# Chapter 5 - Responsive Web Design

## 5.1 Mobile Web

The introduction of the iPhone to the market in 2007 changed the world forever. Up to that point, cellphones had small screens and were not powerful enough to load and display most of the websites available at the time. If we wanted to browse the Web, we had to do it on a computer. These conditions simplified the work for developers. There was only one type of screen, and it was wide enough to include everything they needed. The most popular resolutions at that time included at least 1024 pixels horizontally, which allowed developers to design their websites with a standard size (like the one introduced in the previous chapter). But when the iPhone came out, websites with fixed layouts were difficult to read on such a small screen. The first solution was to develop two separate websites, one for computers and another for iPhones, but the introduction of new devices to the market, such as the iPad in 2010, forced developers to look for better alternatives. Making the elements flexible was one of them, but it was not enough. Pages with multiple columns still looked awful on small screens. The final solution had to include flexibility and the possibility to adapt the design to every device. This is how Responsive Web Design was born.

Responsive Web Design is a technique that combines flexible layouts with a tool provided by CSS called *Media Queries* that lets us detect the size of the screen and perform the necessary changes to adapt the design to every situation.

## Media Queries

A Media Query is a reserved rule in CSS that was incorporated to allow developers to detect the media in which the document is being shown. For instance, using Media Queries, we can determine whether the document is being displayed on a monitor or sent to a printer and assign the appropriate styles for each case. For this purpose, Media Queries offer the following keywords.

**all**—The properties are applied to all media.

**print**—The properties are applied when the page is sent to a printer.

**screen**—The properties are applied when the page is shown on a color screen.

**speech**—The properties are applied when the page is processed by a speech synthesizer.

What makes Media Queries useful for web design is that they can also detect the size of the media. With Media Queries, we can do things like detecting the size, resolution and orientation of the viewport (the part of the browser window where our pages are displayed), and from this information define different CSS rules for each device. There are several keywords available to detect these characteristics. The following are the most frequently used in Responsive Web Design.

**width**—This keyword determines the width at which the properties will be applied.

**height**—This keyword determines the height at which the properties will be applied.

**min-width**—This keyword determines the minimum width from which the properties will be applied.

**max-width**—This keyword determines the maximum width to which the properties will be applied.

**aspect-ratio**—This keyword determines the aspect ratio at which the properties will be applied.

**orientation**—This keyword determines the orientation at which the properties will be applied. The possible values are **portrait** and **landscape**.

**resolution**—This keyword determines the pixel density at which the properties will be applied. It can take values in dots per inch (dpi), dots per centimeter (dpcm) or by pixel ratio (dppx). For instance, to detect a screen with a Retina display of a scale of 2, we can use the value **2dppx**.

Using these keywords, we can detect certain aspects of the media and the viewport where the page will be displayed, and modify the properties to adjust the design accordingly. For example, if we define the Media Query with the **width** keyword and the value **768px**, the properties will only be applied when the page is displayed in regular iPads in portrait mode, since that is the width of the viewport on that device and condition.

To define a Media Query, we can declare only the keywords and values we need. The keywords that describe a feature, such as the width of the viewport, have to be declared between parentheses. If more than one keyword is present, we can associate them with the logical operators **and** or **or**. For example, the Media Query **all and (max-width: 480px)** assigns the properties to the document in all medias but only when the width of the viewport is 480 pixels or less.

There are two ways to declare Media Queries, we can do it from the document, using the **media** attribute of the **<link>** element, or from the Style Sheet with the **@media** at-rule. When we use the **<link>** element, we can select the CSS file with the Style Sheet we want to load for a specific configuration. For example, the following document loads two CSS files, one with generic styles for every situation and media and another with the styles required to present the page in a device with a small screen (480 pixels wide or less).

***Listing 5-1:*** *Loading styles with Media Queries*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>This text is the title of the document</title>
  <meta charset="utf-8">
  <meta name="description" content="This is an HTML5 document">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <link rel="stylesheet" href="responsiveall.css">
  <link rel="stylesheet" media="(max-width: 480px)"
href="responsivephones.css">
</head>
<body>
  <section>
    <div>
```

```
      <article>
        <h1>Title First Post</h1>
        <img src="myimage.jpg">
      </article>
    </div>
  </section>
  <aside>
    <div>
      <h1>Personal Info</h1>
      <p>Quote from post one</p>
      <p>Quote from post two</p>
    </div>
  </aside>
  <div class="clearfloat"></div>
</body>
</html>
```

When the browser reads this document, it first loads the responsiveall.css file and then, if the width of the viewport is 480 pixels wide or less, it loads the responsivephones.css file and applies the styles to the document.

CSS stands for Cascading Style Sheets, which means that the properties are processed in cascade, and therefore new properties replace old ones. When we want to modify an element for a particular viewport, we have to declare the property that we want to change, but the values of the rest of the properties defined before remain the same. For instance, we can assign a background color to the body of our document in the responsiveall.css file and then change it in the responsivephones.css file, but this new value will only be applied when the document is presented on a small screen. The following is the rule for the responsiveall.css file.

***Listing 5-2:*** *Defining the styles by default (responsiveall.css)*

