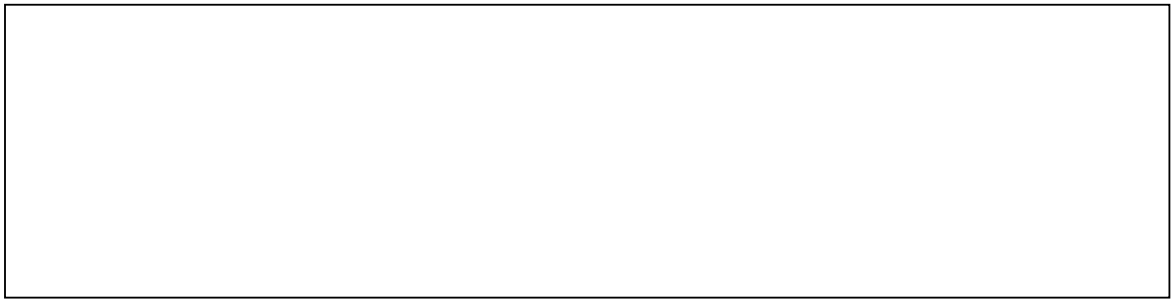
Once you have it, you can change its style and attributes like any other element. But if you want to draw on it, you need to retrieve a drawing "context". The simplest context available is the "2d" context:

```
var ctx=c.getContext("2d");
```

The above line retrieves a special "drawing context" **object** that implements the 2D drawing API for your canvas.

Do It Yourself

Here's an example **HTML5 canvas**, with a width of 400 pixels and a height of 100 pixels. It's CSS `width` **property** has been set to `100%` so that it will stretch to fit the available space. It also has a border and a background color of white specified in CSS, and it has an `id` of "testCanvas" so we can easily retrieve it from a JavaScript program.



In the JavaScript console of your browser, try the following commands:

```
var c=document.getElementById("testCanvas");
var ctx=c.getContext("2d");
ctx.fillStyle="red";
ctx.fillRect(125,25,150,50);
```

The first two lines retrieve the canvas and drawing context to **variables** named `c` and `ctx` respectively. The third line sets a drawing color. Colors can be specified using any CSS color **value**. The final line places a red rectangle in the middle of the canvas. The top left corner is at an `(x,y)` position of (125,25), the width is 150 and the height is 50 pixels.

Following the above example, enter two more lines in the **JavaScript** console to create a black rectangle that fills the entire canvas.

Draw two overlapping rectangles in different colors, but with their alpha values set so that you can see the first rectangle through the second.

Draw the red rectangle again, but use `(-40,-40)` as its `(x,y)` location. Can you explain the result?

<exercises section='4.1'>

1. Set up an HTML page for a **pong** game. Include a **canvas** with a black background, a heading and some instructional information.

</exercises>

4.2 Drawing Commands

There are two basic kinds of drawing supported on the 2D drawing context (hereafter abbreviated `ctx`): "stroke" draws outlines and "fill" fills them in.

4.2.1 Rectangles and Text

You have already seen the `ctx.fillRect()` **method**, but there is also a `ctx.strokeRect()` method that uses the same parameters to draw the outline of a rectangle instead of filling it in. There is also a `ctx.clearRect()` method that clears a rectangle to the CSS background color.

When you do "fill" drawing, the **canvas** uses the current contents of the `ctx.fillStyle` **field** to decide how to do the drawing. The `ctx.fillStyle` field can hold a color **string** (any legal CSS color **value** is supported) or a special **object** representing a gradient or a pattern (for more information on **gradients** and **patterns** see the

w3Schools Canvas Reference).When you do "stroke" drawing, the canvas uses the current contents of `ctx.strokeStyle`, which can take the same values as `ctx.fillStyle`. Another relevant field for stroke drawing is `ctx.lineWidth` which holds a number representing the thickness of the line to draw in pixels.

If you want to draw text, there are `ctx.fillText()` and `ctx.strokeText()` methods. You can set the font by assigning a string representing a CSS font to the `ctx.font` field.

Do It Yourself

Here's another **HTML5 canvas** with a `width` of 400 pixels, a `height` of 100 pixels and an `id` of "testCanvas2".

In the JavaScript console of your browser, try the following:

```
var c=document.getElementById("testCanvas2");
var ctx=c.getContext("2d");
ctx.strokeStyle="#FF0000";
ctx.lineWidth=10;
ctx.strokeRect(125,25,150,50);
```

Now add some text to the canvas:

```
ctx.fillStyle="#0000FF"
ctx.font="22px Arial";
ctx.fillText("Hello, world!",135,55);
```

Now try the following to clear part of the drawing:

```
ctx.clearRect(0,0,200,50);
```

4.2.2 Where and When to Draw

Drawing code should be placed in a **function** so that it can be called in response to an **event**. Then you can, for example, draw in response to a button click or when someone mouses over the canvas.

Often, though, you will want to draw on the canvas as soon as the page loads. You might think the right approach in this case is to not use a function, but simply include the drawing commands inside a script **element**. The problem is that your code might execute before the **DOM** is completely constructed. If that happens, the DOM **node** for the canvas might not be available at the time your code runs.

The solution is to put the drawing code in a function and then call it in response to the `onload` event of the `<body>` element, like this:

```
<body onload='draw()'>
```

The `onload` event is fired once, just after the browser has finished constructing the DOM. You should use this event to execute any kind of initialization code, including canvas drawing.

<exercises section='4.2.2'>

1. Continue your **pong** game setup. Use the rectangle and text drawing commands to draw the player's paddles, a net, and a score at the top of the screen. Put the drawing commands inside a **function** that is called in response to the `onload` **event**.

2. Get the **canvasTestBed.html** file from the **example pack**. Write a function that creates an 8x8 checker board pattern in the **canvas** when the button 1 is clicked.

</exercises>

4.2.3 Polygons

If you want to draw a polygon other than a rectangle, you have to first trace a path on the canvas and then either `fill()` or `stroke()` it.

Drawing a polygon has five steps:

1. Begin the path with `ctx.beginPath()`
2. Move to the starting position with `ctx.moveTo()`
3. Trace out the path with `ctx.lineTo()`
4. Close the path with `ctx.closePath()` (this adds a line that goes back to your starting position)
5. Draw the shape with either `ctx.fill()` or `ctx.stroke()`

Do It Yourself

Here's another **HTML5 canvas**, with an `id` of "testCanvas3".

In the JavaScript console of your browser, try the following commands to draw an orange triangle:

```
var c=document.getElementById("testCanvas3");
var ctx=c.getContext("2d");
ctx.fillStyle="rgb(255,128,0)";
ctx.beginPath();
ctx.moveTo(200,20);
ctx.lineTo(260,80);
ctx.lineTo(140,80);
ctx.closePath();
ctx.fill();
```

The context still remembers the path even after you have used it to draw a shape. So if you want to put an outline around your triangle you can do so without retracing the path:

```
ctx.strokeStyle="rgba(0,128,255,0.5)";
ctx.lineWidth=10;
ctx.stroke();
```

To draw a new shape, just begin a new path:

```
ctx.beginPath();
ctx.moveTo(140,20);
ctx.lineTo(260,20);
ctx.lineTo(200,80);
ctx.closePath();
ctx.fillStyle="black";
ctx.fill();
```

<exercises section='4.2.3'>

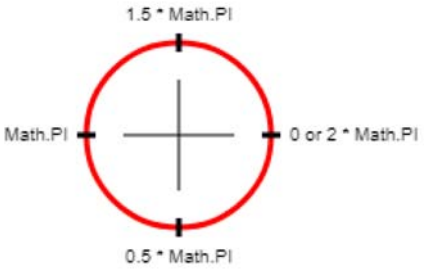
1. Get the **canvasTestBed.html** file from the **example pack**. Write a **function** that draws a simple line drawing of a house on the screen when button 2 is clicked.
2. Extend the code you wrote for question 1 so that each time button 2 is clicked, a new house appears to the right of the last one drawn. (Hint: use a **global variable** to keep track of the starting x-coordinate of the next house.)
3. In the **canvasTestBed.html** file, write a **javascript** function to draw a star in the top left corner of the **canvas**. Add an `onclick` **event** to button 3 so that when it is clicked, a star is drawn.
4. Extend the code you wrote for question 3 so that when button 3 is clicked, a star is drawn at a **random** position in a random color. (Hint: Generate random x and y **values** using `Math.random()` and then add these values to the coordinates you use in the drawing functions.)

</exercises>

4.2.4 Circles and Curves

The `ctx` object also contains methods for drawing circles and arcs, to be used alongside `ctx.beginPath()`, `ctx.fill()`, `ctx.stroke()` and the other path drawing methods.

You can use the `ctx.arc()` method to draw a circle or part of a circle. You specify the center of the circle, its radius and then the starting and ending angle for your curve. These angles are specified in radians, which is an alternative to using degrees (in the same way that inches are an alternative to centimetres). When you work with radians, π radians (that's the Greek letter Pi, where π≅3.14) is equivalent to 180 degrees. So π/2 is 90 degrees, π/4 is 45 degrees, and so on. You can use the built in **JavaScript** constant `Math.PI` for this.



Arc Angle Quick Reference

At right is a quick **reference** for common radian values to use in arc drawing. (The picture itself is actually a `<canvas>` element and the diagram is drawn using JavaScript.)

Do It Yourself

Here's yet another **HTML5 canvas** with an `id` of "testCanvas4".

In the JavaScript console of your browser, try the following commands to draw a pink circle:

```
var c=document.getElementById("testCanvas4");
var ctx=c.getContext("2d");
ctx.fillStyle="pink";
ctx.beginPath();
ctx.arc(200,50,45,0,Math.PI*2);
ctx.closePath();
ctx.fill();
```

If you want half a circle, you should draw from π/2 to 3π/2.

```
ctx.strokeStyle="black";
ctx.beginPath();
ctx.arc(200,50,45,Math.PI/2,3*Math.PI/2);
ctx.closePath();
ctx.stroke();
```

Arcs are always drawn clockwise by default. What would you need to change in the above commands to draw the other half of the circle?

Can you clear the canvas and use the `ctx.arc()` **method** to draw **Pac-Man**?

There are also commands available to draw curves as part of the path (that is, continuing from wherever your path left off). They are `ctx.arcTo()`, `ctx.quadraticCurveTo()` and `ctx.bezierCurveTo()`. There are also other advanced drawing commands are all discussed in full in the **w3Schools** Canvas Reference.

<exercises section='4.2.4'>

1. Continue your **Pong** game setup by adding a ball to the screen.
2. Get the **canvasTestBed.html** file from the **example pack**. Write a function that draws a simple line drawing of a car on the screen when button 4 is clicked. Have the car appear under the house from the previous section's exercises.
3. Extend the answer to question 1 so that each time button 4 is clicked, a new car appears to the right of the last one drawn. (Hint: use a global variable to keep track of the starting x-coordinate of the next car.)
4. (Courtesy of Ann Cadger) Write a function that uses the curve drawing methods to draw a turtle on the screen when the mouse moves over the canvas (**click here for turtle ideas [https://www.google.ca/search?q=turtle+drawing]**).

</exercises>

4.3 Canvas Input Events

If your goal is to create a game or some kind of interactive display, you need to be able to react to user input on the **canvas** from a keyboard or a mouse. (You also need to be able to produce animations, but that will be covered in Chapter 6).

4.3.1 Mouse Input

You were introduced to all the relevant mouse **event attributes** in Chapter 5. As a reminder, they are `onclick`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmouseleave` and `onmousemove`. But to take full advantage of these events for a canvas, it is not enough to know that the canvas was clicked or that the mouse is moving on top of the canvas, we also need to know where the mouse is. For this, you can use the `event` **object**.

Whenever an event happens, an `event` object is created that contains information about the event. That event object is passed to the code you put inside the event attribute (this is done through a wrapper function that is created to contain the code you write there). You can access this event object and pass it on to the function you wrote to respond to the event, like this:

```
<canvas onclick='myFunction(event);' ... ></canvas>
```

Now you can get information out of it. For example, `event.target` will be a **reference** to the **element** that was the target of the event, in this case the canvas.

Finding out where on the canvas the mouse event happened takes a little bit of math:

```
var x = event.pageX - c.offsetLeft
var y = event.pageY - c.offsetTop
```

The `pageX` and `pageY` properties get you the x and y location of the exact pixel on the web page where the click happened, with `x==0` and `y==0` being the very left and top edges of the page. But you want to know where on the canvas the click happened, with `x==0` and `y==0` being the left and top edges of the canvas. To find out, you simply have to subtract the `offsetLeft` and `offsetTop` properties of the canvas (`c`), which tell you how many pixels the canvas is away from the left and top of the screen.

Do It Yourself

Load up **canvasTestBed2.html** from the **example pack** and click on each canvas canvas. Each canvas calls a function when clicked that reports the location of the click using the equation above. The location is reported to the canvas and also in the console using `console.log()`

But the second canvas locations are all off by 30 pixels. This is because this canvas has a 30 pixel border that counts as part of the clickable area of the element. So to get the x and y calculation right for this canvas, the size of the border has to be subtracted.

Load **canvasTestBed2.html** into **netbeans [https://netbeans.org/]** and fix the `whereClicked` function so that it uses this calculation for the second canvas (but not the first):

```
var x = event.pageX - c.offsetLeft - 30;
var y = event.pageX - c.offsetLeft - 30;
```

Now change `onclick` to `onmousemove` for one of the **canvases**. This could be the beginning of a painting app...

There are a few other event **fields** that might also be useful for mouse input: `event.ctrlKey`, `event.altKey` and `event.shiftKey` will each be true if the `ctrl`, `alt` or `shift` keys (respectively) were held down when the mouse event happened. Otherwise they will be false.

<exercises section='4.3.1'>

1. Change **canvasTestBed2.html** so that it contains only one canvas and modify the code so that it draws a circle with a 5-pixel radius when the mouse moves over the canvas.

2. Now add a global variable to indicate whether the paintbrush is a large circle, small circle, large square, or small square. Have the four buttons call functions to switch between brushes. Make sure the square brushes are drawn centered around the x and y coordinates of the mouse.

3. Add an HTML color element to allow the user to choose the drawing color. In the drawing function, set the fillStyle using the value of this element.

4. Now change the app so that drawing only happens if the mouse button is pressed down. This is a little trickier than it sounds to get working exactly right. It will require the use of another global variable and a number of different events.

5. In your Pong game, get a paddle moving under mouse control (fix its x location and make its y location match the y coordinate of the mouse). This is a little different from paint because if you have been following the development of Pong in these exercises, then you already have the canvas redrawing every 16 milliseconds. So you now need a global variable for the y location of the paddle, then in the `onmousemove` event, all you need to do is change this global variable - the drawing will happen in the frame drawing method.

6. In the paint app, change things so that if the user is holding down the `ctrl` key, dragging the mouse will erase a square area (with `clearRect`).

7. In the paint app, change things so that if the user is holding down the `alt` key it doubles the size of the brush they are currently using.

8. **Image Magnifier Project.** Use the mouse events to implement an **image magnifier like this one [http://mark-rolich.github.io/Magnifier.js/]**. An image magnifier uses a large and small version of an image and displays the small version by default. Then when the user mouses over the image, it reveals a previously hidden **element** containing the large version of the image, but showing only the part of it that is centered around the current mouse location. There are a number of ways to accomplish this, but one way is to use a `<div>` element for the magnified image and set its `background-image` CSS **property** to display the large version of the image. Then adjust the `background-position-x` and `background-position-y` properties in response to mouse movements over the target image so that the correct feature is shown.

</exercises>

4.3.2 Keyboard Input

There are two keyboard events that are guaranteed to work on all browsers. The `onkeydown` event will fire when a key is pressed down, and the `onkeyup` event will fire when the key is lifted. In theory these event attributes can be placed on most **HTML elements**, but keyboard events will only ever be fired for elements that have **focus**.

It is possible to give focus to a `<canvas>` element. To do this you need to give the `<canvas>` a `tabindex=1` attribute and then call the **javaScript** `focus()` **function** on the element to give it the focus. But this is a little awkward and if there are other focusable elements on the page, they could steak focus away.

A better solution is often to put the `onkeydown` and `onkeyup` events on the `<body>` element. This is the one element that always has focus.

Just like with mouse events, you should pass the `event` object to the function you are calling:

```
<body onkeydown='foo(event);'>
```

Then you can access the integer code for the key that was pressed as follows:

```
function foo(event) { alert(event.keyCode) }
```

Just like with mouse events you can also use `event.ctrlKey`, `event.altKey` and `event.shiftKey` to find out if any of these keys were held down when the event happened.

Do It Yourself

Load up **canvasTestBed3.html** from the **example pack** and slowly press and release a key to see the keyboard events in action.

You can use this little test app to figure out key codes or you can go to the **Mozilla Developer Network keyCode page [https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent]** for a full list of key codes.

<exercises section='4.3.2'>

1. Write code in **canvasTestBed3.html** or create your own page with a **canvas** on it. Have a square or circle in the middle of the screen and respond to the arrow keys by moving and redrawing the circle 10 pixels up, down, left, or right.

2. Modify your code from the last exercise so that the shape begins to move one pixel at a time when you press down an arrow key, and stops when you lift it up. This requires a very different architecture: treat it like an animation with 60 frames per second. When a key is down, record that fact in a **global variable**. Then in each animation frame, if a key is down, move the shape one pixel in the appropriate direction.

3. Get the second paddle in your **Pong** game moving under keyboard control. This is a little trickier than it sounds but one way to do it is to have a **global variable** for up and another for down. When the user presses down the up button, set the up variable to `true`. When they release it set it to `false`. Do the same for the down button. Then in the frame drawing **function** (the one that is controlling the ball) move the paddle up or down depending on which of these variables are currently `true`.

4. In the paint app, make the `backspace` and `delete` buttons clear the screen. (It might be nice to pop up a confirmation dialog before you do that.)

</exercises>

5. Functions as Values

Up until now, you have been adding **event** handlers to **elements** by placing **JavaScript** code into the corresponding HTML **attributes**, like this:

```
<input type="button" onclick="myFunction()">
```

This is not considered best practice by most JavaScript programmers for at least three reasons.

1. It spreads your JavaScript code throughout the HTML file, making the code harder to maintain. (i.e. It would be much better if you could add all your event handler code in a single place.)
2. It violates the clean separation between the structure of the interface (the job of HTML), the style of the interface (the job of CSS) and the functionality of your app (the job of JavaScript).
3. It causes the browser to create a new **anonymous function** with the code you put in the attribute as its contents. This is unnecessary and wasteful. (the term "anonymous function" will be explained in a moment.)

It is widely considered best practice to add event handlers using JavaScript **statements** after the document has loaded. But to be able to do that, you need to understand how JavaScript **functions** work in a little more depth.

5.1 The Truth About Functions

The truth about **JavaScript functions** is that they are actually **values**, just like **Strings**, Numbers, **Booleans**, and **Objects**. Like any other value, functions can be stored in **variables**, passed as parameters to other functions, and returned from function **calls**.

The technical term for this is that JavaScript functions are **first-class functions**.

When you use a function declaration like this:

```
function foo (a, b, c) {
    alert(a+b+c);
}
```

What you are really doing is creating a variable named foo, and assigning a function value to it.

The following shows the assignment of a function literal to a variable. It is legal JavaScript code, and is equivalent to the function declaration above:

```
var foo = function(a, b, c) {
    alert(a+b+c);
};
```

> ***Java Connection***
>
> **Methods vs Functions.** In **Java**, **methods** and **variables** are completely different entities. (Question: how would you describe this difference?) In **JavaScript**, **function** names are actually variable names, and functions are data.

The above **statement** creates a variable named `foo`, and assigns to it an **anonymous function** with 3 parameters. (Note that since the statement above is an assignment statement, we put a semicolon at the end of it.) The function being assigned is called an anonymous function

because it is created without being given a name. The "name" used to call it comes from the assignment of the anonymous function to a variable.

Anonymous functions are also referred to as "**function literals**" because they are like Number, String or Array **literals** - they are values that are hard-coded into the program.

The main difference between the two notations is that you can only use function **declaration** in certain contexts, but you can use variable declaration and function literals in any context. In fact, the function declaration notation is often avoided by JavaScript programmers on the grounds that it is limited in use and adds nothing to the language. These programmers always use the variable declaration / function literal notation instead.

Do It Yourself

No matter whether you use function declaration or variable declaration with function literals, if you refer to foo in your program, you are referring to a variable. To see this, try the following in the JavaScript console of your browser:

```
function foo () {alert("hi");} ← foo is a function
foo;
foo();
```

Question: What is the difference between the last two lines above?

```
alert(foo);
alert(foo());
```

Question: What is the difference between the above two lines?

Try the lines below to see that you can switch `foo` back and forth between a function and other data types.

```
foo = 5; ← foo is now a number
alert(foo);

foo = function() {alert("hi");}; ← foo is a function again
alert(foo);

foo = "hi"; ← foo is now a string
alert (foo);
```

Answers: In the console window, `foo` returns the value of the variable. It's a function. But `foo()` calls the function so you will see the popup. On the other hand, `alert(foo)` will display a popup whose content is the value of the variable `foo` (if it's a function, it will show you something to that effect). Finally, `alert(foo())` will first call the `foo` function, displaying a popup. Then it will create another popup to display the **return value** of `foo`. Since there is nothing explicitly returned, it will return the special value `undefined`.

5.2 Better Ways to Add Event Handlers

If you want to add **event** handlers to an **element** after it is created, you can do it by retrieving the element from the **DOM** and assigning a **function literal** to the **attribute** for the event handler.

Do It Yourself

Try the following in the javaScript console for this textbook.

```
document.getElementById("test1").onclick = function(event) {
    alert("hello!");
};
```

Since this DIY box has an id of `test1`, you should now be able to click it and see the result.

P.S. Don't worry about the `event` parameter above. It will be explained later.

Or you can create the **function** (either through function **declaration** or using a function **literal**) first, and then assign it to the event handler.

Do It Yourself

Try the following in the JavaScript console for this textbook.

```
var clickHandler = function clickHandler (event) {
    alert("goodbye! ");
}
document.getElementById("test2").onclick = clickHandler;
```

Now you should be able to click this box for a message.

Another option that some programmers prefer is to use the `addEventListener` **method** which takes two parameters. The first is the event name (without the "on" -- so use "click" instead of "onclick") and the second is the handler function.

Do It Yourself

Try this in the JavaScript console for this textbook. Note that you will have to define clickhandler first. You can get it from the last DIY box.

```
function clickHandler (event) {
    alert("goodbye! ");
}
document.getElementById("test3")
    .addEventListener("click",clickHandler);
```

Of course you can also add a handler using a function literal. Try this:

```
document.getElementById("test3")
    .addEventListener("click", function(event) {
        alert("hello!");
    });
```

If you execute both code snippets above, then when you click this box you should get two popups, not one. This illustrates one crucial difference between assigning to `onclick` and using `addEventListener`. The latter allows you to add many **event** handlers of the same type to the same **object**.

Java Connection

Event Listeners. Java and **JavaScript** both use `addListener` **methods** to register **event** handlers. The difference is that in Java an **event listener** is an **object**, while in JavaScript it's a **function**.

5.2.1 The `event` Object and the `this` Keyword

Note that the functions above all contain an `event` parameter. Any event handler is always passed a special event **object** that represents the event that caused this function **call**. You can use it to access information about the event, such as which element triggered the event (`event.target`), which mouse button was pressed (`event.button`), the location of the mouse on the page (`event.pageX` and `event.pageY`), etc.

Do It Yourself

Here are two last things to try in the JavaScript console for this textbook.

```
document.getElementById("test4").onclick = function(event) {
    alert(event.pageX+","+event.pageY);
};
```

Now when you click on this box, you will get to see the exact X and Y location of your click, in pixels relative to top left corner of the page. If you change `pageX` and `pageY` to `clientX` and `clientY` you get the location relative to the top left corner of the browser window.

The **event object** contains a number of useful **fields**, but probably the most useful is the `target` field which returns the DOM node that was the subject of the event.

Try this:

```
document.getElementById("test4").onclick = function(event) {
    alert(event.target.innerHTML);
};
```

Now clicking this box should get you the `innerHTML` of the **element** you clicked on.

But sometimes `event.target` won't get you exactly the element you wanted. in this case, you should use the `this` keyword.

Do It Yourself

In the last example the contents of the alert will vary depending on where you click inside the DIY box. This is because the `target` is not necessarily the same as the **element** you added the listener to. Can you figure out by inspecting the DIY box what the `target` actually refers to?

If you want to make sure you access the element that the listener was added to, use the keyword `this` instead of `event.target`.

Try this:

```
document.getElementById("test4").onclick = function(event) {
    alert(this.innerHTML);
};
```

Now clicking the DIY box above this one should get you the `innerHTML` of the entire box, no matter where you click on it.

For more information on the Event object, see the w3schools' **Dom Events** page.

5.2.2 The `window.onload` Event

It is also considered best practice in **JavaScript**, to write all of your code inside a `window` load event handler, like this:

```
window.addEventListener("load", function() {
    // ALL YOUR CODE GOES HERE
});
```

When you put all your code inside the load event, the browser will wait until the page is ready before executing it. So you can put your code into the `<head>` of your document without having to worry about whether the DOM is ready at that point. This is a big advantage.

Another advantage is that now all **variables** you use (including function names) become **local** to the **anonymous function** you assign to the `load` event. This is good too because it minimizes potential name conflicts with other JavaScript code or libraries you might be using.

> ***Java Connection***
>
> **Functions within functions.** One consequence of the fact that **functions** in **JavaScript** are **values** stored in **variables** is that you can **declare** functions within functions. There is no way to do anything like this in **Java**.

Note that because function declarations are just variable declarations in JavaScript, even other helper functions can be defined inside the `window` load event.

<exercises section='5.2.2'>

1. Load **eventHandlerTest.html** from the **example pack**. Compare how the handlers have been added to the **elements** with `id` `test1` and `test2`.
    a. Consider the page as it is loading into the browser. At what point in time is each handler added?
    b. Try commenting out the `window.onload` line and the `};` line that goes with it and reload the page. Describe how the page works (or not) now and explain why.
2. Add two handler **functions** to the element with `id="message"` using **JavaScript** (not HTML **attributes**). These functions should respond to the `onmouseover` and `onmouseout` **events**. When the mouse is inside the message box, one of the other boxes should be pink and the other should be white. When the mouse is outside the message box, the one that was white should become light blue and the one that was pink should become white. All your code should be inside a window load event handler.
3. Go back to the other exercises from this chapter (messages, catch the rabbit, etc.) and refactor them so that all of your code (including code in `onclick` or other HTML attributes) is tucked away inside a `window` load event handler.

</exercises>

5.3 Functions, Timers and Animation

Now that you know the truth about **functions**, we can also talk about how to use timers to create animation effects on a canvas

Animation refers to any picture in which **elements** of the picture seem to move or change over time. The basic technique in any form of animation is to present a series of animation frames (i.e. pictures) one after another very quickly, with each frame slightly different from the last. For example, an **object** might move very slightly to the right or grow slightly from frame to frame while the rest of the picture remains unchanged. If this is done quickly enough, the eye will be fooled into seeing smooth motion.

To animate an element using **JavaScript**, we need three things:

1. A set of **variables** that specify important details about the object being animated (e.g. x and y position, color, size, etc.)
2. A function that updates these variables and then makes changes to the CSS for that object using the new **values**.
3. A way to trigger the update function repeatedly

(If you want to animate drawings on a **canvas**, use a very similar **method**, but the update function will need to clear and redraw the entire contents of the canvas each time it is called rather than making changes to the CSS.)

5.3.1 JavaScript Timers

We'll start with item 3 from the list above: We need a way to trigger a function repeatedly. JavaScript timers are the answer to that problem.

JavaScript functions are called in response to **events** (e.g. the user presses the mouse button or the page loads). If you want a function to be called repeatedly, or just once but at some time in the future, you have to schedule a timer event. The `window` object in JavaScript contains a `setTimeout` function to trigger an event some time in the future, and a `setInterval` function to trigger an event repeatedly at set intervals.

To trigger an event repeatedly, you can use the following command:

```
setInterval( functionName ,  interval );
```

In the above, `functionName` is the name of the function you want to **call** when the timer events happen, and `interval` is the number of milliseconds between each event.

For example, if you want to call a function named `tick` every half a second (500 milliseconds), the command would be:

```
setInterval(tick, 500);
```

Note that you must not use `tick()` with the brackets. This would call the `tick` function immediately. But we don't want to call it now, we want to pass it like a parameter to the `setInterval` function so that it can be called later, when the timer event happens.

To call a function just once, but at some time in the future, use the following command to schedule a single timer event:

```
setTimeout( functionName ,  interval );
```

The parameters are the same as `setInterval`, but the function you specify will only be called once, after `interval` milliseconds.

> ***Do It Yourself***
>
> Try the following in the **JavaScript** console of Chrome:
>
> ```
> function hi() { alert("Hello"); }
> setTimeout(hi,5000);
> ```
>
> Now wait five seconds for your message to appear.
>
> Try the same thing on a repeating timer:
>
> ```
> setInterval(hi,5000);
> ```
>
> Now you will get a popup every 5 seconds. You can reload the page to make it stop.

Both `setTimeout` and `setInterval` return an integer id for the timer you have created. If you want to stop a timer, you can do so with the `clearTimeout` function. But you have to have saved the id of the timer in a variable:

```
var timerId = setInterval(myFunction, 1000);
```

With this variable you can stop the timer at any time:

```
clearTimeout(timerId);
```

This will also work for `setTimeout`.

Do It Yourself

Go back to the previous DIY box and execute the repeating popup command again, but this time store its return value in a variable:

```
var timerId = setInterval(hi,5000);
```

Now you can stop the popups by clearing the timer:

```
clearTimeout(timerId);
```

Do It Yourself

Load the **timerExample.html** file from the **example pack** and run it. You should see the word "tick" appear every 0.5 seconds, and after just over 5 seconds you should see the word "*** ALARM ***" appear. You can stop the ticking by pressing the "stop" button.

Examine the code for this file make sure you can answer these questions:

1. Which functions stop and start the timers?
2. When does each of these functions get called?
3. What is the purpose of the `alarm` and `tick` functions?
4. How does the stop function know which timer to stop?

Now add code to the file so that there is a "stop alarm" button that, when pressed, will prevent the alarm message from being displayed (if it hasn't already been displayed).

5.3.2 Getting Things Moving

Recall that we needed three things to get an animation going:

1. **Global** animation variables,
2. A function to update those variables and change CSS properties (or to redraw the canvas if you are using one), and
3. A way to trigger that function repeatedly.

We now have the last part: `setInterval` can be used to call a function every few milliseconds. All we need now is some **global variables** and an `updateAnimation` function.

The global variables should keep track of position and other information for the objects that are moving.

The `updateAnimation` function should take the following actions:

1. Update the animation variables (e.g. `x = x + 1`)

2. Make the required changeSize
    a. If you are animating **HTML elements**, set the relevant CSS properties (e.g. `e.style.left = x + "px"`). Note that the CSS **property** `position` must be set to `absolute` or `fixed` for this to work.
    b. If you are animating a picture on a canvas, clear the canvas (e.g. `ctx.clearRect(0, 0, canvas.width, canvas.height)`) and then draw the new frame, using the animation variables (e.g. `ctx.fillRect(x, 100, 50, 50);`).

For the appearnace of smooth motion, the `setInterval` function should be used to trigger a call to `drawFrame` about 60 times per second. That works out to an interval of 16 milliseconds.

Do It Yourself

HTML Animation: Load the **sampleAnimation.html** file from the **example pack** and watch it run.

Canvas Animation: Load the **sampleCanvasAnimation.html** file from the example pack and watch it run.

Whichever option you chose above, make sure you can answer the following questions:

1. When and how does the animation start?
2. When and how does the animation stop?
3. For the canvas animation, what would happen if you skipped the clear canvas step?

<exercises section='5.3.2'>

1. Fetch either **sampleAnimation.html** or **sampleCanvasAnimation.html** from the **example pack**.
    a. Modify it so that the ball moves down and to the right.
    b. Now modify it so that the ball shrinks as it moves.
    c. Finally, modify it so that the ball starts off as black and ends as red (Hint: define a red variable to start at 0 and grow to 255. Then set the color using `rgb(___,0,0)` with the value of your variable in the blank.
2. Modify **sampleAnimation.html** or **sampleCanvasAnimation.html** from the example pack so that the ball bounces back when it hits the edge of the screen. (Hint: instead of `x=x+1` use `x=x+xSpeed` where `xSpeed` is initially set to a positive number. Then when the ball reaches the edge of the screen, multiply `xSpeed` by `-1` to start moving the ball in the opposite direction.
3. Modify **sampleAnimation.html** or **sampleCanvasAnimation.html** or update **moveme.html** from the example pack so that the ball moves diagonally and bounces off all four walls. (Hint: define an xSpeed and a ySpeed.)
4. Get **flyin.html** from the example pack and add code to it to make the `<h1>` elements fly in from the side of the screen. Start with them all moving together, then see if you can extend it to make them fly in at different times.
5. Create your own animation from scratch with at least two elements moving or changing at once.
6. Go back to **catchTheRabbit.html** from the example pack. Delete 3 of the rabbits and create a new version of the game where the rabbit runs to a different (randomly selected) location on the screen when you mouse over it. (Hints: Use `window.innerWidth` and `window.innerHeight` to get the height and width of the screen. For global variables, use `x`, `y` to represent the rabbit's position and `targetX`, `targetY` to represent where the rabbit is going. Each call to updateAnimation should just move `x` and `y` closer to the targets. When the user mouses over the rabbit, set a new target.)
7. More Canvas Exercises
    a. Now get the ball moving in the Pong game you are developing.
    b. Get the **canvasTestBed.html** file from the example pack. Write a function that creates an 8x8 checker board pattern in the canvas when the button 1 is clicked (if you haven't already done that). Now make the modifications necessary so that after button 1 is clicked the checkerboard reverses every 250 milliseconds.
    c. Extend your solution to the previous exercise so that when you press button 1 again, it stops the animation.
    d. **The Greeting Card Project:**

    Create a web app to display custom greeting cards for a mobile phone user (with a screen size of 320px by 480px).

    When the user loads the page, they should see a heading and a form that lets them customize their card. The form should let them enter a date, the name of the event, the name of the recipient and other information as you see fit. There should be a "make card" button that, when they hit it, hides the form and reveals a canvas showing the "front" of the card, with a simple but elegant design that incorporates the text they entered. The card design should include some type of animated element (something moving, changing color, growing or shrinking, etc).

    BONUS: When the user clicks the card, make it "open" to show them a different design on the inside.

```
</exercises>
```

5.4 Functions, Images and Sound

To make a nicer **canvas** game, you might want to load images from a file and then draw them on the canvas. To make a better overall experience for the user, you also might want to play sounds in response to certain **events**.

5.4.1 The Basics

To draw an image, you have to create an `Image` object, then load an image file into it and then draw it. Creating the image object and loading the image should be done once and stored in a global variable.

```
var myImage = new Image();
myImage.src = "images/myImageFile.jpg";
```

Once the image is loaded, you can draw it on the canvas in response to an event.

```
ctx.drawImage(myImage, x, y);
```

To play a sound, you have to create an `Audio` object, then load an audio file into and then play it. Creating the Audio object and loading the audio file should be done once and stored in a global variable.

```
var myAudio = new Audio("sounds/myAudioFile.mp3");
```

Once the audio is loaded, you can play it any time. This is done in two steps.

```
myAudio.load(); // resets the sound so it can be played repeatedly
myAudio.play(); // plays the sound
```

5.4.2 Dealing with Lag

There is one problem lurking in the above instructions. When your app is on line, the HTML page will load first and the sounds and images will only begin to be downloaded from your site when you create the `Audio` and `Image` **objects**. What if you try to draw your images or play your sounds before they have loaded?

To deal with this problem for images, you can set up an `onload` event like this:

```
myImage.onload = myImageReady;
```

Where `myImageReady` is the name of a **function** that will be called when the image is loaded. You can use it to set a **global** flag to let the rest of the app know the image is ready

The solution for audio is similar but uses a different event:

```
myAudio.oncanplaythrough = myAudioReady;
```

In this case, `myAudioReady` will be called when the sound can be played through from beginning to end.

> ***Do It Yourself***
>
> Try out **soundAndVision.html** from the **example pack** to see images and audio in action.

<exercises section='5.4.2'>

1. Follow the instructions above to put sounds into your **Pong** game and use images for the paddles and ball. For images you can legally use, search **creative commons [http://search.creativecommons.org/]**. For sound effects, try **freesound.org [http://freesound.org]** or **soundjay.com [http://soundjay.com]**.

</exercises>

5.5 Creating Your First Canvas Game

Now you have everything you need to make a simple game in the **HTML5 canvas**. Here's a list of basic components you can adapt to suit your needs:

1. A set of **global variables** to control position, speed, direction and other **attributes** of the movable **objects**. You may also need **variables** to control the state of the game (game on, paused, game over, etc.) and possibly the score.
2. A set of **functions** that are called in response to keyboard and mouse **events**. These function should make changes to the **global** variables that control the player objects in the game.
3. A `computeFrame` function that is called once very 16 milliseconds. This function contains the main code for the game.

The primary job of `computeFrame` is to draw the current animation frame. But before it does that, it should update the positions of all the objects based on the relevant variables. It should also check for collisions between objects or between objects and the edges of the screen and make changes as a result (reverse speeds to simulate a bounce, increase the score, mark an alien as "dead", end the game, etc.)

Here are a few extra tips to help you out:

1. An object's direction and speed can be represented using two variables for xSpeed and ySpeed. In `computeFrame`, use `x += xSpeed` and `y += ySpeed` to move it one step.
2. You don't have to use integers. If you set the x and y speeds to -0.5 and 0.25 you will get an object that moves slowly to the left and slightly down.
3. To make an object bounce, you can reverse its speed with `xSpeed = xSpeed * -1` and `ySpeed = ySpeed * -1`. If you reverse both, it will bounce back the way it came. If you reverse only the x speed it will seem to bounce off a vertical surface. If you reverse the y speed it will seem to bounce off a horizontal surface.
4. Gravity can be simulated by adding a small amount (e.g. 0.01) to the ySpeed of an object at every step.
5. Collisions can be detected by measuring how close two objects are to one another. Use their x and y locations to compute the distance between them using the **length of a line segment formula [http://www.freemathhelp.com/length-line-segment.html]**. If they are too close, it's a collision.

That's it. The rest you will have to work out for yourself. Happy gaming!

<exercises section='5.5'>

1. Finish that **pong** game!
2. For an extra challenge, try a brick-breaker game like breakout or arkanoid as well.

</exercises>

6. Custom Objects

Throughout this course, you have been using **objects** in your programs (e.g. the `document` object) but you have not been programming in an **object-oriented** manner.

Javascript supports many **programming paradigms**. It supports **imperative programming** by letting you place JavaScript **statements** anywhere you want on a page. These statements are executed in the order they are encountered (from top to bottom) when the page is first loaded. It also supports **procedural programming** by letting you **declare functions** that can be executed over and over again as necessary. And it supports functional programming through its support for **first-class functions**.

In this chapter, you will learn some of the ways in which JavaScript also supports object-oriented programming, allowing you to create your own objects with whatever instance **variables** (**fields**) or **methods** you wish.

One caveat before we begin: The JavaScript approach to object-oriented programming is quite different from Java and almost every other language. A full discussion is beyond the scope of this course, but you can read more in **Introduction to Object-Oriented JavaScript [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript]** on the Mozilla Developer Network.

6.1 Some Important Facts

Java Connection

Classes and Objects. JavaScript lets you create **objects** but does not let you define any **classes**.

In **Java**, if you want to create an **object**, you define a **class** to use as a blueprint and then create objects using it. In **JavaScript** there is no such thing as a class, and there are many different ways to create a new object. We'll look at two simple ways in the next section, but first here are some important reminders about how JavaScript objects work.

6.1.1 Object Notation

As you have already seen, in JavaScript, you can access an object using the dot operator or by using **associative array** brackets.

For example: `document.forms.["myForm"]` is the same as `document.forms.myForm`, and `document["getElementById"]("test1")` is the same as `document.getElementById("test1")`

This will be true for the objects you create as well.

6.1.2 Flexible Objects

In Java, the class creates the blueprint for an object and it cannot be modified at run time. You cannot add or remove instance **variables** or **methods** after creating an object.

JavaScript objects are more flexible. You can add a **field** to an existing object simply by assigning something to a new field name. You can also add a new method simply by assigning a **function** to a new field name.

Terminology Note: A JavaScript object is an associative **array** linking key names to **values**. Each key name is an instance variable of that object. As you learned in the previous chapter, functions can be used as values as well. When an Object's instance variable contains a function, we call it a method. Otherwise, we call it a field.

Java Connection

Flexible Objects. In **JavaScript**, the structure of an **object** can be changed after it is created. In **Java**, you can't do that.

Do It Yourself

In a JavaScript console, try the following:

```
document.myVariable = 45;
document.myMethod = function() {alert("hi");}
```

You have just added a new **field** and a new **method** to the `document` object. You can access them like this:

```
alert(document.myVariable); // access the new field
document.myMethod();        // call the new method
```

You can also delete a field from an object using the `delete` operator. This operator can be used to remove any variable from the current context.

Do It Yourself

In the same JavaScript console from the last Do it Yourself box, remove the new method and field like this:

```
delete(document.myVariable);
delete(document.myMethod);
```

Now you should not be able to access this **field** and **method** any more.

6.1.3 Silent Errors

Now you are in a position to understand some of JavaScript's notorious silent errors.

Do It Yourself

In a JavaScript console for this textbook, try the following:

```
document.getElementById("test10").style.color="blue";
document.getElementById("test10").innerHtml = "hi!";
```

Note that the first line worked but the second didn't (the text is blue now, but the contents have not changed).