```
body {
  margin: 0px;
  padding: 0px;
  background-color: #990000;
}
```

The styles defined in the responsivephones.css file have to modify only the values of the properties we want to change, in this case, the **background-color** property, but the rest of the values remain as they were declared in the responsiveall.css file.

***Listing 5-3:*** *Defining the styles for small devices (responsivephones.css)*

```
body {
  background-color: #3333FF;
}
```

> **Do It Yourself:** Create a new HTML file with the document of Listing 5-1. Create a CSS file called responsiveall.css with the rule in Listing 5-2 and a CSS file called responsivephones.css with the rule in Listing 5-3. Open the document in your browser. You should see a red background. Drag one side of the window to reduce its size. When the width of the viewport gets to 480 pixels or less, you should see the background color changing to blue.

With the **<link>** element and the **media** attribute we can load different Style Sheets for every situation, but when just a few properties are changed, we can include them all in the same Style Sheet and define the Media Queries with the **@media** at-rule, as in the following example.

***Listing 5-4:*** *Declaring Media Queries with the* @media *at-rule (*responsiveall.css*)*

```
body {
  margin: 0px;
  padding: 0px;
  background-color: #990000;
}
@media (max-width: 480px) {
  body {
    background-color: #3333FF;
  }
}
```

The Media Query is declared with the **@media** at-rule followed by the keywords and values that define the characteristics of the media and the properties between braces. When the browser reads this Style Sheet, it assigns the properties of the **body** rule to the **<body>** element, and then, if the viewport's width is 480 pixels or less, it changes the background color of the **<body>** element to **#3333FF** (blue).

**Do It Yourself:** Replace the rules in your responsiveall.css file with the code in Listing 5-4 and remove the second **<link>** element from the document of Listing 5-1 (now you only need one CSS file for all your styles). Open the document in your browser and reduce the size of the window to see how CSS applies the rule defined by the Media Query when the viewport's width is equal or smaller than 480 pixels.

# Breakpoints

In the previous example, we have declared a Media Query that detects whether the viewport's width is equal or smaller than 480 pixels and applies the properties if the condition is satisfied. This is common practice. In today's market, there are several types of devices available with a broad range of screen sizes. If we use the **width** keyword to target a device, some devices we do not know or new devices entering the market may not be affected, and therefore they will display our website incorrectly. For instance, the Media Query **width: 768px** targets only regular iPads in portrait mode because those devices in that orientation have that size. An iPad in landscape mode, or an iPad Pro, which presents a different resolution, or any other device with a screen size smaller or larger than 768 pixels, will not be affected by these properties, even if the size varies in just one or two pixels. To avoid mistakes, we do not select specific sizes. Instead, we set maximum or minimum points where the design has to change, and rely on the box model to adapt the size of the elements to the space available between those points. These maximum and minimum points are declared with the **max-width** and **min-width** keywords and are called *Breakpoints*.

Breakpoints are the points at which the design of a web page has to go through significant changes to adapt to the size of the screen. These are the

widths at which flexibility is insufficient to adjust the boxes to the space available, and we have to move the elements around to keep our website usable. There are no standards that we can follow to set these points; everything depends on our design. For instance, we may decide that when the viewport's width is equal or smaller than 480 pixels, we have to use only one column to present the information, but that depends on whether in our original design we have multiple columns or not. Common values are 320, 480, 768, and 1024.

To set a breakpoint, we have to declare a Media Query with the **max-width** or **min-width** keywords, so the properties will be valid until the size of the viewport exceeds one of these limits. Which keyword to apply depends on the direction we want to follow. If we want to target small devices first, we may use the **min-width** keyword, but if we want to provide a generic design for large screens and then modify the properties as the screen gets smaller, we will be better with the **max-width** keyword. For instance, the following Style Sheet assigns a background color by default to the body, but as the screen gets smaller, the color is changed.

***Listing 5-5:*** *Adding multiple breakpoints*

```
body {
  margin: 0px;
  padding: 0px;
  background-color: #990000;
}
@media (max-width: 1024px) {
  body {
    background-color: #3333FF;
  }
}
@media (max-width: 768px) {
  body {
    background-color: #FF33FF;
  }
}
@media (max-width: 480px) {
  body {
    background-color: #339933;
```