What went wrong? If you're not sure, try the following in the JavaScript console:

```
document.getElementById("test10").innerHtml
```

This is an example of a very common error in JavaScript: The programmer wants to put a message into a **DOM Node** and she retrieved the **Element object** correctly using `getElementById` but she got the case wrong on the field name. Instead of `innerHTML` she typed `innerHtml`. But rather than throw an error, JavaScript just added a new field to the **Element** object (called `innerHtml`) and assigned the **string** value `"hi"` to it.

This is a prime example of how JavaScript makes a tradeoff between flexibility and ease of debugging.

6.2 Creating Objects from Scratch

With the preliminaries out of the way, we can talk about **object** creation. The first way to create an object is to simply create an empty Object and then add **fields** and **methods** to it one by one. **JavaScript** contains a built in constructor **function** called `Object` that lets you do just that.

Do It Yourself

In a JavaScript console, create a new object and assign it to a variable like this:

```
var person = new Object();
```

If you examine person (type "person" in the console and open the Object that is returned) you will see that it's an object with no contents. To add fields, you can do the following:

```
person.firstName = "Joe";
person.lastName = "Smith";
person.fullName = function() {
    return this.firstName+" "+this.lastName;
}
```

<exercises section='6.2'>

1. TODO - exercises go here!

</exercises>

6.3 Creating Objects With Literals

A **literal** is a **value** that is hard-coded into a program. Whenever you use a number like `-4` or `5.43`, a quoted sring like `"hello"`, a **Boolean** value like `true`, or a bracketed **array** like `[23,52,-3]` in a program, you are using a literal. Like most languages, **JavaScript** supports Number, **String**, Boolean, and Array literals.

You already know that unlike many languages, JavaScript supports **function literals** so it should come as no great shock that, also unlike many languages, JavaScript supports **Object literals** too.

An **Object** literal must be enclosed in curly brackets. Inside the curly brackets, you put a comma separated list of keys and values (remember, in JavaScript an Object is just an **associative array**). The **field** names must be unquoted and the values can be properly formatted literals (including **function** and object literals) or they can be previously defined **variables**.

So this is another way of creating an object in JavaScript.

Do It Yourself

In a JavaScript console, try the following:

```
var x = {name: "Joe", age: 40, male: true};
var y = {person: x, numPeople:1};
var z = {person1: {name: "Jane", age:31}, person2: {name:"Ruth", age:25}};
```

Now type `x`, `y`, and `z` and take a look at the contents of each **object**. Is it what you expected?

Try typing `x.age`, `y.person` and `z.person1`. Is the **return value** what you expected?

Do It Yourself

Here is another example, this time including a method and with some nicer indenting. Paste it into a JavaScript console:

```
var circle = {
    radius: 10,
    getArea: function() {
        return Math.PI*this.radius*this.radius;
    }
};
```

Now call the `circle` object's `getArea` **method**.

<exercises section='6.3'>

1. TODO - exercises go here!

</exercises>

6.4 Creating Objects With Constructors

Here is the code for a Circle class in Java (not JavaScript!) along with a main method that creates two circles and prints their information to the screen. Take a moment to review this code and remind yourself what each line is for. The numbers are not part of the code, of course, they are just there so we can talk about the differences between JavaScript and Java.

```
public class Circle { 1
    public double radius; 3

    public Circle(int radius) { 2
        this.radius = radius;
    }

    public double area() { 4
        return Math.PI * radius * radius;
    }

    public static void main(String[] args) { 5
        Circle c = new Circle(10);
        Circle c2 = new Circle(5);
        System.out.println("Circle 1 Radius: " + c.radius);
        System.out.println("Circle 2 Area: " + c2.area());
    }
}
```

Here is some equivalent code in JavaScript, again with numbers to help the comparison to Java.

```
function Circle(r) { 2
    this.radius = r; 3

    this.area = function() { 4
        return Math.PI*this.radius*this.radius;
    }
}

c = new Circle(10); 5
```

```
c2 = new Circle(5);
alert("Circle 1 Radius: "+c.radius);
alert("Circle 2 Area: "+c2.area());
```

> ***Do It Yourself***
>
> You will find the above code in the **example pack** in the files **Circle.java** and **Circle.html**. Run both programs to verify that they do more or less the same thing. (Note that **Circle.html** uses **jQuery** to encapsulate everything properly and outputs to a <p> **element** rather than to an alert box, but other than that it is the same as above.)

6.4.1 Classes in JavaScript

> ***Java Connection***
>
> **Classes. Java** uses **classes** to create **objects**. **JavaScript** does not.

In **Java**, **object** creation and inheritance are based on **classes** (labeled "1" in the examples above). You define a class as a blueprint for an object and then you can create multiple objects of that type.

JavaScript does not use classes to define objects. That is why the JavaScript example above does not have a "1" label on it anywhere.

6.4.2 Constructors in JavaScript

In Java, constructors (labeled "2" in the examples above) are special **methods** defined inside a class. Their job is to create an object, execute some initialization code, and then return the object created.

In JavaScript, any **function** can be used as a constructor. In the above example, the function labeled "2" is being used as a constructor. Notice that the function name is capitalized – this is to indicate its intended use as a constructor, in much the same way that Java programmers capitalize the names of classes.

> ***Java Connection***
>
> **Constructors. Java** constructors are special **methods** inside a **class** definition. Any **JavaScript function** can be used as a constructor.

6.4.3 Instance Variables (Fields)

In Java, instance **variables** (labeled "3" in the examples above) are declared in the body of the class, outside of any method.

> ***Java Connection***
>
> **Instance Variables. Java** instance **variables** have to be declared. **JavaScript** instance variables are not explicitly declared.

In JavaScript, variables do not have to be declared. If you want to create an instance variable in a constructor function, you use the this keyword to indicate that the variable is a **field** of the object created when the constructor function is called.

6.4.4 Instance Methods

In Java, instance methods (labeled "4" in the examples above) are also declared in the body of the class. These methods can access the instance variables of the object.

In JavaScript, instance methods are actually instance variables that contain a function. (Recall that in JavaScript, functions are **values** that can be assigned to variables.) Any function declared using the this keyword, like the one labeled "4", can access the other instance variables of the object using the this keyword.

> **Java Connection**
>
> **Instance Methods. Java** makes a distinction between **methods** and **variables**. **JavaScript** supports **first-class functions** so it does not need to make that distinction.

6.4.5 Creating and Using Objects

In Java, you create objects by calling their constructors with the new keyword, and you access their public fields and methods with the dot operator (labeled "5" in the examples above). The variables that "store" objects actually store **references** to those objects rather than the objects themselves. The **statements** to create and access objects must be contained within a method of some kind. In the Java example above, I have defined a main method so that the objects will be created at run time.

JavaScript works just like Java for creating and accessing objects. The only difference is that since JavaScript allows **imperative programming**, you don't have to put the statements that create and access the objects inside a method, though it is often good practice to do so.

> **Do It Yourself**
>
> Load **Circle.html** from the **example pack** into a browser, open the **JavaScript** console, create a third circle of radius 52.3 and display its area.

6.4.6 Information Hiding

In Java, you can control access to your variables and methods using keywords like private and protected. This allows you to hide some of the details of the internal workings of your class.

> **Java Connection**
>
> **Information Hiding.** In **Java** you can mark **variables** with the `private` keyword to limit outside access. In **JavaScript** any variable you **declare** inside a **function** (without the `this` keyword) is private.

In JavaScript, you can also have private variables in an object. You can do this by declaring variables or methods in a constructor function without using the this keyword. These variables and methods are private and can be accessed only within the object.

Here's an example, from **Circle2.html** in the **example pack** (see **Circle2.java** for the Java equivalent).

```
function Circle(r) {
    var radius = r; // PRIVATE VARIABLE. ONE PER OBJECT CREATED.

    this.getRadius = function() { // GETTER METHOD FOR PRIVATE VARIABLE.
        return radius;
    };

    this.area = function() {
        return Math.PI*radius*radius;
    };
}
```

JavaScript has an equivalent to Java's private and public, but what about the other two **access levels**, protected and package? These levels concern access by subclasses and classes in the same package. Since JavaScript doesn't have classes or packages, these two access levels are not necessary.

6.4.7 Inheritance

JavaScript does not define objects using classes, which means that inheritance has to work in a very different way.

In Java (and most other **object-oriented** languages in wide use) you define a class which can inherit fields and methods from its superclasses. JavaScript is much more flexible. You can create objects using a constructor function, and then tweak them to add or remove fields as you wish.

In JavaScript, an object is also associated with a prototype that can be used to create similar objects. Prototypes can be used to implement an inheritance mechanism and the correct use of prototypes when creating objects can lead to a more efficient implementation of your code.

Java Connection

Inheritance. Java uses class-based inheritance. **JavaScript** uses prototype **chaining** to implement inheritance.

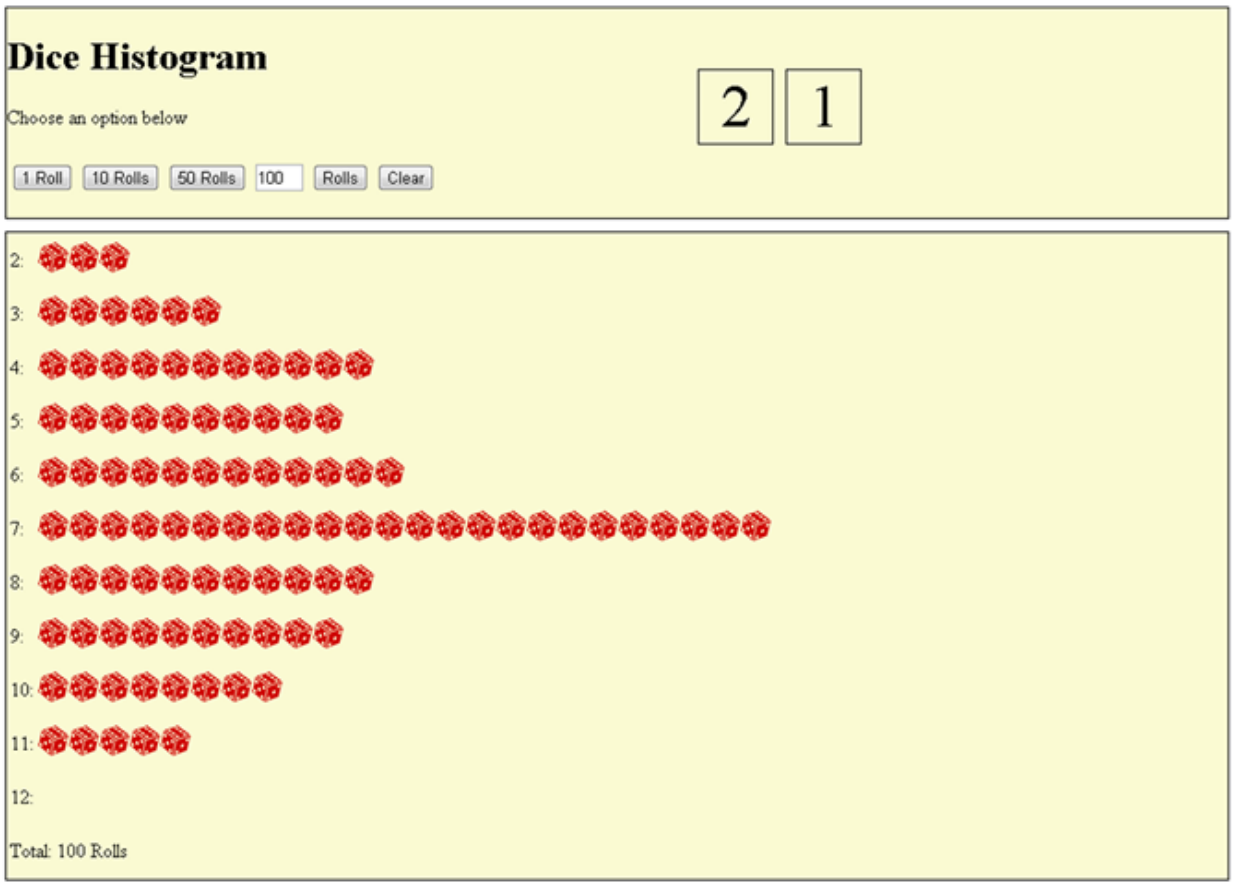
Prototypes are well beyond the scope of this textbook, but it is helpful to know for future reference that you can implement a form of inheritance in JavaScript and that you can make object-oriented implementations more efficient using prototypes. (In Chrome's JavaScript console, when you examine an object, you will notice its prototype information stored in the `__proto__` field.)

7. Programming Challenges

At this point, you have all the tools you need to create a fully functioning, **client-side web app**. Here are some ideas for mini-projects that will give you a lot of great practice. In each case, the instructions are displayed in step-by-step stages so you can do all of the project, or just part of it.

7.1 Dice Histogram

This **web app** could be used by elementary or secondary school students to explore the probabilities involved in rolling a pair of dice.



1. Create a web app with one button. When the user presses the button, it should "roll" two dice (i.e. generate **random numbers** from 1 to 6) and display the result of each die the page. (Hint: Use `Math.floor(Math.random()*6+1)` to simulate each die.)

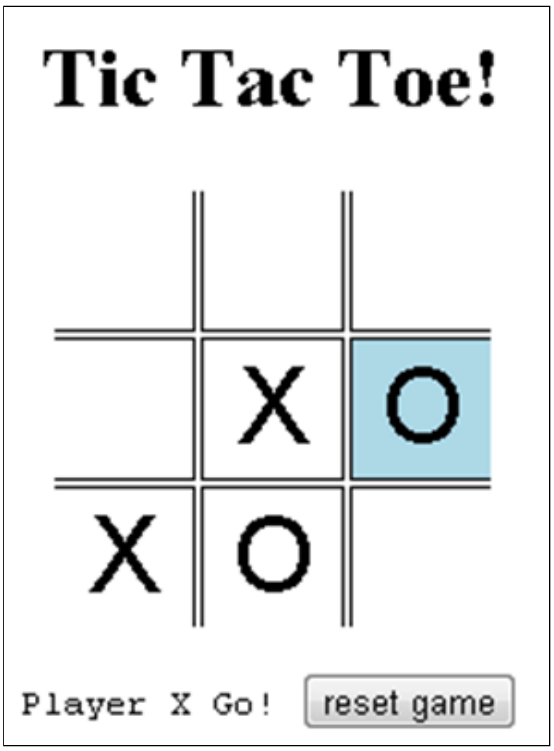
2. Now modify your web app so that it also displays a histogram for the user like the one in the example above, showing the number of times each total from 2 to 12 has been rolled. You can use the file **images/six_sided_die.png** from in the **example pack** to show the number of rolls graphically. You should also show how many rolls there have been in total. (Hint: Use a **table**, alter the `innerHTML` of the appropriate `<td>` **element** for each roll).

3. Add a "clear" button to clear the histogram contents.

4. Add "roll 10 times" and "roll 50 times" buttons.
5. Add a number input element with another button so that they can choose how many times to roll.

7.2 Tic Tac Toe

Who doesn't love Tic Tac Toe? In this project you will develop a **web app** that could run on a desktop or a mobile phone.



1. Create a web app that shows a 3 x 3 grid. When the user clicks on a grid square, it should place an X in that square. Make the total grid width less than 320 pixels so it looks good on a mobile phone (also, see the tip below).
2. Alternate turns. The first click should show an X, then next should show an O, the next an X, and so on. Display whose turn it is in a message under the grid.
3. Add code to make sure the grid square is empty before you allow someone to place an X or O. Pop up an error message if they try to place a piece in an occupied spot.
4. Add a reset button that clears the grid.
5. Add code to detect when the board is full and notify the players of a tie game.
6. Add code to detect a win and display a win message. (You might want to use an **array** for this, though you don't need one. If you want to use an array, check Chapter 7 first.)

Tip: Mobile phone browsers often default to a really wide screen and zoom out. If they do that, it will make your Tic Tac Toe app hard to use, even though it is sized for a phone. To stop mobile browsers from zooming out, you will need to include the following tag in the `<head>` section of your HTML:

```
<meta name="viewport" content="width=device-width">
```

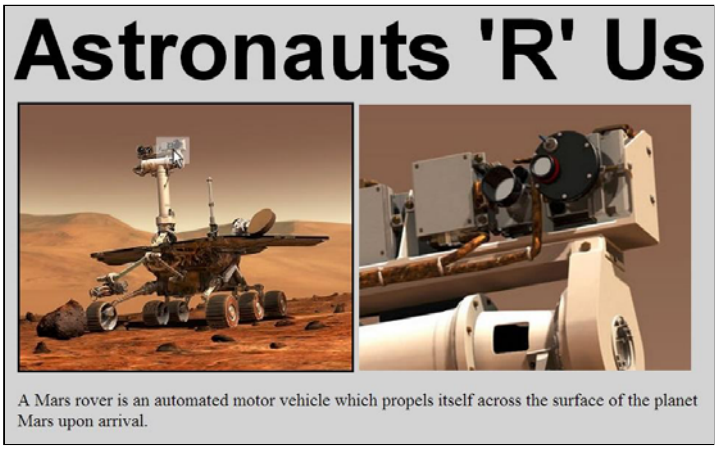
7.3 Image Magnifier

Many shopping sites let you roll over an image to see a magnified version of that image appear beside it. In the web site pictured here, as the user mouses over the main image the detail appears in a window beside it. A small brightened box on the main image follows the cursor to show what is being displayed in the detail image.

Implementing an image magnifier like this is not trivial but the basic trick is to have two images - one that is small to server as the main image and one that is a very large version of the smaller image. This large image can be set as the background of a `<div>` or similar **element** using **CSS rules** and then changing the position of that background (also specified in CSS properties) can be changed in response to mouse movements over the original image. The `<div>` element will act like a "window" on the image, so if you position the image within the window using top and left coordinates which are negative, only the part of the image that would appear under the `<div>` element will show through the "window". Getting this right might take some math!

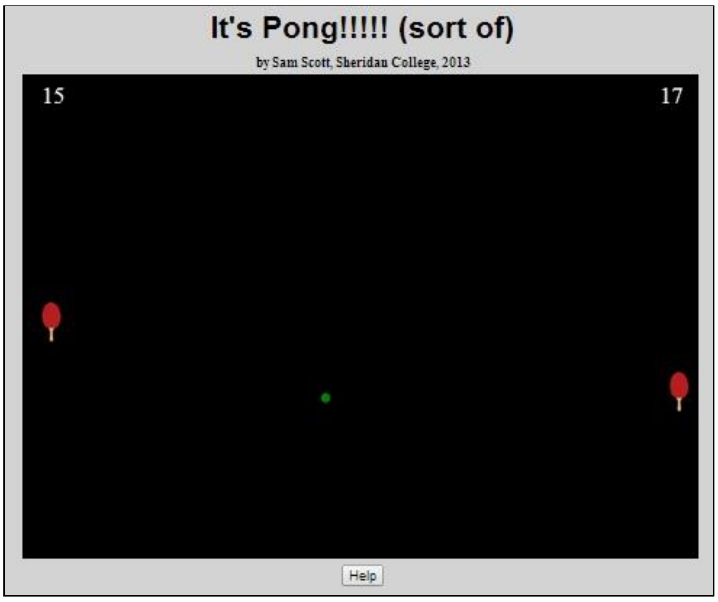
If you can get the movement and display of the image right, there are also lots of other details to worry about including hiding/showing the detail image when the user mouses in and out and deciding what to do when the user approaches the very edge of the original image.

The images shown above are available in the **marsrover.zip** file in the **example pack**.



7.4 Pong

Make a simple live action game using the **HTML5 Canvas**. See the sections on animation, game control, and canvas drawing for ideas on how to make this work.



If you want to program games, it's best to start with recreating something relatively simple, like your own version of **pong**. Other seemingly simple games like **Flappy Bird**, **Space Invaders** or **Pac-Man** are surprisingly challenging for new programmers, but definitely worth a shot if this sort of programming appeals to you.

Of course there are lots of JavaScript games already on the web with code that you can look at, but many of these games are made with frameworks or use advanced programming techniques. To see a JavaScript game with code that might be a bit more understandable, take a look at **Twitty Bird [http://www-acad.sheridanc.on.ca/staff/scottsam/twittybird/]**, this author's version of Flappy Bird.

For some sounds and images to help you create a game like Pong, get the file **pong.zip** from the **example pack**.

8. Server-Independent Web Apps

You now have almost have all the tools you need to create a true **client-side web app**. You can define an interface in HTML and CSS, using form **elements** if required, and then add functionality to it in **JavaScript**. But these web apps are still dependent on a connection to the server. They are not independent in the way that standalone apps are.

1. Web apps don't have any file access. This means you can't open a file on the client machine and store data for the user to access the next time they run your app. The only alternative before **HTML5** was to store the user's data on the server and retrieve it when they return, making the app dependent on a connection to the server.
2. Web apps have to be reloaded each time the browser is restarted. Before HTML5 this meant that if the user loses their internet connection, or if your server goes down, they will not be able to load and run your app.

> ***Java Connection***
>
> **File Access. Java** applications have complete access to the machine they are running on. They can read and write to files and databases. Because **JavaScript** programs run on web pages, it would be too dangerous to give them this kind of access to the **local** machine – some evil person could embed JavaScript into a web page to wipe out the user's hard drive!

But HTML5 is changing all that. The new **web storage** system in HTML5 provides an easy way to store data on the client's machine either temporarily or permanently, and the new **app cache** allows a user to "reload" a web app they have visited before even when there is no connection to the server. These two new capabilities allow us to create fully functional apps in a web browser that are independent of the server.

8.1 Web Storage

Web developers often want their sites to remember users, maintain their settings between page loads, and even keep **local** copies of the data the user entered for the next time they visit. In the past, this was done by setting **HTTP Cookies** on the user's hard drive.

Cookies are small pieces of data (maximum 4KB) that are stored on the client machine by the browser, linked to a particular URL. They are accessible through **JavaScript** and are automatically sent to the server along with every HTTP request. They are typically used to store a unique identifier for each user or session, and then the bulk of a user's data is stored on the **server side** using this unique identifier. This is how sites like **Facebook [http://www.facebook.com]** recognize you when you return.

As an alternative to cookies and server-side databases, **HTML5** offers two new JavaScript objects: `localStorage` for long term data storage and `sessionStorage` for temporary data storage. Each domain name gets a single `localStorage` **object** in the browser as well as a `sessionStorage` object for each open tab. These objects can store up to 10 MB of data, and this data is stored on the client. It is not automatically sent to the server the way that cookies are. The data in `sessionStorage` is automatically deleted when the browser tab is closed, but the data in `localStorage` is only ever deleted when a JavaScript program decides to delete it, or when the user clears their browser history. Otherwise, it's permanent.

Do It Yourself

Go to a site like Google or Facebook in Chrome. Press F12 and go to the resources tab to see what these sites have put in your sessionStorage and localStorage objects.

Now go to the JavaScript console and type the following to see the same information:

```
sessionStorage
localStorage
```

The `sessionStorage` and `localStorage` objects work just like any other object, except that they can only store **string** data. Each item of data is stored under a defined "key", and is accessed the same way as any other object, with square bracket notation or dot notation.

Do It Yourself

On the same web you used for the last DIY box, go to the JavaScript console and type the following:

```
localStorage["name"]="Bob";
localStorage.number=22342;
```

Now type the following to see what you entered.

```
localStorage localStorage.name;
localStorage["number"];
```

Notice that the response from the last line is in quotes. This means that the number you entered was converted to a **string** before being stored.

Now repeat the above commands with `sessionStorage`. Then restart the browser, go back to the page you were on and check the `localStorage` and `sessionStorage` **objects** to see what data remains.

An aside: In the DIY box above, you might notice that we are adding **fields** called `name` and `number` to the `localStorage` object on the fly. The fact that **variables** don't have to be declared in JavaScript (even though you should always **declare** them!) extends to object fields as well. You can add a field to any object just by assigning something to it. You can even add new fields to the `document` object if you want, as shown below.

```
document.newField = 12;
```

Both **web storage** objects also contain a `removeItem` that can be used to remove a single field and a `clear` **method** that can be used to remove all fields at once.

Java Connection

Flexible Objects. In **Java**, once an **object** is created, you can't add any new **fields** or **methods**. In **JavaScript**, any object can be extended with new fields and methods any time. This makes JavaScript very flexible, but also creates some debugging nightmares. If you misspell a field name in Java, you get a **syntax error**. But do the same in JavaScript and you get a new field added to the object!

Do It Yourself

Go back to the page from the last DIY box and open the JavaScript console. Type the following:

```
localStorage.removeItem("name")
localStorage
```

Notice that the name **field** is now gone from the `localStorage` object. Now try this:

```
localStorage.clear()
localStorage
```

Now the `localStorage` **object** should be completely empty.

8.1.1 Initializing Web Storage

Suppose you are counting how many times a user visits your site. On the user's first visit, you will have to create the counter item in `localStorage` and set it to 1. On subsequent visits, you should not re-create the counter. Instead, you should access it and increment it. So you need a way to figure out if the user has been here before.

The code below accomplishes this by taking advantage of the fact JavaScript treats the special **value** `undefined` the same as `false` in a **Boolean** expression, and everything else (other than 0 and the empty string) as `true`. So if the user has not been here before, the else clause will initialize the counter. Otherwise, the counter will be incremented. Notice also the mandatory use of `parseInt` here. What would happen if `parseInt` was left out?

```
if (localStorage.counter)
    localStorage.counter = parseInt(localStorage.counter) + 1;
else
    localStorage.counter = 1;
```

Do It Yourself

Load up **webStorageDemo.html** from the **example pack** to see the above code in action. Hit reload a couple of times to see the counters move. Then open the page again in another tab to see what happens to the counters.

This web page uses `localStorage` to count the total number of visits and `sessionStorage` to count the number of return visits this session. If you open the page in a new tab, the session count will reset, but even if you close the browser and re-open it, the total count will be remembered.

Note the use of the `onload` **event** in the `<body>` **element**. This makes sure that the count() **function** will run once on each page load, but not until after the entire `<body>` is loaded.

8.1.2 Advice on Sharing Web Storage

The one drawback to `localStorage` is that if your site is on a domain that has multiple users accounts (like `mobile.sheridanc.on.ca`), somebody else's **web app** could access, clear, and overwrite your `localStorage` data. There is nothing that can be done to avoid this other than buying your own private domain, but failing that, it's a good idea to name your data in such a way that is likely to be unique.

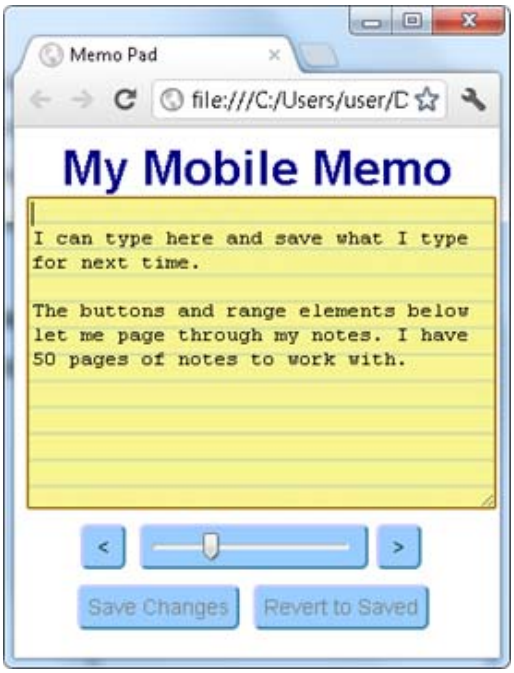
For example…

```
localStorage["App2938475SamScott_vehicle"]="Aveo";
```

… is less likely to be tampered with than…

```
localStorage["vehicle"]="Aveo";
```

<exercises section='8.1.2'>



1. Write a simple standalone **web app** with two pages. On page one, have the user fill in a form with their name and the color shirt they are wearing. If they have been here before, you should fill in their name automatically. Store their name in `localStorage` and their shirt color in `sessionStorage`. Provide a link to page 2. On page 2, you should greet the user by name and tell them whether you like their shirt or not, based on the color they entered last time (e.g. write an **if statement** that only says you like their shirt if they say it was red). Provide a link back to page 1.

2. Write a mobile "memo pad" application. This application should allow the user to type into a text area. The text should be stored in `localStorage` and displayed when they return for them to read or modify. Make it look professional (like a real app you might see on a mobile phone).

3. As an added challenge to question 2, allow 50 pages of memo storage with buttons or other form **elements** to allow them to flip through the pages.

</exercises>

8.2 The App Cache

With `localStorage` you can create a standalone **web app** like the mobile memo pad above, but you're still dependent on the server because each time the user wants to load your app, they have to send an **HTTP** request and download it again. It would be much better if the app could be stored locally like a native app. And that's exactly what the **HTML5 app cache** does.

To use the app cache, you need two things:

1. An app cache "manifest" file, and

2. 2. A `manifest` **attribute** on the `<html>` tag of the site's front page.

We'll cover these in reverse order.

8.2.1 The `manifest` Attribute

The `manifest` attribute is part of the `<html>` tag. It specifies the location of a manifest file, like so:

```
<html manifest="myManifest.appcache">
    … (your page goes here) …
</html>
```

The name of the manifest file doesn't really matter, but the `.appcache` file extension is recommended to ensure that the server marks it with the **MIME Type** `"text/cache-manifest"`. The usual rules apply for specifying the file path (same as the rules for the `href` or `src` attribute). So you could also put the `.appcache` file in its own folder as long as you specify the path to it. (In the above example, the manifest file is in the same folder as the HTML file that **references** it.)

The presence of a manifest attribute in an HTML file will cause the manifest file to be automatically stored locally to be accessed when the user is offline. The contents of the manifest file specify which other files should

also be stored locally along with the manifest.

If your web app uses the app cache, it will only be loaded one time. Future visits to the page will load the remote manifest file and check it against the locally stored copy. If they match, none of the cached files will be re-loaded. **Local** copies will be used instead. So using the app cache also saves bandwidth. But the main point is that if there is no connection or the server does not respond, the app will still load using cached copies of all the files.

8.2.2 The Manifest File

Here is the content of a simple cache manifest file:

```
CACHE MANIFEST
#May 29, 2012, 6:34 pm
js/memo.js
css/memo.css
images/116_2.jpg
```

The first line is required. The line that starts with a # character is just a comment. The remaining lines specify which files on the site are to be cached (along with the HTML file that loaded this manifest). These are file paths relative to the location of the manifest file.

You can also include a NETWORK section. This section specifies files that must be loaded fresh from the server every time. And you can also include a FALLBACK section that specifies a file to be loaded if a page is not accessible. You can read more about these options on the **w3Schools HTML5 Application Cache** page.

> *Do It Yourself*
>
> You can only test the app cache if you have access to a web server – either a remote web server on a different machine, or a development web server like WampServer or **XAMPP [http://www.apachefriends.org/en/xampp-windows.html]** running on your own machine.
>
> If you have a server, upload the files in the **appCacheExperiments** folder of the **example pack** and try them out. Try accessing them each while on line, then sever your network connection and try again. Try to figure out why each one loads or does not load.

Important: Remember that returning users check the manifest first and if it has not changed, they will load local copies of the files only. So when you update your site, it is crucial that you update your manifest file as well, even if all the file names are the same, you must at least change one comment line in the manifest file. If you don't do this, returning users will not see the updated site.

<exercises section='8.2.2'>

1. Update your two-page app from section 9.1 to use the **app cache** so that it still loads even if the user is off line.

2. Update your memopad app so it works off line as well.

</exercises>

9. Advanced DOM Manipulation

It's time to get more serious about the HTML **DOM**. Up until now you have navigated the DOM by using **methods** like `getElementById` or `querySelectorAll` to pull out specific **elements**. You have also made changes to the DOM by accessing a node's `innerHTML` **property**, or by changing **attributes** and CSS style properties. These techniques can get you a long way, but there will be times when you will want to navigate and manipulate the DOM in more sophisticated ways.

For example, this textbook uses **JavaScript** to build and insert its own contents section after it is loaded (check the page source and you will not see a **table** of contents there). This could be done by getting the `innerHTML` of the `<body>` and then building and inserting a new substring for the contents section, but it would be tricky to reliably find the right place in the **string** for the insertion, and building such a long HTML string would be tedious and prone to error. It's easier to do it by creating new DOM **nodes** from scratch and using the DOM `appendChild` and `insertBefore` methods. You'll learn how to do that in this chapter.

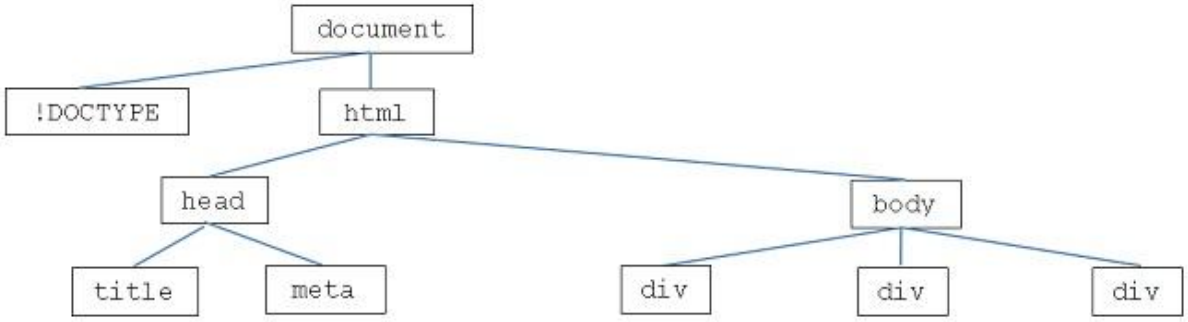
This textbook also intelligently resizes images using JavaScript when the browser width changes so that an image is always either its original size, or just big enough to fit inside the `<div>` that contains it. This would be very tricky to do using `innerHTML`. It's much easier to do with the techinques of **tree traversal** you will learn in this chapter.

9.1 The Full DOM

Consider the HTML source below (**domExample.html** from the **example pack**).

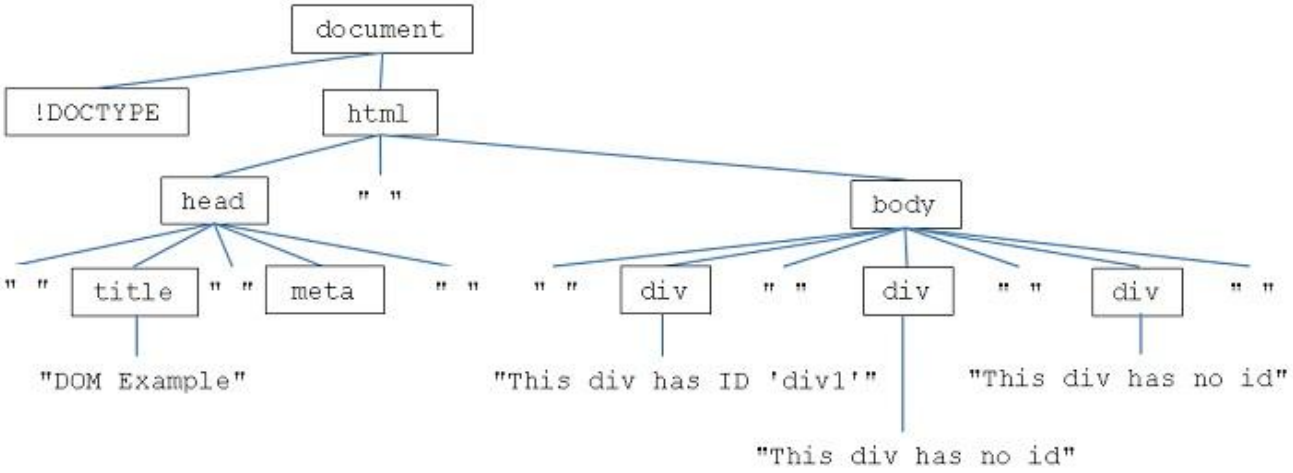
```
<!DOCTYPE html>
<html>
    <head>
        <title>DOM Example</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <div id="div1">This div has ID "div1"</div>
        <div>This div has no ID</div>
        <div>This div has no ID</div>
    </body>
</html>
```

Following the analysis we developed in Chapter 4, we should expect the **DOM** for this document to look like this:



Here's a terminology reminder. In the picture above, there are 10 nodes. The **root node** is `<document>`. It has two **children** or **child nodes**. Each of the children share the same **parent**. The line from parent to child means that the child is contained within the parent. If there are many children, they are usually shown in order from left to right. Nodes that share a parent are **siblings** (this is the gender-neutral term for "brother" or "sister"). Each of the children of the `<body>` **node** has two siblings. Each Node contains a **field** for every **attribute** as well as `style` and `className` fields.

So far, so good. But we were oversimplifying in Chapter 2. In fact, the browser will create a surprisingly complicated DOM from this source, as shown below.



The things in quotation marks are TextNodes. They are a different kind of **object** from a regular DOM Node. They don't have attributes, style, className or `innerHTML`. Rather they just have a field named `data` that contains a **string** representing the text. When the browser reads the document, a TextNode will be created for almost every sequence of characters in the document that is not enclosed within < and > angle brackets, including every stretch of whitespace characters at the beginning or end of a line. The only stretches of whitespace that don't get turned into TextNodes are those that are not inside or between the `<head>` and `<body>` tags.

<exercises section='9.1'>

1. Go back to the **helloWorld.html** file from the **example pack** that we looked at in Chapter 2. Look at the source and use it to draw the full **DOM tree**, including all text **nodes**.

</exercises>

9.2 Traversing the DOM

A **DOM object** in **JavaScript** is a collection of **Node** objects that are all linked together into a **tree** structure with the `document` object at the root. In order to effectively manipulate the DOM, you will sometimes have to understand the various ways to navigate this structure, moving from node to node by following the links. The following **fields** and **methods** will help you with tree **traversal**.

`childNodes`:	returns an array of all child nodes of the current node
`firstChild`:	returns the first child of the current node
`lastChild`:	returns the last child of the current node
`nextSibling`:	returns the sibling to the right of the current node
`previousSibling`:	returns the sibling to the left of the current node
`data`:	if this is a TextNode, gets the text

Do It Yourself

Load up **domExample.html** from the **example pack**, open a JavaScript console, and try the following. Try to predict the result of each before you hit enter:

```
document.childNodes
document.firstChild
document.lastChild
document.childNodes[0]
document.lastChild.childNodes
```

Can you explain the result in each case?

Now try `document.lastChild.childNodes[0].childNodes` and explain the result.

Now for a challenge. Find as many ways as you can to get the TextNode from the second `<div>` **element**. Remember that the TextNode is a special child **node**, not an **attribute** or a **value** (i.e. don't use `innerHTML`).

<exercises section='9.2'>

1. Load **domExample.html** from the **example pack** into a browser. Use the `document` **object** along with `childNodes`, `firstChild`, `lastChild` and other **fields** you learned about in the previous section to verify that the document does indeed have the structure shown above.
2. Check the structure you drew for **helloWorld.html** using the same **method**.
3. In the last DIY box you were asked to find as many ways as you can to get the TextNode from the second `<div>` **element**. How many ways do you think there are in total to do this using the **DOM** navigation methods give above? Justify your answer.

</exercises>

9.3 Getting a DOM Node from an Event

There is another way to get hold of a **node** from the **DOM** that is sometimes useful. Whenever you register an **event** handler on an **element** (such as a click event handler on a button or a mouse over event handler on a **table** element) you have the option of using an `event` parameter.

Do It Yourself

Try the following in a browser console for this book. It will attach an event handler to this DIY box.

```
document.getElementById("test5").addEventListener("click", function (event)
{
    alert(event.target.innerHTML);
});;
```

Now click this DIY box to see the result. Depending on where you click, you might get the entire `innerHTML` for this DIY box in an `alert` window, or you might just get part of it. Try clicking in different places. What do the results tell you about the `event.target` **object**?

One very important piece of information in the `event` **object** is the **reference** to the DOM node that triggered it, known as the "target". This DOM node object is stored in the `event.target` **field**. In the above example, the user will get an alert with the `innerHTML` contents of the node that triggered the event.

In addition to `innerHTML`, you could also use any other DOM fields or **methods** with `event.target`, such as `event.target.parentNode.innerHTML`, `event.target.nextSibling.id`, and so on.

This can be extremely useful when you are adding events to a page with lots of similar elements. In the above example, there could be 500 different `<div>` elements on the page, and without `event.target`, you would need to write 500 different event handlers.

For a complete reference on the DOM Event object and all its associated fields and methods, go to the **w3Schools event object reference**.

<exercises section='9.3'>

1. The file **domExample2.html** in the **example pack** shows a similar example to the one above. Add some `<div>` **elements**, each with different `id` **attributes**, and then modify the **event** handler to display the id attribute of the element that was clicked.
2. Now put `<input type="button" value="ID">` beside each of the `<div>` elements, and add click handlers to all these buttons that display the id attribute of the `<div>` beside them (hint: the `<div>` for each button should be its `previousSibling`).

</exercises>

9.4 Using innerHTML to Modify the DOM

There is one really quick way to make modifications to the **DOM**, and that's to alter the `innerHTML` of a **node**. You've probably already used this **field** to change the text of an **element**, but it's actually much more powerful than that. With the `innerHTML` field of a node, you can quickly delete all its child nodes, replace them with new ones nodes, or append child nodes to the end of its existing list of child nodes.

9.4.1 Delete All Child Nodes

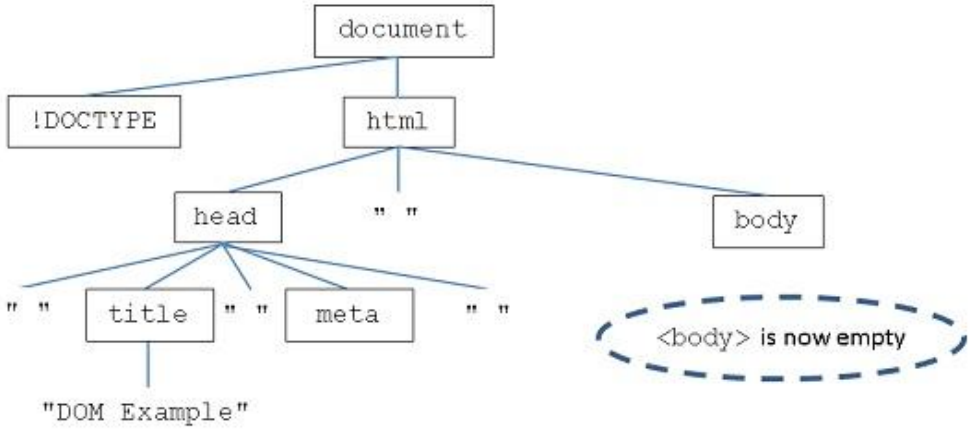
If you assign an empty **string** to the `innerHTML` field of any DOM node, you will delete every node that is a child of that node.

Do It Yourself

Load **domExample.html** from the **example pack** into a browser and try one of these in the JavaScript console.

```
document.querySelector("body").innerHTML = "";
document.lastChild.lastChild.innerHTML = "";
```

After the above operation, the DOM tree looks like the one below.



Do It Yourself

You can verify the above claim in the **elements** tab of the Chrome developer tools, or by inspecting the `document` **object**.

Go to page source, and compare that to what you see in the Elements tab in Chrome's developer tools. Can you explain the difference?

9.4.2 Replace All Child Nodes With a Text Node

If you put some plain text in the string you assign to `innerHTML`, you will replace all the children of that node with a single Text Node (or insert a new Text Node if the element had no children).

Do It Yourself

Try the following on domExample.html.

```
document.querySelector("body").children[1].innerHTML="Hello";
```

What does the document **tree** look like now? Now try the same thing with the `<body>` **element**. Now what does the tree look like?

Compare the page source view of the document to the view of the document in the Elements tab of the developer console in chrome. Can you explain the differences?

9.4.3 Replace All Child Nodes With New Nodes

In the example below the Text Node child of the selected `<div>` element will be deleted and replaced with two `<p>` nodes, each with their own Text Node contents.

Do It Yourself

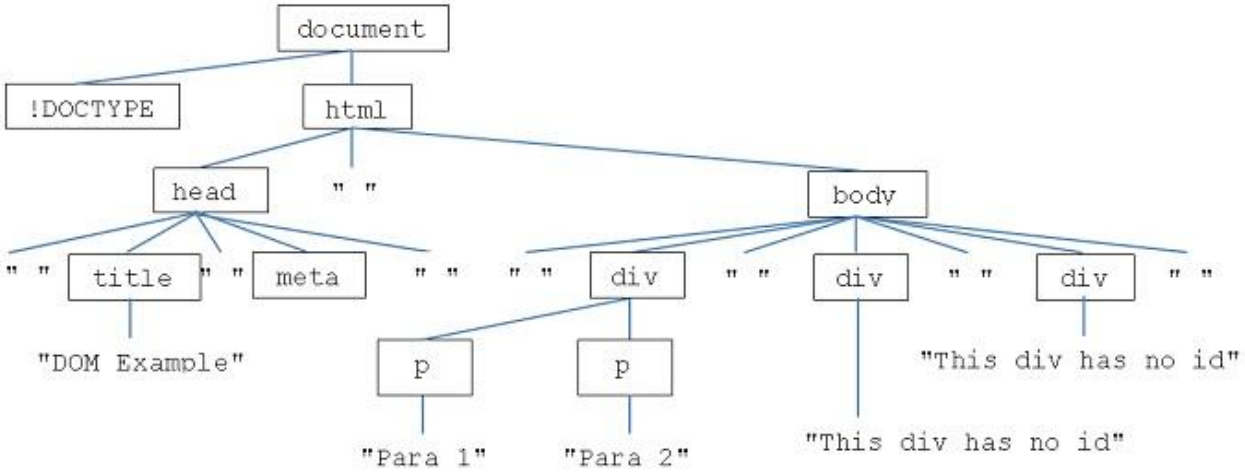
Try the following in a browser console for **domExample.html** from the **example pack**.

```
document.getElementById("div1").innerHTML = "<p>Para 1</p><p>Para 2</p>";
```

What does the **tree** look like now?

This is true DOM manipulation because you have removed one DOM element and added two new ones in its place.

The new **tree** looks like this:



9.4.4 Append New Child Nodes

Finally, you can always append new DOM elements by using += instead of =.

Do It Yourself

Try the following in a browser console for **domExample.html** from the **example pack**, and then draw the resulting **DOM tree**.

```
document.lastChild.lastChild.firstChild.nextSibling.innerHTML += "<p>Here
is a <a href='http://www.google.ca'>link</a></p>";
```

<exercises section='9.4.4'>

1. Add a button to domExample.html. Every time the button is pressed, add a new `<div>` **element** to the `<body>` element, each with a text **node** "New Div X" where X is the number of the new node, counting upwards from 1.

</exercises>

9.5 Creating New Nodes From Scratch

Manipulating the `innerHTML` **field** is a very powerful technique for changing the HTML **DOM** and is often easier than other **methods**. But it can get complicated in some cases, and it is often easier to create the new DOM **elements** yourself rather than trying to manipulate the innerHTML **string**.

To create a new DOM **Node**, you can use the `document.createElement` method, giving it the tag name as a string parameter. Here are some examples:

`d=document.createElement("div");`

creates a new `<div>` node

`e=document.createElement("a");`

creates a new `<a>` node

`e.href="http://www.google.ca";`

sets the `href` attribute of the new `<a>` node

`e.innerHTML="Click for Google."`

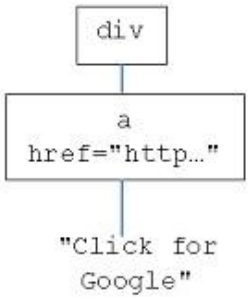
adds a text node to the new `<a>` node

The above code creates two new DOM Nodes and stores them in **variables** named d and e respectively. The `<div>` node in d is empty, but we can make the e element its child by using the `appendChild` method.

`d.appendChild(e);`

adds e as the last child of d

What we now have is a little piece of a DOM, depicted at right. At the moment it is not included in the tree, but we could easily add it as the last child of the body of any document by using appendChild again, like this:




```
document.body.appendChild(d);
```

Note the use of the body field above. This is a special field in the `document` **object** that takes you straight to the `<body>` element. There's also a `head` field.

Do It Yourself

Try out this code in a JavaScript console for **domExample.html** from the **example pack**.

```
var d=document.createElement("div");
var e=document.createElement("a");
e.href="http://www.google.ca";
e.innerHTML="Click for Google.";
d.appendChild(e);
document.body.appendChild(d);
```

Check that the new `<div>` is inserted and that the link works. Then use the same code twice to add two identical new **elements**. What if you wanted to add 100 new identical elements? How could you do it?

It is also possible to insert a new child anywhere within the list of existing child nodes, using the `insertBefore` method. The `insertBefore` method requires three things: the parent Node, the new Node to insert, and the existing child of the parent you want to insert in front of, like this:

```
parent.insertBefore(newNode, existingChild);
```

Do It Yourself

In a **JavaScript** console for **domExample.html** from the **example pack**, create the new **node** d as before:

```
var d=document.createElement("div");
var e=document.createElement("a");
e.href="http://www.google.ca";
e.innerHTML="Click for Google.";
d.appendChild(e);
```

Now get the second div and the body **element** and store them as `parent` and `existingChild`.

```
var parent = document.body;
var existingChild = parent.childNodes[3];
```

Now insert the new node before the second `<div>` using `insertBefore`.

```
parent.insertBefore(d, existingChild);
```

Now follow the above example to create and insert a node between the second and third `<div>` elements.

<exercises section='9.5'>

1. Go back to **domExample.html** from the **example pack** and add another button. This button should create a **DOM** fragment like the one described above (a `<div>` **element** containing an `<a>` element containing a Text **Node**) and append it to the end of the `<body>` each time the button is clicked.
2. Add another button to **domExample.html**. This one should add a four-element checkerboard to the document every time it is clicked. Remember you can use the `style` **field** of the new nodes to specify style properties, or you can use the `className` fields to assign **classes**.

</exercises>

9.6 Removing a Single Child Node

Finally, it is also possible to remove particular **nodes** from the **DOM** using the `removeChild` **method**. The only tricky thing about `removeChild` is that you have to specify a reference to the child to remove, like this:

```
document.getElementById("body").removeChild(x);
```

In the above, x has to be a variable containing the child to remove. For example:

```
x = document.getElementById("body").childNodes[3];
document.getElementById("body").removeChild(x);
```

One thing you often might want to do is delete an element that the user clicks or mouses over. If you have the event.target and you want to remove it, you can do it like this:

```
event.target.parentNode.removeChild(event.target);
```

<exercises section='9.6'>

1. Again, go back to **domExample.html** from the **example pack** and add a `mouseover` handler that deletes a `<div>` when that `<div>` is moused over.

2. See if you can combine the above with the button from the previous exercise that adds `<div>` **elements**. When this is finished, you should be able to add new `<div>` elements by clicking the button, and then remove them by mousing over them.

</exercises>

10. AJAX

*The text in this chapter mirrors the text in Chapter 12. In this chapter, you learn to use **AJAX** with raw **JavaScript**. In Chapter 12, you learn to use AJAX with the **jQuery** library and you also learn a little bit about how to work with **XML** data.*

Even if you don't know what AJAX is, you have experienced its effects. When you start typing into the **Google [http://www.google.com]** search box, it doesn't take long before you start getting some suggestions about what to search for. When you view your list of friends on **Facebook [http://www.facebook.com]**, it starts you with a small list, and when you scroll down, it loads some more for you automatically (often without you even noticing). Both of these are examples of AJAX in action.

Before AJAX, if you wanted to get some more content for the user, you had to get the browser to load and display a whole new page. With AJAX, you can launch an **HTTP** request from JavaScript. Then when the response data comes back from the server, you can use JavaScript code to incorporate this data into the page without reloading it.

10.1 The Basic Idea

The term **AJAX** was coined by a web developer named **Jesse James Garrett [http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications]** in 2005. AJAX is a term for a new way of thinking about and making use of a collection of already existing technologies. It's not a new language or library or plugin. AJAX stands for **Asynchronous JavaScript and XML**. Let's break this phrase down.

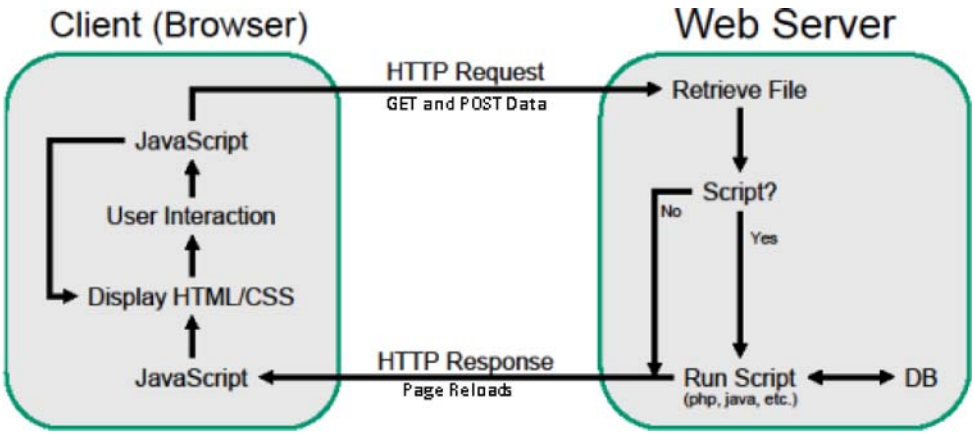
JavaScript: You already know what this is.

Asynchronous: Asynchronous processes are processes that happen outside of the regular flow of the program. In Asynchronous programming, you make some kind of request that will take time to complete, and then continue on with other tasks. In many cases you also specify a **callback function** to be triggered when the asynchronous task is finished. You saw something a bit like this when you used the `setTimeout` function in Chapter 5. This function starts a timer and then allows the app/page to continue running and responding to the user while it waits for the timer to go off, at which point the function you specified gets called. If the timer task was not asynchronous, everything would be at a standstill until the timeout **event** happened.

XML: You may have encountered this before. **XML** stands for eXtensible **Markup Language**. It's a generic markup language for representing structured data. It looks a lot like HTML, but it can be used to represent any kind of data (sports scores, the weather, a shopping list, etc.). XML is one possible format for the data that comes from the server in response to an AJAX request. But you can get your data from the server in any format you want. It can be XML, HTML, **JSON** (**JavaScript Object Notation**) or even just plain text. No matter what kind of data you're getting, most developers would say that if you're getting it asynchronously by launching a request from a **JavaScript** program, you're using AJAX.

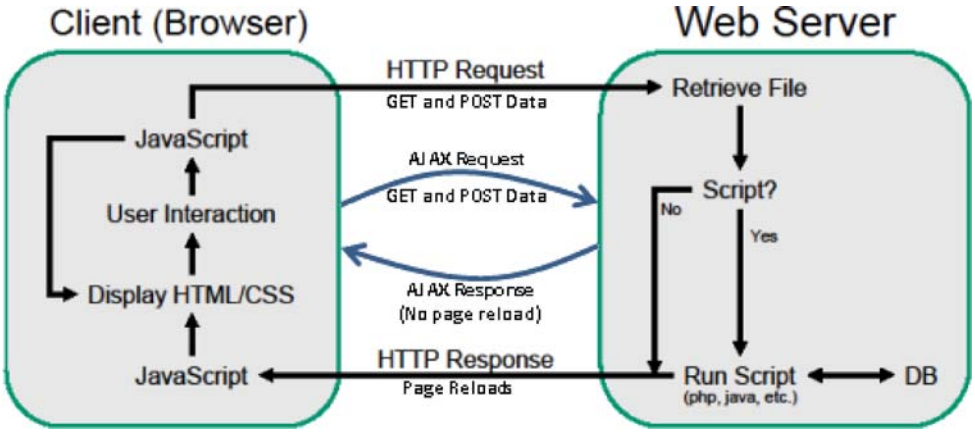
10.2 A New Web App Architecture

Up until now, we have been assuming a **web app** architecture that looks like this.



In this architecture, an **HTTP** request is launched by the user either by typing a URL, clicking a link, or submitting a form. (HTTP Requests are also generated by the browser to automatically to load images, CSS files, **JavaScript** files and other resources. But I'm ignoring that minor complication in the story I'm telling here.) The request goes to the server where a file is retrieved and/or a script is run to generate the HTTP Response. When the response arrives, the browser interprets the HTML and CSS in the document to display the page, and executes the JavaScript commands in the page. Then it continues to run JavaScript in response to user interactions and other **events**, refreshing the page view when something changes. It doesn't go back to the server again unless the user clicks a link, types in a URL, or submits a form.

But with **AJAX**, the life cycle looks a bit different:



An AJAX request is an HTTP Request just like any other, and can have **GET and POST** parameters associated with it just like any other. At the server it is treated in exactly the same way as any other HTTP request. The only difference is where it came from: it was launched by a JavaScript **function**. Since HTTP Requests can take time to complete they are done asynchronously and the programmer register a listener (a.k.a. **callback** function) to deal with the response when it comes. (What would happen if the requests were not **asynchronous**?)

The HTTP Response from the server is also the same as any other. The only difference is that when the response arrives at the client, it does not trigger a page reload. Instead, the content of the response is passed to the listener as a parameter.

10.3 How AJAX Works

AJAX queries are launched using the **XMLHttpRequest API** that is built in to all modern browsers. To launch an AJAX request, you create a new `XMLHttpRequest` **object** and build up your request over several lines of code, adding headers, **GET and POST** parameters and registering a listener **function** to be called when the response arrives. Then you launch the request and the page continues operating while the request is in progress.

10.3.1 A Data Source

You can't do much in AJAX without access to a data source on a server somewhere. There are many excellent sources of data on the web that are free to use in your AJAX programs. For example Yahoo's **YQL [http://developer.yahoo.com/yql/]** (**Yahoo Query Language [http://developer.yahoo.com/yql/]**) can be used to retrieve information about geographic locations, weather, music, maps, and a number of other domains. But for now, you need an easier-to-use source of information.

> ***Do It Yourself***
>
> In the **example pack**, you will find a **servers** folder containing two PHP scripts: a "time server" named **servers/timeServer.php** and a "Lorem Ipsum server" named **servers/loremIpsumServer.php**.
>
> Load the two server files by clicking on the links above (don't try to open them on your own machine). Then read over the **APIs** they display and try them out. You can experiment with the GET parameters by typing them into a URL. For example, if you wanted to set the `pstart` parameter to 2 and `plength` to 1, you could do it like this:
>
> ```
> servers/loremIpsumServer.php?pstart=2&plength=1
> ```
>
> And if you want to test POST parameters with the Lorem Ipsum server, you can use the **servers/loremIpsumPostTest.html** file which uses a `<form>` **element** with its `method` **attribute** set to "post" to send requests to the Lorum Ipsum server.
>
> If you've forgotten what all this means, you may need to review **w3schools** or a similar source to refresh your memory about **GET and POST** and **PHP Form Handling**.

10.3.2 AJAX GET Requests

To send an AJAX GET request, you need to execute a **JavaScript statement** to perform each of the following steps:

1. **Create an `XMLHttpRequest` Object:** This is the object you will use to send the request.
2. **Open the Connection:** You will have to specify a URL which includes the GET parameters for your request. The URL can be absolute or it can be relative to your current location.
3. **Register an Event Listener:** The event listener function will be called whenever the status of the request changes.
4. **Send the Request:** The final step sends the request. Then the page continues to function while waiting for the response.

Do It Yourself

Load **ajaxTime2.html** from the **example pack** into a desktop browser. Ignore the contents of the page for now. Just open the JavaScript console and launch your first AJAX request by executing the code below for the four steps outlined above:

```
// step 1
var xmlhttp = new XMLHttpRequest();

// step 2
xmlhttp.open("GET", "servers/timeServer.php?request=time", true);

// step 3
xmlhttp.addEventListener("readystatechange", function () {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200)
        alert(xmlhttp.responseText);
    })

// step 4
xmlhttp.send();
```

The above should pop up the current time for you in an alert box.

In step 2, there are three parameters. The first specifies the type of request ("GET"). The second is the URL which includes the GET parameters. The third is a **boolean** that is set to `true` to make sure this is an **asynchronous** connection.

In step 3, you are registering a listener for a special **event** called the `readystatechange` event. The listener is the **callback** function and it makes use of the xmlhttp object created earlier to figure out whether it should take action or not. The `readystatechange` event will happen multiple times throughout the life of the **HTTP** request, and will trigger the listener function each time with a different `readyState`. But for most purposes only state 4 ("Response Ready") is useful. When state 4 is detected, the next thing to check is the HTTP status that was returned. Status 200 means "OK", 404 means "Page Not Found", etc. Usually, you just check for state 200.

Do It Yourself

Paste the commands from the last DIY box into the `<Script>` element of a new web page and add the following line to the callback function to trace every call to the listener:

```
console.log(xmlhttp.readyState+" "+xmlhttp.status);
```

You will also have to replace the URL with the absolute address of the time server (`http://javascript.sheridanc.on.ca/examples/servers/timeServer.php/`)

Now load the page and check the console to see the call trace

Now try again but change the URL to invalidate it to view the calls of calls to the listener **function**.

Do It Yourself

Try out the form interface provided on the **ajaxTime2.html** page. Do you think you could write the code for this page now, given what you know about **AJAX** and **jQuery**?

10.3.3 AJAX POST Requests

For AJAX requests, the distinction between GET and POST may seem less important since the parameters you send will never be visible in the browser. However, you should still use POST requests in the following cases:

1. Sending Large Amounts of data (GET has a 1KB limit).
2. Sending Requests to Update a Database (GET requests might access cached copies of the page, in which case the database update will not go through).

The process for sending POST data is very similar to the process for GET data, but you must set a special header on the request to specify that there are parameters in the body of the message, and you specify the parameters when you call the send **method** rather than in the URL.

Do It Yourself

Paste the commands below into the `<Script>` element of a new web page. The bold-faced parts show the stuff changed for a POST request.

```
var xmlhttp = new XMLHttpRequest();

xmlhttp.open("POST", "servers/timeServer.php", true);

xmlhttp.setRequestHeader("Content-type","application/x-www-form-
urlencoded");

xmlhttp.addEventListener("readystatechange", function () {
    if (xmlhttp.readyState === 4 && xmlhttp.status === 200)
        alert(xmlhttp.responseText);
    })

xmlhttp.send("request=time&timezone=Canada/Pacific");
```

The above should pop up the current time for you in an alert box.

Now try changing some of the parameters to get the day or month, or use a different time zone.

Now try changing the listener **function** so it puts the text somewhere on the screen (maybe in the `innerHTML` of an `<h1>` **element**, or in the **value attribute** of an `<input>` element).

10.3.4 AJAX On a Web Page

The JavaScript code below is from the **ajaxTime2.html** example. Take a moment to read it over and try to understand it.

```
window.addEventListener("load", function () {
    function launchAjax() {
        var address="servers/timeServer.php";
        var timezone = document.querySelector("input[name=timezone]").value;
        var request = document.querySelector("select[name=type]").value;
        var delay = document.querySelector("input[name=delay]").value;
        var URL = address + "?timezone=" + timezone + "&request=" + request +
        "&delay=" + delay;

        var xmlhttp = new XMLHttpRequest();
        xmlhttp.open("GET", URL, true);
        xmlhttp.addEventListener("readystatechange", function () {
            if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
                document.getElementById("result").innerHTML =
                xmlhttp.responseText;
            }
        });

        xmlhttp.send();
    } document.getElementById("ajaxButton").addEventListener("click",
    launchAjax);

});
```

The first thing to note is that all the code is inside the `window` object's `ready` event handler. This is just following the best practices that were introduced in chapter 5.

The second thing to notice is that the `launchAjax` function contains calls to the `XMLHttpRequest` object that should look very familiar from the previous section. The only difference is that one more parameter is set (`delay`) and the **values** for each parameter are coming from the form **elements**. Also, the listener function is placing the data into the `innerHTML` of the element with an `id` of "result".

The **syntax** `$("[attribute=value]")` is used to select all the elements with a particular **attribute** value. In this case it's being used to retrieve named form elements. The programmer could just as easily give them `id` attributes and use `document.getElementById`.

After the definition of `launchAjax`, the function is added as a `click` event handler onto the element with an `id` of "ajaxButton".

Now you have two data sources (**servers/loremIpsumServer.php** and **servers/timeServer.php**) and a worked example (**ajaxTime.html**). It's time to build something of your own.

<exercises section='10.3.4'>

1. Take the file **servers/loremIpsumPostTest.html** from the **example pack** and change it so that instead of launching a brand new POST request and reloading the page, it instead launches an **AJAX** query and displays the result below the form.

2. In the exercises folder, look at the file **findLorem.html**. You have the HTML and CSS here for a game. You just have to fill in the **JavaScript**. This game will pull out a **random** piece of Lorem Ipsum text from the server, using an AJAX query. Then it will allow the user to guess where the text came from, using more AJAX queries to check whether they were right or not. See the comments in the **findLorem.html** file for detailed instructions.

3. In the example pack you will find **more.html** and its associated files. Add the necessary JavaScript code so that this page allows the user to load one paragraph at a time from **servers/loremIpsumServer.php**. When they first load, they should see the first paragraph, retrieved with AJAX. Then whenever they hit the "more" button, they should get the next paragraph in sequence.

</exercises>

10.3.5 What To Do While You Wait

Because networks can be slow, AJAX requests can sometimes take a few seconds to complete. If things don't happen instantly, it can cause the user to start pressing buttons over and over again, or behaving in other destructive ways which could end up launching more and more AJAX requests. This is a problem you have to think about whenever you use AJAX.

Do It Yourself

Load up **ajaxTime2.html** from the **example pack**, set the delay on the form to 15 seconds, and set the request to "time", then hit the "**AJAX**!" Button. This will send a request to the **servers/timeServer.php** page which instructs the php scripts to pause for 15 seconds before completing. So this simulates a delay over a slow network.

While you're waiting for your first request, set the delay back to 0 and try some different requests (anything other than "time"). After 15 seconds, the original request will get its response and this will disrupt what you are currently doing.

The usual solution to this problem is to control the user somehow while you are waiting for the AJAX request to complete. For example, if the user clicks something to load more content, you could do the following:

1. In the click handler, disable the button the user just clicked, and perhaps give them some visual indication that we are waiting for something to load.

2. In the listener function that responds to the AJAX request, re-enable the button.

This will ensure that only one AJAX request is dealt with at a time.

Another approach might be to have a **global variable** with a name like `waitingForAJAX`:

1. Initialize `waitingForAJAX` to false.

2. In the click handler, check if `waitingForAJAX` is `true`. If it is, give the user an error message or simply ignore the request. If it's `false`, launch the AJAX query and set `waitingForAJAX` to `true`.

3. In the listener function, set `waitingForAJAX` back to to `false` once the ready state is 4.

You could also consider placing an animated gif on the page like in the example below.

Load

<exercises section='10.3.5'>

1. Follow up on exercise 1 from the previous section by adding more code to the **servers/loremIpsumPostTest.html** page so that the submit button is disabled until the **AJAX** request has completed. Then test it by setting the `delay` parameter.

2. Add code to the **ajaxTime.html** page so that the user has to wait until the previous request has completed before a new one will be launched. This time don't disable the button - instead set a **global boolean variable** and if the user hits the button again before the AJAX request is completed give them an error message in an alert box.

3. Repeat the above, but this time use the other suggested strategy for what to do while you wait.

4. Alter your solution to the **more.html** exercise from the previous section. Set up the AJAX requests to include a short delay and make sure you disable the user while they wait for their next paragraph.

5. Wouldn't it be nice if you could just call a **function** like this and have it take care of all the messy details for you: `sendAjax(type, url, parameters, callback_function);` You should write one! Then put it in its own .js file, add it to one of your pages and try it out. Make sure to test both **GET and POST**.

</exercises>

11. The jQuery Library

Now you're able to write some powerful, standalone **web apps** using forms, **events** and the HTML **DOM**. But there are easier ways to program web apps than plain JavaScript. This chapter is about **jQuery**, a public domain JavaScript library that has caught on like wildfire and is changing the way JavaScript code is written. **JQuery [http://jquery.com/]** provides a powerful set of **methods** for DOM manipulation and event handling that solve many of the **cross-browser compatibility** issues that arise when using plain **JavaScript**. To use jQuery effectively, you need to finally understand JavaScript **functions** a little better, so this chapter will cover that as well.

This chapter will give you the basic tools you need to use jQuery, but I will leave a lot of the details for you to explore yourself. The jQuery **API** is the definitive source of information about jQuery, and the official **Learn jQuery [http://learn.jquery.com/]** site has some nice tutorials. There are also some useful tutorials in the **w3Schools Learn jQuery** section.

The jQuery library is publicly available for download, but if you are using **NetBeans [https://netbeans.org/]** version 7.3 or later, it is also available automatically as an add-in library for projects of type HTML5.

11.1 Selectors and Effects

The **jQuery** library extends the power of **CSS selectors** in **JavaScript**. Recall that a CSS selector is the first component of a **CSS rule** – the part that specifies which **elements** the rule applies to. For example, the CSS rule below uses the selector `"p, #myDiv, span.movie"` to specify that it applies simultaneously to all elements of type `<p>`, the unique element with its `id` **attribute** set to `myDiv`, and any `<span>` element with a `class` attribute that includes `movie`.

```
p, #myDiv, span.movie {
    color:red; width:100%;
}
```

In plain old JavaScript, if you wanted to do make same changes to a diverse collection of elements like this (e.g. suppose you wanted to turn them all green) you would need to write one expression or a set of expressions to get the collection of elements, and then you would need to write a loop to apply the changes to the collection. But now you can use jQuery selectors to do it all in one line, like this:

```
$("p,#myDiv,span.movie").dosomething();
```

Do It Yourself

Here's a simple example that you can use on this book if you are viewing it on line.

Open the JavaScript console and type the following to turn all the headers green.

```
$("h1,h2,h3").css("color","green");
```

Note that entering **jQuery** commands in the console will only work if the current page includes a link to the **jQuery [http://jquery.com/]** library. Since this book makes use of jQuery, it can be used as a jQuery testbed.

All the DIY boxes in this book are `<div>` **elements** with the `class` name `"DIY"`. Based on the above example, see if you can turn the backgrounds of all DIY boxes in the document pink.

You can always reverse any changes by hitting "reload".

11.1.1 The jQuery Sandbox

In what follows, we will use the **jQuery [http://jquery.com/]** "sandBox" a lot. It's just a page with the jQuery library loaded so that you can try out commands in the JavaScript console. The main page is **jQuerySandbox/index.html** from the **example pack**. When you explore the **jQuerySandbox** folder and files, you will notice that the jQuery library is included in the `js` folder, and that the `index.html` file loads the jQuery library using the following line:

```
<script type="text/javascript" src="js/jquery-1.10.0.js"></script>
```

This line causes all the code in the jQuery library to be included in the sandbox page. If you want to use jQuery in your own pages, you have to do two things:

1. Download the library and put it somewhere in your project; and
2. Link to it in a `<script>` element as above.

Java Connection

Importing Libraries. In **Java**, you often have to import **classes** that are extensions to the base language. **jQuery** is a bit like that. It also has to be explicitly included in your program.

At the time of writing, version 1.10.0 was the latest version of jQuery, but that might have changed by now. The latest version is always freely available at the **jQuery website [http://jquery.com/]**, where you can also choose the "minimized" version of the library to save bandwidth (it's the same as the regular library, but with whitespace and other unnecessary characters removed, **variable** names shortened and so on).

Do It Yourself

To use the sandbox, load **jQuerySandbox/index.html** from the **example pack** into a browser and open the JavaScript console. Because the `index.html` file loads the **jQuery** library, you can play with **jQuery [http://jquery.com/]** commands in this window.

You'll notice there are some instructions for activities on the page. You can try them now if you want, but you might do better to wait for a bit more information.

11.1.2 The `jQuery` ($) Function

Everything you will do with the jQuery library will be based on the new `jQuery` **function**. There are two ways to write this function, shown below:

1. `jQuery( selector )` or

2. `$( selector )`

From now on, I will just use $ version.

The **selector** part of the **syntax** works just like a CSS selector, with a few extensions unique to **jQuery selectors**. The jQuery function will return an **object** of type `jQuery` that looks and behaves very much like an **array** of **DOM** elements.

Before moving on, you may need to review the **w3Schools CSS selector reference** and you should also take a look at the jQuery **API** selector **reference** as well. Note that some CSS selectors are not available in jQuery (e.g. `:hover`) and some jQuery selectors are new (e.g. `:not()`). Why would `:hover` make sense for CSS but not for jQuery?

Do It Yourself

Open the JavaScript console for this textbook and try this command:

```
$("h1")
```

The jQuery object that is returned to you should look something like this:

```
[<h1>Contents</h1>,<h1>1. Getting Started</h1> ... ]
```

If there are more than 100 elements in the array, things might look a little different. For instance if you type:

```
$("a")
```

you'll get something like this instead:

```
> st.fn.st.init[432]
```

Which means there were 432 **objects** returned. If you open the result by clicking on the arrow, you will be able to see the contents, arranged into groups of 100.

Do It Yourself

Try the following in a JavaScript console for the **jQuerySandbox/index.html** page and make sure the results are correct. Remember that in Chrome you can mouse over the resulting objects to see the corresponding DOM elements highlighted on the original page.

1. Get all `<td>` elements with the `class` attribute set to "odd"
2. Get all `<h1>` elements
3. Get all `<span>` elements
4. Get the element with the `id` set to "instructions" as well as every `<tr>` element.
5. Execute the jQuery command `$("*")`. What elements are in the resulting **jQuery object [http://learn.jquery.com/using-jquery-core/jquery-object/]**?
6. Execute the **jQuery** command `$(document)` with no quotation marks inside the brackets. What element is in the resulting **jQuery [http://jquery.com/] object**?

The object returned from the $ function can be treated a lot like any an array. For example, this command will change the border style of the checkerboard:

```
$("#checkerboard")[0].style["borderStyle"] = "dashed";
```

The above statement is equivalent to the following:

```
document.getElementById("checkerboard").style["borderStyle"] = "dashed";
```

Similarly, you could put an X into every even checkerboard square of **jQuerySandbox/index.html** with the following:

```
var squares=$("td.even");
for (var i=0; i<squares.length; i++)
    squares[i].innerHTML = "X";
```

But wait! Although in some rare cases you might want to do array processing on the **jQuery object [http://learn.jquery.com/using-jquery-core/jquery-object/]**, in most cases there is a far easier way. Read on...

11.1.3 Chaining Methods: `innerHTML`, styles and attributes

So now you've mastered selectors, and you can process the jQuery object as if it were an array. But where jQuery really gets its power is in the **chaining** of other jQuery **methods** onto the end of the $ function. Four such methods are `html`, `css`, `val` and `attr`. When chained at the end of the $ function using the dot operator, these functions can be used to set the `innerHTML`, style properties, `value` attributes (for form elements) and other attributes of an entire set of selected elements all at once.

Do It Yourself

Here are some examples to try in the JavaScript console of **jQuerySandbox/index.html**:

```
$("#instructions").html("No more instructions!");
$("td.odd").css("backgroundColor","blue");
$("a").attr("href", "http://www.facebook.com");
$("input:text").val("A new value");
```

Verify the changes made to the page in each case. Note that the last one makes use of a special **jQuery selector** `input:text` which selects all input **elements** of type text. There are also selectors for `:button`, `:checkbox` and so on.

You can also use each of these methods with one less parameter, and they will return the innerHTML, style property, or attribute of the first element in the matching set. Try the following:

```
$("#instructions").html();
$("td.odd").css("backgroundColor");
$("a").attr("href");
$("input:text").val();
```

Verify that the results make sense in each case.

And method chaining doesn't stop at just one method! Each chained jQuery method returns a copy of the original jQuery object. This means methods can be chained one after another as many times as you want.

Do It Yourself

Try the following command in **jQuerySandbox/index.html**, but try to predict the result before you press enter.

```
$("#instructions").css("backgroundColor","red").css("color","rgba(255,255,2
55,0.5)").html("Hello, World!");
```

There is no limit to how many **methods** you can chain together in this way.

You can also change the class(es) an element belongs to with the `addClass` and `removeClass` methods, and you can find out if it belongs to a particular **class** with the `hasClass` method. More information on these methods can be found in the **w3schools** jQuery section called **jQuery CSS Classes**.

The jQuery methods mentioned above all have counterparts in ordinary JavaScript (`style`, `innerHTML`, `className`, etc.). But in many cases the jQuery versions are more powerful.

Do It Yourself

For example, try the JavaScript statement below in **jQuerySandbox/index.html**.

```
document.getElementById("instructions").style["color"]
```

It returns an empty string because the color property of this element is not explicitly set in the style attribute of the element.

Now try the equivalent jQuery statement.

```
$("#instructions").css("color")
```

This command searches the hierarchy of CSS styles and returns the correct color `"rgb(0,0,0)"` which it retrieves from the default browser style.

<exercises section='11.1.3'>

1. Try the following on **jQuerySandbox/index.html** using the JavaScript console of your browser. Note that you can always reload the page to get it back to its initial state between commands.
   a. Use the `.css()` **method** to change the checkerboard border style to dashed.
   b. Use the `html()` method to put an "X" in every even checkerboard square.
   c. Use the `addClass()` and `removeClass()` methods to change the odd squares to even.
   d. Use the `html()` method to change every link on the page to read "this is a link".
   e. Change the contents and color of all `<td>` elements in one line.
   f. Change the class of even to odd and odd to even in two lines. Can you do it in one line?
2. The following require changes to the **jQuerySandbox/index.html** file of the sandbox. Make sure you also have the CSS and JavaScript files. You can view the source of the page and open the extra files from there, or you can get everything in the **example pack**.
   a. Put a button on the sandbox page that, when clicked, toggles the checkerboard from white and red squares to brown and black squares, and back again.
   b. Put a second button on the sandbox page that prompts the user for two numbers, then moves the instructions panel to those coordinates.
3. Get the **basicForm.html** file from the example pack and make the following changes:
   a. Load the **jQuery** library in a `<script>` element.
   b. Using **jQuery [http://jquery.com/]**, add an `onsubmit` **event** handler to the form, and check to make sure the user name and password **fields** are filled in, that the password is at least 8 characters long and contains letters and numbers, and that at least two interests are selected. If not, return `false` and issue an alert telling the user what went wrong.
   c. Add a "fill in" button that auto-fills the form with **values** of your choosing. Use jQuery for all processing.

</exercises>

11.1.4 Effects and Animations

The jQuery library also contains methods to make various effects and animations easy. These methods can be chained after the $ function just like the ones discussed in the previous section.

Do It Yourself

Here's an example to try in **jQuerySandbox/index.html**:

```
$("#instructions").fadeOut();
```

The above will fade out the instruction panel, and then remove it by setting its display **property** to "none".

Note that to be at its most effective, the `fadeOut()` method should only be applied to elements with absolute positioning. Try the following to see what happens otherwise:

```
$("#td.odd").fadeOut();
```

I will leave it to you to explore most of these effects on your own. You can find information on all of them in the **jQuery API [http://api.jquery.com/]**. Look in the "effects" section. Read the "basics", "fading" and "sliding" sections. Click on each method name for an explanation. You can also go to the **w3schools Learn jQuery** section for more help.

Note that most of these methods have optional parameters. For example, the page for the `fadeOut()` method has the following header:

```
.fadeOut( [duration] [, complete ] )
```

The square brackets mean that the `duration` and `complete` parameters are optional. If you call this function with no parameters, it will use default **values**. You can also set the duration if you call it with one parameter, or set both duration and "complete" if you call it with two parameters. (I will explain more about the "complete" parameter in the next section).

There is also a generic `animate()` method which is explained under "custom" in the "effects" section of the **jQuery API [http://api.jquery.com/]**. This method needs at least one parameter, specifying a set of CSS properties and values to "move towards". Any of the specified properties that can be animated (usually numeric properties) will be animated.

Do It Yourself

Try this in **jQuerySandbox/index.html**:

```
$("#instructions").animate({left:"200px", width:"100px"})
```

The parameter on the above instruction is a little odd if you've never seen anything like before. The curly brackets indicate that it is actually a specification of a JavaScript object - it's an object **literal**, something that is unique to languages which, like JavaScript, were derived from **ECMAScript**. The contents of the curly brackets consist of a list of **field** names (unquoted) and values (quoted).

So `{left:"0px", width:"20%"}` creates a new object with the fields `left` and `width` set to contain the **strings** `"200px"` and `"100px"` respectively. Then this object is passed as a parameter to `animate()`. You will learn a lot more about creating JavaScript objects in a later chapter.

<exercises section='11.1.4'>

1. Complete the **jQuerySandbox/index.html** exercises as laid out in the instructions panel.
2. Explore the **jQuery API** a bit more. Take a look at the "Attributes", "CSS", "Dimensions", "Effects" and "Manipulation" sections. Find and try out at least 3 more interesting **methods**.
3. Go back to the memo pad exercise from previous chapters and tweak it so it makes effective use of **jQuery [http://jquery.com/]**.
4. Go back to the exercise based on **dropdownMobileExercise.html** in which you created menus that appear and disappear when buttons were clicked. Modify this to make good use of **jQuery selectors** and method **chaining**.
5. Go back to the **catchTheRabbit.html** exercise from previous chapters. Using jQuery, change your original solution so that when you mouse over the rabbit, it slides to a new, randomly-generated location on the screen. (Tip: use the **JavaScript window object** to find out the height and width of the screen.)
6. Take the **slidingTiles.html** code from the **example pack** and add JavaScript (using jQuery) to implement a sliding tiles puzzle. When the user clicks a tile (i.e. an `<a>` **element**) that is beside the blank tile, it should slide smoothly into its new position using the jQuery `animate()` method. Note that this code can be very tricky to write if you have never done anything like this before.

</exercises>

11.2 jQuery Events and the Truth about Functions

Up until now, you have probably been adding **event** handlers to **elements** by placing **JavaScript** code into the corresponding HTML **attributes**, like this:

```
<input type="button" onclick="myFunction()">
```

This is not considered best practice by most JavaScript programmers for at least three reasons.

1. It spreads your JavaScript code throughout the HTML file, making the code harder to maintain. (i.e. It would be much better if you could add all your event handler code in a single place.)
2. It violates the clean separation between the structure of the interface (the job of HTML), the style of the interface (the job of CSS) and the functionality of your app (the job of JavaScript).
3. It causes the browser to create a new **anonymous function** with the code you put in the attribute as its contents. This is unnecessary and wasteful. (I'll explain what I mean by this in a moment.)

It is widely considered best practice to add event handlers in JavaScript after the document has loaded. But to be able to do that, you need to understand how functions work in JavaScript in a little more depth. Once you have this understanding, you will also be able to use the new **jQuery** `ready` **method**, as well as the **callback** parameters of the **jQuery [http://jquery.com/]** effects **functions**.

11.2.1 The Truth About Functions

The truth about JavaScript functions is that they are actually **values**, just like **strings**, numbers, **Booleans**, and **objects**. Like any other value, functions can be stored in **variables**, passed as parameters to other functions, and returned from function calls.

The technical term for this is that JavaScript functions are **first-class functions**.

When you use a function declaration like this:

```
function foo (a, b, c) {
```

```
    alert(a+b+c);
}
```

What you are really doing is creating a variable named foo, and assigning a function value to it.

The following shows the assignment of a function literal to a variable. It is legal JavaScript code, and is basically equivalent to the function declaration above:

```
var foo = function(a, b, c) {
    alert(a+b+c);
};
```

Java Connection

Methods vs. Functions. In **Java**, **methods** and **variables** are completely different entities (how would you describe the difference?) In **JavaScript**, function names are actually variable names, and functions are data.

The above **statement** creates a variable named `foo`, and assigns to it an anonymous function with 3 parameters. Note that since this is an assignment statement, we put a semicolon at the end of it.

The main difference between the two notations is that you can only use function **declaration** in certain contexts, but you can use variable declaration and **function literals** in any context. Because it is limited and adds nothing to the language, many programmers avoid function declaration altogether and always use the variable declaration / function **literal** notation.

Do It Yourself

No matter whether you use function declaration or variable declaration with function literals, if you refer to foo in your program, you are referring to a variable. To see this, try the following in the JavaScript console of your browser:

```
function foo () {alert("hi");} ← foo is a function
foo;
foo();
```

Question: What is the difference between the last two lines above?

```
alert(foo);
alert(foo());
```

Question: What is the difference between the above two lines?

Try the lines below to see that you can switch `foo` back and forth between a function and other data types.

```
foo = 5; ← foo is now a number
alert(foo);

foo = function() {alert("hi");}; ← foo is a function again
alert(foo);

foo = "hi"; ← foo is now a string
alert (foo);
```

Answers: In the console window, `foo` returns the value of the variable. It's a function. But `foo()` calls the function so you will see the popup. On the other hand, `alert(foo)` will display a popup whose content is the value of the variable `foo` (if it's a function, it will show you something to that effect). Finally, `alert(foo())` will first call the `foo` function, displaying a popup. Then it will create another popup to display the **return value** of `foo`. Since there is nothing explicitly returned, it will return the special value `undefined`.

11.2.2 Adding Event Handlers in Plain JavaScript

If you want to add event handlers to an element after it is created, you can do it by getting the element out of the **DOM**, creating a function, and assigning that function to the attribute for the event handler.

Do It Yourself

Try the following in the javaScript console for this textbook.

```
document.getElementById("test6").onclick = function(event) {
    alert("hello!");
};
```

Since this DIY box has an id of `test6`, you should now be able to click it and see the result.

P.S. Don't worry about the `event` parameter above. It will be explained at the end of this section.

Or you can create the function (either through function declaration or using a function literal) first, and then assign it to the event handler.

Do It Yourself

Try the following in the JavaScript console for this textbook.

```
var clickHandler = function (event) {
    alert("goodbye! ");
}
document.getElementById("test7").onclick = clickHandler;
```

Now you should be able to click this box for a message.

Another option that some programmers prefer is to use the `addEventListener` method which takes two parameters. The first is the event name (without the "on" -- so use "click" instead of "onclick") and the second is the handler function.

Do It Yourself

Try this in the JavaScript console for this textbook. Note that you will have to define clickhandler first. You can get it from the last DIY box.

```
function clickHandler (event) {
    alert("goodbye! ");
}
document.getElementById("test8")
    .addEventListener("click",clickHandler);
```

Of course you can also add a handler using a function literal. Try this:

```
document.getElementById("test8")
    .addEventListener("click", function(event) {
        alert("hello!");
    });
```

Now, when you click this box you should get two popups, not one. This illustrates one crucial difference between assigning to `onclick` and using `addEventListener`. The latter allows you to add many **event** handlers of the same type to the same **object**.

You should also notice that in the second case above, the **function literal** is not assigned to any particular **variable** name. This makes it an **anonymous function**.

Java Connection

Event Listeners. Java and **JavaScript** both use `addListener` **methods** to register **event** handlers. The difference is that in Java an **event listener** is an **object**, while in JavaScript it's a **function**.

Note that the functions above all contain an `event` parameter. Any event handler is always passed a special event object that represents the event that fired to trigger this function. You can use it to access information about the event, such as which element was the target (`event.target`), which mouse button was pressed (`event.button`), the location of the mouse (`event.clientX` and `event.clientY`), etc.

Do It Yourself

Here are two last things to try in the JavaScript console for this textbook.

```
document.getElementById("test9").onclick = function(event) {
    alert(event.pageX+","+event.pageY);
};
```

Now when you click on this box, you will get to see the exact X and Y location of your click, in pixels from the top left corner of the page.

The **event object** contains a number of useful **fields**, but probably the most useful is the `target` field which contains the DOM node that was the subject of the event.

Try this:

```
document.getElementById("test9").onclick = function(event) {
    alert(event.target.innerHTML);
};
```

Now clicking this box should get you the full `innerHTML`.

For more information on the JavaScript Event object, go to w3schools and click on **HTML DOM**, then **HTML DOM Objects**, then **DOM Events**.

<exercises section='11.2.2'>

1. Load **eventHandlerTest.html** from the **example pack**. Compare how the handlers have been added to the **elements** with `id` `test1` and `test2`.
    a. Consider the page as it is loading into the browser. At what point in time is each handler added? To check your answer you will probably have to look up the **JavaScript window object** and onload **event** on **w3schools**.
    b. Try commenting out the `window.onload` line and the `};` line that goes with it and reload the page. Describe how the page works (or not) now and explain why.
2. Add two handler **functions** to the element with `id="message"` using JavaScript (not HTML **attributes**). These functions should respond to the `onmouseover` and `onmouseout` events. When the mouse is inside the message box, one of the other boxes should be pink and the other should be white. When the mouse is outside the message box, the one that was white should become light blue and the one that was pink should become white.

</exercises>

11.2.3 Event Handlers in jQuery

The jQuery library contains its own event management system that extends the regular HTML and JavaScript system, making it more robust, useful and cross-browser compatible. In jQuery, event handlers are added to elements by chaining an event method on the jQuery selector function, like this:

```
$( selector ). event ( handler );
```

In the above, "selector" is a normal **jQuery selector**, "event" is an event name (`load`, `click`, `mousenter`, etc.), and "handler" is a function.

Do It Yourself

Here's an example of how to add a `clickHandler` function to the all DIY boxes in this document. Try it in the JavaScript console.

```
var clickHandler = function (event) {
    alert(event.pageX+","+event.pageY);
}
$("div.DIY").click(clickHandler);
```

The `event` **object** above is actually a new **jQuery event** object. It works exactly like the original **JavaScript** object, but with full **cross-browser compatibility**.

The correct place to add handlers is either in the `load` event of the `window` object, or in the new jQuery `ready` event of the `document` object. Here's what this would look like.

Using the `load` event of the `window`:

```
$(window).load( function() {
    $( selector ). event ( clickHandler );
    // more code, function declarations, etc.
});
```

Using the `ready` event on the `document`:

```
$(document).ready( function() {
    $( selector ). event ( clickHandler );
    // more code, function declarations, etc.
```

```
});
```

(In the above examples, you would replace "selector", "event", and "clickHandler" with appropriate values.)

The difference between the above two options is that the `window` `onload` event will not fire until all resources (images, etc) of the are loaded. But the jQuery `ready` event will fire as soon as the DOM is constructed. So unless you want to access images and other external resources that may not be loaded yet, the `ready` event is the one you should use.

We should also say something here about `$(window)` and `$(document)`. These calls to the jQuery function do not use a CSS-style selector. Instead they simply provide an object as a parameter. The jQuery function supports this by return a **jQuery object [http://learn.jquery.com/using-jquery-core/jquery-object/]** with a single object in it - the one that was passed in. So if you have an object and you want to chain jQuery methods on it, you can use the jQuery function and just pass it the object.

Do It Yourself

Try the following in the JavaScript console for this textbook.

```
$(document.getElementById("test4"));
```

and

```
var x = document.getElementById("test4");
$(x);
```

and

```
$("#test4");
```

You should get the same result in each case. We will see another very important use of this soon.

Go to the jQuery **API** events **reference** for more information on the events supported by jQuery. The most important categories of events are keyboard, mouse, and form events, but you should take a look at the others as well. Note that jQuery provides cross-browser support for some events and event utilities you may not have seen before (e.g. Internet Explorer's `mouseenter` event).

<exercises section='11.2.3'>

1. Load the file **eventHandlerTest2.html** from the **example pack** into a browser. Take a look at how the handler has been added to the **table** cell **elements**. Change the file so that the handler is added on the load **event** of the window.
2. Add a `<textarea>` element to the HTML. Using **jQuery**, add a handler that responds to the `keyUp` event, checks the element's **value attribute** for certain "secret words" and changes the colors of other elements on the screen in response.

</exercises>

11.2.4 A Note on Best Practices

It is actually considered best practice in JavaScript, whether you're using jQuery or not, to write **all** of your code inside a `window` load event or `document` ready event. Even other helper functions should be defined inside these events, like this:

```
$(document).ready(function() {
    ALL YOUR CODE GOES HERE
});
```

or like this:

```
$(window).load(function() {
    ALL YOUR CODE GOES HERE
});
```

or if you're not using jQuery, like this:

```
window.onload = function() {
    ALL YOUR CODE GOES HERE
});
```

When you do this, the browser will wait until the page is ready before executing your code. So you can put your code into the `<head>` of your document without having to worry about whether the DOM is ready at that point. This is a big advantage. Another advantage is that now all variables you use (including function names) become **local** to the anonymous function you assign to the `ready` or `load` event. This is good too because it minimizes potential name conflicts with other JavaScript or jQuery code or libraries you might be using.

11.2.5 jQuery Callback Functions

Many jQuery effects have parameters for callback functions. Now that you understand a bit more about functions, you can use these effectively as well. For example, the `slideDown()` method has parameters called `duration` (in milliseconds) and `complete`. The `complete` parameter should be a function that is to be called after the slide operation has finished. A function used in this way is often referred to as a callback function.

Do It Yourself

Load up the file **callBackTest.html** from the **example pack** and slide up all the `div` elements, like this:

```
$("div").slideUp();
```

Now try the following in the JavaScript console to slide them down and turn them red when the slide is finished:

```
$("div").slideDown(1000, function() {
    $(this).css("backgroundColor","red");
});
```

Java Connection

This (keyword). In **Java**, the `this` keyword inside a **method** always refers to the **object** that the method is linked to. The `this` keyword in **JavaScript** does the same thing.

Notice the use of the `this` keyword above. In the example shown, the callback function is going to get called for every single `div` element on the page. It's job is to turn each one red. So each time the function runs, it needs to know which object it is being run against - that's what the `this` keyword is for - it's a variable that has been automatically loaded with the object the function is being applied to. As I mentioned earlier, you can use the jQuery function to wrap any DOM element inside a jQuery object. So the `$(this)` selector lets you chain jQuery methods against whichever element the function is being called against.

Do It Yourself

You might be thinking that we don't need the callback function. Why not just do the following in **callBackTest.html** (reload the page before trying it):

```
$("div").slideDown(1000).css("backgroundColor","red");
```

Question: What went wrong here?
Answer: The call to the `slideDown` **method** only starts the slide. Once the slide is started, it lets the `css` **function** run. Since this happens very quickly, it loos like the `css` function goes first, followed by the `slideDown`.

One more thing to notice about **callBackTest.html** is that it is following the best practice of defining all code in a `$(document).ready` event handler.

<exercises section='11.2.5'>

1. Add to **callBackTest.html** so that when the user mouses over `div2`, it slides downwards 100 pixels and then changes color and fades out.
2. Add to **callBackTest.html** so that there is a third `<div>`. When it is clicked, it should start sliding around the screen randomly and not stop until it is caught. (This is a bit tricky.)
3. **JQuery** also contains a cool `toggle` **method**. This method takes two **functions** as parameters and then calls one or the other each time the toggle function is used. So you can go back to exercise 2a from 11.1.3 and use this toggle method to switch the colors back and forth on each click.

</exercises>

11.3 DOM Manipulation with jQuery

As with most things in JavaScript, **jQuery** can make **DOM** manipulation a lot easier. If you go to the **jQuery [http://jquery.com/] API** and look into the "Manipulation" section. There you will find a multitude of **methods** like the following:

`.html()`	use this to change the innerHTML property of the selected elements
`.append()`	appends a DOM Node or an HTML string after the selected elements
`.before()`	inserts a sibling DOM Node or an HTML string before the selected elements
`.after()`	inserts a sibling DOM Node aor an HTML string fter the selected elements
`.remove()`	removes the selected elements from the DOM

Remember that you can use the `this` keyword to refer to each selected **element**, and then the selector `$(this)` to chain jQuery methods.

Do It Yourself

Try the following in the JavaScript console for this textbook:

```
$("div.DIY").click(function(event) {
    $(this).remove();
});
```

Now try clicking on any DIY box and watch it get removed. Notice that no matter where within the DIY box you click, the whole box is removed.

Do It Yourself

To truly appreciate what jQuery is doing for you in this instance, try this roughly equivalent raw JavaScript code in the console for this textbook.

```
var diys = document.querySelectorAll("div.DIY");
for (var i=0; i<diys.length; i++)
    diys[i].onclick = function(event) {
        event.target.parentNode.removeChild(event.target);
    };
```

Now click on a sentence or a line of code within a DIY box. Notice that only part of the DIY box content disappears.

In raw **JavaScript** the target is the lowest level **DOM node** that was clicked on (i.e. the <p> **element** within the). But **jQuery** makes sure the target you were interested in (i.e. the one that matches the selector) is the one removed.

<exercises section='11.3'>

1. In **morePractice.html** from the **example pack** you'll find HTML to define a little **table** and four buttons. Add **events** using **jQuery** to these buttons so they delete and add rows and columns from the table. Careful, it's a bit trickier than it sounds!

</exercises>

12. AJAX with jQuery

*The text in this chapter mirrors the text in Chapter 10. In this chapter, you learn to use **AJAX** with the **jQuery** library and you also learn a little bit about how to work with **XML** data. In Chapter 10, you learn to use AJAX with raw **JavaScript** and without XML.*

Even if you don't know what AJAX is, you have experienced its effects. When you start typing into the **Google [http://www.google.com]** search box, it doesn't take long before you start getting some suggestions about what to search for. When you view your list of friends on **Facebook [http://www.facebook.com]**, it starts you with a small list, and when you scroll to the bottom of that list, it loads some more for you. Both of these are examples of AJAX in action.

Before AJAX, if you wanted to get some more content for the user, you had to get the browser to load and display a whole new page. With AJAX, you can launch an **HTTP** request from JavaScript. Then when the response data comes back from the server, you can use JavaScript code to incorporate this data into the page without reloading it.

12.1 The Basic Idea

The term **AJAX** was coined by a web developer named **Jesse James Garrett [http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications]** in 2005. At the time, AJAX was a term for a new way of thinking about and making use of a collection of already existing technologies. AJAX stands for **Asynchronous JavaScript and XML**. Let's break this phrase down.

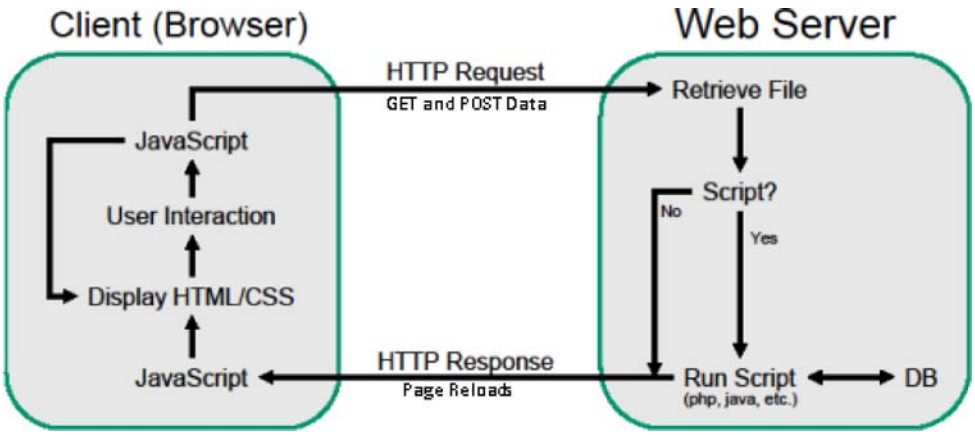
JavaScript: You already know what this is.

Asynchronous: You already know about this too, even if you don't realize it. **Asynchronous** processes are processes that happen outside of the regular flow of the program. In Asynchronous programming, you make some kind of request that will take time to complete, and then continue on with other tasks. In many cases you also specify a **callback function** to be triggered when the asynchronous task is finished. You saw this in the last chapter where I introduced you to jQuery's animation and effects functions like `slideUp`, `fadeIn`, and `animate`. Because these are asynchronous tasks, the user can still interact with the web page while the animation is happening. If they were not asynchronous, everything would be at a standstill until they finished.

XML: You may have encountered this before. **XML** stands for eXtensible **Markup Language**. It's a generic markup language for representing structured data. It looks a lot like HTML, but it can be used to represent any kind of data (sports scores, the weather, a shopping list, etc.). XML is one possible format for the data that comes from the server in response to an AJAX request. But you can get your data from the server in any format you want. It can be XML, HTML, **JSON** (**JavaScript Object Notation**) or even just plain text. No matter what kind of data you're getting, most developers would say that if you're getting it asynchronously by launching a request from a **JavaScript** program, you're using AJAX.

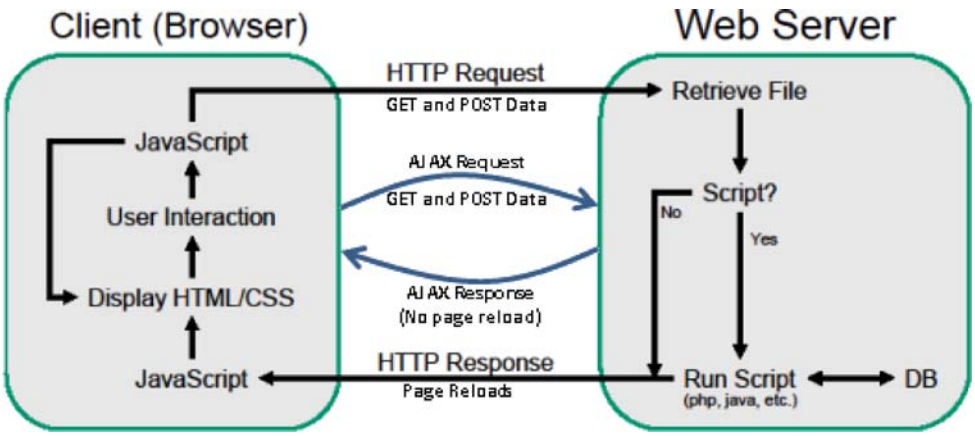
12.2 A New Web App Architecture

Up until now, we have been assuming a **web app** architecture that looks like this.



In this architecture, an **HTTP** request is launched by the user either by typing a URL, clicking a link, or submitting a form. (HTTP Requests are also generated by the browser to automatically to load images, CSS files, **JavaScript** files and other resources. But I'm ignoring that minor complication in the story I'm telling here.) The request goes to the server where either a file is retrieved and becomes the HTTP Response, or a script is run and the output of the script becomes the HTTP Response. When the response arrives at the browser, it interprets the HTML and CSS in the document to display the page, and executes the JavaScript commands in the page. Then it continues to run JavaScript in response to user interactions and other **events**, refreshing the page view when something changes. It doesn't go back to the server again unless the user clicks a link, types in a URL, or submits a form.

But with **AJAX**, the life cycle looks a bit different:



An AJAX request is an HTTP Request just like any other, and can have **GET and POST** parameters associated with it just like any other. At the server it is treated in exactly the same way as any other HTTP request. The only difference is where it came from: it was launched by a JavaScript **function**. Since HTTP Requests can take time to complete they are done asynchronously and you register a **callback** function to deal with the response when it comes. (What would happen if this was not **asynchronous**?)

The HTTP Response from the server is also the same as any other. The only difference is that when the response arrives at the client, it will not trigger a page reload. Instead it will be passed to the callback function you registered.

12.3 How jQuery Makes AJAX Easier

AJAX queries are launched using the **XMLHttpRequest object** that is built in to all modern browsers. To launch an AJAX request from scratch, you would create a new XMLHttpRequest object and build it up over several lines of code, adding headers, **GET and POST** parameters, registering a **callback function**, etc. It is not that hard to do (for more information, go to the **w3schools Learn AJAX** section) but it's messy and easy to make mistakes. As with so many other tasks in **JavaScript**, **jQuery** will make your life a lot easier here.

12.3.1 A Data Source

You can't do much in AJAX without access to a data source on a server somewhere. There are many excellent sources of data on the web that are free to use in your AJAX programs. For example Yahoo's **YQL [http://developer.yahoo.com/yql/]** (**Yahoo Query Language [http://developer.yahoo.com/yql/]**) can be used to retrieve information about geographic locations, weather, music, maps, and a number of other domains. But for now, you need an easier-to-use source of information.

> ***Do It Yourself***
>
> In the **example pack**, you will find a **servers** folder containing two PHP scripts: a "time server" named **servers/timeServer.php** and a "Lorem Ipsum server" named **servers/loremIpsumServer.php**.
>
> Load the two server files by clicking on the links above (don't try to open them on your own machine). Then read over the **APIs** they display and try them out. You can experiment with the GET parameters by typing them into a URL. For example, if you wanted to set the `pstart` parameter to 2 and `plength` to 1, you could do it like this:
>
> ```
> servers/loremIpsumServer.php?pstart=2&plength=1
> ```
>
> And if you want to test POST parameters with the Lorem Ipsum server, you can use the **servers/loremIpsumPostTest.html** file which uses a `<form>` **element** with its `method` **attribute** set to "post" to send requests to the Lorum Ipsum server.
>
> If you've forgotten what all this means, you may need to review **w3schools** or a similar source to refresh your memory about **GET and POST** and **PHP Form Handling**.

> ***Do It Yourself***
>
> The Lorem Ipsum and Time servers will be used as a test bed for the AJAX examples and exercises to come. But PHP files will not run outside a web server, and AJAX requests that do not come from the same web server will usually be rejected due to the Same Origin Policy. So you will need to nail down one of the two following web server options to do any further work in this chapter.
>
> 1. Install WampServer or XAmpp on your machine to turn it into a web server, or
> 2. Find some web space (e.g. mobile.sheridanc.on.ca) and upload your programs to it for testing.
>
> If take the first option and you are using **NetBeans [https://netbeans.org/]**, you should be able to use a project of type PHP and set up an option so that NetBeans automatically copies files to the WampServer or **XAmpp [http://www.apachefriends.org/en/xampp-windows.html]** server running on your machine. If you get that right, you can "run" your PHP projects from **NetBeans [http://www.netbeans.org]**.
>
> Take a moment to set yourself up before moving on.

12.3.2 Your First AJAX Request

The jQuery **API** lists a number of shortcut and helper functions to make your life easier using AJAX. We will **focus** on just two here, both listed under the heading "shorthand **methods**" in the AJAX portion of the API. The two methods are `jQuery.get()` and `jQuery.post()`. The name `jQuery` is just another way to access the familiar `$` object, so we will use `$.get()` and `$.post()` in the examples below.

As you might have guessed, these functions are for sending GET and POST requests, respectively. Other than their name, the **syntax** of these two functions is identical. There are three parameters of note:

1. **URL:** A string parameter that encodes the base URL of the request. This can be an absolute URL or it can be relative to your current location.

2. **Data:** This is a map of your GET or POST parameters.

3. **CallBack:** This is a function that will be called when the **HTTP** Response arrives.

Do It Yourself

Load **ajaxTime.html** from the **example pack** into a desktop browser (if you are working with your own copy, remember that you need to have a PHP server running or it won't work - see the previous section).

Ignore the contents of the page for now. Just open the JavaScript console and launch your first AJAX request, like this:

```
$.get(
    "servers/timeServer.php",
    {
        timezone: "Canada/Eastern",
        request: "time"
    },
    function(data) {
        alert(data);
    }
);
```

The above should pop up the current time for you in an alert box.

There are three parameters in the `$.get()` method above. The first is the URL. Note that it is specified relative to whereever the browser is now. The second is a map of GET parameters (this object has a syntax exactly like the map you had to pass to the `animate()` method in the previous chapter). Together, the URL and data parameters above are equivalent to the following:

```
… /servers/timeServer.php?timeZone=Canada/Eastern&request=time
```

The final parameter is the callback function. This function could take up to 3 parameters, but we will only be using the first (`data`), which contains the text of the response from the server.

Do It Yourself

Repeat the command from the last DIY box, but this time use `$.post()` instead of `$.get()`.

Now try changing some of the parameters to get the day or month, or use a different time zone.

Now try changing the **callback function** so it puts the text somewhere on the screen (maybe in the `innerHTML` of all `<h1>` **elements**, or in the **value attribute** of all `<input>` elements).

Finally, try out the form interface provided on the **ajaxTime.html** page. Do you think you could write the code for this page now, given what you know about **AJAX** and **jQuery**?

12.3.3 AJAX From a Web Page

The JavaScript code below is from the **ajaxTime.html** example. Take a moment to read it over and try to understand it.

```
$(document).ready( function() {
    var launchAjax = function() {
        $.get(
            "servers/timeServer.php",
            {
                timezone: $("[name=timezone]").val(),
                request: $("[name=type]").val(),
                delay: $("[name=delay]").val(),
            },
            function(data, textStatus, jqXHR) {
                $("#result").html(data);
            };
        )};
    $("#ajaxButton").click(launchAjax);

});
```

The first thing to note is that all the code is inside the `document` object's `ready` **event** handler. This is just following the best practices that were introduced in the last chapter.

The second thing to notice is that the `launchAjax` function contains a call to `$.get()` that should look very familiar from the previous section. The only difference is that one more parameter is set (`delay`) and the **values** for each parameter are coming from the form **elements**. Also, the callback function is placing the data into the `innerHTML` of the element with an `id` of "result".

The syntax `$("[attribute=value]")` is used to select all the elements with a particular **attribute** value. In this case it's being used to retrieve named form elements. The programmer could just as easily give them `id` attributes and use `$("#id")`.

After the definition of `ajaxFunction`, the function is added as a `click` event handler onto the element with an `id` of "ajaxButton". This could have been done in HTML, but again, it is considered best practice to do it here.

Now you have two data sources (**servers/loremIpsumServer.php** and **servers/timeServer.php**) and a worked example (**ajaxTime.html**). It's time to build something of your own.

<exercises section='12.3.3'>

1. Take the file **servers/loremIpsumPostTest.html** from the **example pack** and change it so that instead of launching a brand new POST request and reloading the page, it instead launches an **AJAX** query and displays the result below the form.

2. In the exercises folder, look at the file **findLorem.html**. You have the HTML and CSS here for a game. You just have to fill in the **JavaScript**. This game will pull out a **random** piece of Lorem Ipsum text from the server, using an AJAX query. Then it will allow the user to guess where the text came from, using more AJAX queries to check whether they were right or not. See the comments in the **findLorem.html** file for detailed instructions.

3. In the example pack you will find **more.html** and its associated files. Add the necessary JavaScript code so that this page allows the user to load one paragraph at a time from **servers/loremIpsumServer.php**. When they first load, they should see the first paragraph, retrieved with AJAX. Then whenever they hit the "more" button, they should get the next paragraph in sequence.

</exercises>

12.4 What To Do While You Wait

Because networks can be slow, **AJAX** requests can sometimes take a few seconds to complete. If things don't happen instantly, it can cause the user to start pressing buttons over and over again, or behaving in other destructive ways which could end up launching more and more AJAX requests. This is a problem you have to think about whenever you use AJAX.

> ***Do It Yourself***
>
> Load up **ajaxTime.html** from the **example pack**, set the delay on the form to 15 seconds, and set the request to "time", then hit the "**AJAX**!" Button. This will send a request to the **servers/timeServer.php** page which instructs the php scripts to pause for 15 seconds before completing. So this simulates a delay over a slow network.
>
> While you're waiting for your first request, set the delay back to 0 and try some different requests (anything other than "time"). After 15 seconds, the original request will get its response and this will disrupt what you are currently doing.

12.4.1 Some Options

The usual solution to this problem is to control the user somehow while you are waiting for the AJAX request to complete. For example, if the user clicks something to load more content, you could do the following:

1. In the click handler, disable the button the user just clicked, and perhaps give them some visual indication that we are waiting for something to load.
2. In the **callback function**, re-enable the button.

This will ensure that only one AJAX request is dealt with at a time.

Another approach might be to have a **global variable** with a name like `waitingForAJAX`:

1. Initialize `waitingForAJAX` to false.
2. In the click handler, check if `waitingForAJAX` is true. If it is, give the user an error message. If not, launch the AJAX query and set `waitingForAJAX` to `true`.
3. In the callback function, set `waitingForAJAX` back to to false.

<exercises section='12.4.1'>

1. Add code to the **ajaxTime.html** page so that the user has to wait until the previous request has completed before a new one will be launched. You can use either of the two strategies above. Then test it by setting the `delay` parameter.
2. Repeat the above, but this time use the other suggested strategy for what to do while you wait.
3. Alter your solution to the **more.html** exercise from the previous section. Set up the **AJAX** requests to include a short delay and make sure you disable the user while they wait for their next paragraph.

</exercises>

12.4.2 AJAX Error Handling

The only problem with the above strategy is that sometimes AJAX requests do not complete (the server is down, there's a typo in the URL, etc.). In this case, the callback function is never called, and the user could end up waiting forever.

In the AJAX section of the jQuery **API**, there's a subsection on **Global AJAX Event Handlers [http://api.jquery.com/category/ajax/global-ajax-event-handlers]**. This section describes the `$(document).ajaxError()` **method**. You can use this function to register an error **event** handler attached to any **element** or elements. Then if an AJAX error happens, this function will be called and you can re-enable the appropriate buttons or reset the appropriate **variables**.

<exercises section='12.4.2'>

1. Add an **AJAX** Error **event** handler to **ajaxTime.html** so that the page leaves its "waiting" state if there is an error. To test, you can create a **404 File Not Found** error by temporarily renaming or moving the **servers/timeServer.php** file.

2. Add an AJAX Error event handler to your solution to the **more.html** exercise from the previous sections. Make sure the error handler works and gets the user out of the waiting state.

</exercises>

12.5 Working With XML Data

You can use **AJAX** requests to retrieve any type of document that is accessible through **HTTP** requests on the World Wide Web. These documents do not have to be in **XML** format despite the "X" in the name AJAX (indeed so far we have only been using plain text files). But most of the time, information does come with structure attached, and XML is by far the most common way of representing that structure.

12.5.1 Basic XML Structure

Here's some data from a book store, adapted from w3Schools.

```
cooking, Everyday Italian, Giada De Laurentiis, 2005, 30.00
web, xQuery Kick Start, James McGovern, Per Bothner, Kurt Cagle, James Linn,
Vaidyanathan Nagarajan, 2003, 49.99
web, paperback, Learning XML, Erik T. Ray, 2003, 39.95
programming, JavaScript for Sheridan Students, Sam Scott, 2013, 0.00
```

This information is a little hard to read, but if you look carefully you might notice it appears to be data for 4 different books, along with book type, title, authors, year of publication, and price.

Here is the same information in XML format:

```
<bookstore>
    <book category="cooking">
        <title lang="en">Everyday Italian</title>
        <author>Giada De Laurentiis</author>
        <year>2005</year>
        <price>30.00</price>
    </book>
    <book category="web">
        <title lang="en">XQuery Kick Start</title>
        <author>James McGovern</author>
        <author>Per Bothner</author>
        <author>Kurt Cagle</author>
        <author>James Linn</author>
        <author>Vaidyanathan Nagarajan</author>
        <year>2003</year>
        <price>49.99</price>
    </book>
    <book category="web" cover="paperback">
```

```
        <title lang="en">Learning XML</title>
        <author>Erik T. Ray</author>
        <year>2003</year>
        <price>39.95</price>
    </book>
    <book category="programming">
        <title lang="en">JavaScript for Sheridan Students</title>
        <author>Sam Scott</author>
        <year>2013</year>
        <price>0.00</price>
    </book>
</bookstore>
```

Notice how much easier it is to see the structure of this document. HTML-like tags are being used to provide it with structure and make it easier to work with, both for machines and for humans. When everybody uses XML, data can be more easily transported and shared between applications.

12.5.2 XML vs. HTML

Here's a list of similarities between XML and HTML:

1. XML and HTML are both **markup languages** with tags that define **elements** containing data. For example, the tag `<author>J K Rowling</author>` defines an element of type `<author>` with the contents "J K Rowling".
2. XML and **HTML tags** can contain **attributes**. For example `<book category="cooking">` is the start of an element of type `<book>` that contains the attribute category. Attributes are descriptive information, or metadata that provides additional descriptive information about the contents of the element.
3. XML and HTML documents must have a single root element that contains all the other elements. In HTML, this is the `<html>` element. In XML it can be any element you want. What is the root element in the book store example from the last section?
4. XML and HTML comments can be added using `<!-- -->`.
5. Both XML and HTML documents are represented in browsers using a **Document Object Model** (**DOM**). More on this in a moment.

And here's a list of differences:

1. XML documents do not do anything. XML is designed only to carry structured data, not to display it. XML tags and attributes cannot be read as commands like HTML tags and attributes sometimes can be.
2. XML has no predefined tags. You can use any set of single-word tag names you want. You can also define your own document types, but we won't get into that here.
3. XML tags and attributes are case sensitive. For example, `<p>` and `<P>` are the same tag in HTML, but would define two different element types in XML.
4. XML attribute **values** must be quoted. In HTML you can sometimes get away with `<p id=myparagraph></p>`. In XML this would cause a **syntax error** because there are no quotation marks around `myparagraph`.
5. XML tags must be closed. In HTML there are many empty tags (`<br>`, `<img>`, etc.) that do not need a closing tag. In XML you must either provide a closing tag, or if the tag has no data you can use a self-closing tag like this `<search query="AJAX"/>`. Self-closing tags end with "/>".
6. XML tags must be properly nested. Nesting is the placing of one structure within another. In HTML, you can do this: `<p>AJAX <b>is</p> cool</b>`. This is ok, even though the `<b>` tag is partly inside and partly outside the `<p>` tag. In XML this would cause a **syntax** error.

7. XML syntax errors will usually (silently) crash an application. Unlike HTML, which always tries to make the best of bad syntax, XML parsers will stop if they find syntax errors.

12.5.3 Navigating the XML DOM

When you go into the Chrome developer console and look at the "elements" view of the document, you are looking at the HTML Document **Object** Model (DOM). When you work with the **JavaScript** `document` object, and use it to make changes to an HTML page, you are accessing and modifying the HTML DOM.

XML documents can also be represented and accessed using a DOM. The act of converting a text representation of an XML file into a DOM object is called **parsing**. Chrome contains a special JavaScript object called `DOMParser` that does this very job.

Do It Yourself

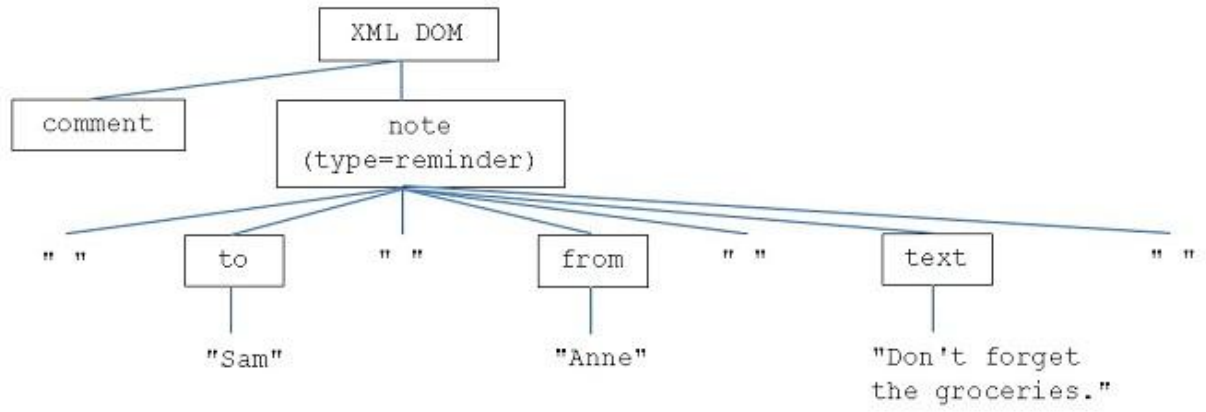
Open your browser's JavaScript console and type the following (or to save typing or copying and pasting, you can use the pre-baked example **note.html** in the **example pack**):

```
var xmldoc = "<note type='reminder'><to>Sam</to><from>Anne</from>
<text>Don't forget the groceries.</text></note>";
var parser = new DOMParser();
var dom = parser.parseFromString(xmldoc,"text/xml");
```

The first command simply defines a **string variable** with **XML** data in it. The second command creates a new `DOMParser` **object**. The third command **parses** the variable and creates a document object.

Now type `dom`, open the document object and examine the structure. Do the same for `document`. Note any similarities or differences you find.

The XML DOM **tree** is constructed just like the HTML DOM trees you learned about in Chapters 4 and 10. The simple `<note>` example above has the following structure:



Here's a terminology reminder. In the picture above, there are 13 Nodes (8 are DOM Nodes and 5 are TextNodes, including whitespace TextNodes - see Chapter 11). The **root node** is `<note>`. It has three **children** or **child nodes**. Each of the children share the same **parent**. The line from parent to child means that the child is contained within the parent. If there are many children, they are shown in order from left to right. Nodes that share a parent are **siblings** (this is the gender-neutral term for "brother" or "sister"). Each of the children of `<note>` has two siblings.

A DOM object in JavaScript is a collection of **Node** objects that are all linked together in a tree structure. This is true whether the DOM is for an XML document or an HTML document, so most of what you learned about the HTML DOM in Chapters 4, 7, and 10 applies to the XML DOM as well. Instead of `document` at the root, you find a DOM object, and every piece of text also appears as a special child node called a Text Node.

Do It Yourself

Go back to the bookstore example from a previous section and draw the **DOM** for this example on a piece of paper.

As with the HTML DOM, you can navigate the structure by moving up and down along the links with the following **fields** and **methods**.

`childNodes`:	returns an array of all child nodes of the current node
`firstChild`:	returns the first child of the current node
`lastChild`:	returns the last child of the current node
`nextSibling`:	returns the sibling to the right of the current node
`previousSibling`:	returns the sibling to the left of the current node
`data`:	if this is a text node, gets the text
`getAttribute()`:	(new to XML Nodes) if the specified attribute exists for the current node, returns its value

Do It Yourself

Continue the `<note>` example from the previous session (**note.html**). Try the following in the browser console and note the results:

```
dom.childNodes
dom.firstChild
dom.lastChild
dom.childNodes[0]
dom.firstChild.getAttribute("type")
```

Can you explain the result in each case?

Now try **dom**.firstChild.childNodes and explain the result.

Now for a challenge. Find as many ways as you can to get the text of the <from> **element**. Remember that the text is a special child **node**, not an **attribute** or a **value**.

Just like with the `document` object, you can also use tag names and `id` attributes (if the XML document has them) to find particular elements inside the DOM. The methods are the same as for the HTML DOM: `getElementById`, `getElementsByTagNames`, `querySelector` and `querySelectorAll`.

These methods can be used to search within any node, not just the root node. For example if you wanted the author name of the third book in the bookstore example, you could load the XML DOM into a **variable** called dom and do the following:

```
dom.getElementsByTagName("book")[2].getElementsByTagName("author")[0]
```

The first `getElementsByTagName` retrieves a list of all the `<book>` elements. The `[2]` retrieves the third node from the resulting **array** of `<book>` elements. Then the next `getElementsByTagName` looks just inside that node for `<author>` elements, and the `[0]` retrieves the first one from the array of found nodes.

Do It Yourself

Using the `<note>` example again (**note.html**), find two more ways to get the text of the `<from>` **element**, using one of the two convenience **methods** above.

12.5.4 Loading XML With AJAX

When you use **jQuery**, loading an XML document is almost as easy as loading plain text. You can use the same commands (`$.get` and `$.post`) with exactly the same parameters as before. The only difference is that the data parameter of the **callback function** will be converted by jQuery into an XML document using the `DOMParser`.

Troubleshooting Tip

If you are trying to put some data from an XML file (say a province name) into your output and you keep getting things that look like this:

```
[object Text]
```

This means that you are trying to print out a Text node. But you don't want to print the whole node, just the data it contains. To do this you need to access the data field.

For example, this won't work:

```
textOutput += dom.getElementsByTagName("name").firstChild
```

Do this instead:

```
textOutput += dom.getElementsByTagName("name").firstChild.data
```

<exercises section='12.5.4'>

This stuff can be tricky! Make sure you give yourself lots of time. Perhaps even working with a partner would be a good idea.

1. Load **ajaxSandbox.html** from the **example pack** into a browser, then open the JavaScript console. (Remember that the html file must be hosted on a server along with the rest of the files for the example pack.)
    a. Use the following command to load the data from the **XMLData/note.xml** file:

    ```
    $.get("XMLData/note.xml",function(data) {x = data;});
    ```

    Note that there are no parameters required, so we are only using the URL and the **callback function** parameters. The callback function simply stores the data parameter into a **global variable** x.

    b. Once you have executed this command, open the x **object** and take a look at its structure to make sure it worked.

    c. Write a **JavaScript** expression that retrieves the <note> **element** from x. Notice that the root of x actually has two children instead of one. Why is that?

    d. Now write a JavaScript **statement** to load **XMLData/bookstore.xml** through **AJAX**. In the callback function, put an alert statement that displays the author name and language **attribute** for the second book in the store.

2. Complete the exercise described in the header comments of the **cdcatalog.html** file.

3. Complete the exercise described in the header comments of the **canada.html** file.

4. Complete the exercise described in the header comments of the **loremIpsum.html** file.

</exercises>

12.5.5 Learning More About XML

XML is a big topic and there is lots more that can be said. For those who want to go further, **W3Schools** has good tutorials on all things XML. On the front page of w3schools, scroll down to XML tutorials and read through the **Learn XML** and **Learn XML DOM** tutorials in particular for more information.