```
  }
}
@media (max-width: 320px) {
  body {
    background-color: #CCCCCC;
  }
}
```

When the styles introduced in Listing 5-5 are assigned to the document of Listing 5-1, the browser presents the page with a red background in a viewport bigger than 1024 pixels, but it changes the color every time the width goes under any of the limits set by the Media Queries.

**Do It Yourself:** Replace the rules in the responsiveall.css file with the code in Listing 5-5. Open the document in your browser and change the size of the window. If the viewport is bigger than 1024 pixels, the body is shown with a red background, but if the size is reduced to 1024 or less, it changes to blue. At 768 pixels or less, it changes to pink, 480 pixels or less it changes to green, and 320 pixels or less it changes to gray.

# Viewport

The viewport is the part of the browser window where our web page is displayed. Because of the relationship between the window and the viewport, in mobile devices, their sizes are the same, but this is not always the case. By default, some smartphones consider the viewport to be 980 pixels wide, regardless of the real size of the screen. This means that the Media Queries in our Style Sheets will see a viewport of 980 pixels wide when the actual size is smaller. To normalize this situation and force the browser to match the size of the viewport with the size of the screen, we have to declare a **<meta>** tag in the head of our documents with the name **viewport** and values that determine the width and scale we want to see. The two values required to set the screen's size and scale are **width** and **initial-scale**. The following example illustrates how we have to configure the **<meta>** tag to ask the browser to return the real size and scale of the device's screen.

***Listing 5-6:*** *Adding the viewport* <meta> *tag*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>This text is the title of the document</title>
  <meta charset="utf-8">
  <meta name="description" content="This is an HTML5 document">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <meta name="viewport" content="width=device-width, initial-
scale=1">
  <link rel="stylesheet" href="responsiveall.css">
</head>
<body>
  <section>
    <div>
      <article>
        <h1>Title First Post</h1>
        <img src="myimage.jpg">
      </article>
    </div>
  </section>
  <aside>
    <div>
      <h1>Personal Info</h1>
      <p>Quote from post one</p>
      <p>Quote from post two</p>
    </div>
  </aside>
  <div class="clearfloat"></div>
</body>
</html>
```

**The Basics:** The viewport **<meta>** tag may include other values to set attributes like the minimum and maximum scales (**minimum-scale** and **maximum-scale**), or if you want to allow users to zoom in and out (**user-**

**scalable**). For more information, visit our website and follow the links for this chapter.

# Flexibility

Flexibility is a requirement in Responsive Web Design. We have to make our web pages flexible, so the elements adapt to the space available between one breakpoint and another. With the Flexible Box Model, this is simple, we have to create flexible containers and declare their boxes as flexible with the **flex** property (see Chapter 4), but this box model is not recognized by some browsers, and therefore most developers still use the Traditional Box Model. Although the Traditional Box Model was not designed to work with flexible elements, we can turn the elements flexible by declaring their sizes in percentage. When we declare the size of an element in percentage, the browser calculates the real size in pixels from the size of its container. For example, if we assign a width of 80% to an element that is inside a container with a width of 1000 pixels, the width of that element will be 800 pixels (80% of 1000). Because the size of the elements changes every time the size of their container changes, they become flexible.

The following example creates two columns with the **<section>** and **<aside>** elements of the document of Listing 5-6 and declares their widths with percentages to make them flexible.

***Listing 5-7:*** *Creating flexible elements with percentages*

```
* {
  margin: 0px;
  padding: 0px;
}
section {
  float: left;
  width: 70%;
  background-color: #999999;
}
aside {
  float: left;
```

```
  width: 30%;
  background-color: #CCCCCC;
}
.clearfloat {
  clear: both;
}
```

The rules in Listing 5-7 declare the sizes of the **<section>** and **<aside>** elements as 70% and 30% of the size of their container, respectively. If the size of the window changes, the size of the elements is recalculated by the browser and therefore they expand or shrink according to the space available.


***Figure 5-1:*** *Flexible elements with the Traditional Box Model*

Every time we declare the width of the elements in percentages, we have to make sure that the total sum of the values does not exceed 100%. Otherwise, the elements will not fit in the space available and will be moved to a new line. In Listing 5-7, this was simple to do because we only had two elements with no margins, paddings or borders, but as soon as we add an extra value, like the padding, we have to remember to subtract that value from the width of the element, or the total will be more than 100%. For example, the following rules reduce the width of the **<section>** and **<aside>** elements to add a padding of 2% on each side and a margin of 5% in the middle.

***Listing 5-8:*** *Adding margins and paddings to flexible elements*

```
* {
  margin: 0px;
  padding: 0px;
```

```
}
section {
  float: left;
  width: 61%;
  padding: 2%;
  margin-right: 5%;
  background-color: #999999;
}
aside {
  float: left;
  width: 26%;
  padding: 2%;
  background-color: #CCCCCC;
}
.clearfloat {
  clear: both;
}
```

The rules in Listing 5-8 assign to the **<section>** element a width of 61%, a padding of 2% at the top, right, bottom, and left side, and a margin of 5% on the right side, and to the **<aside>** element a width of 26% and a padding of 2% at the top, right, bottom, and left side. Because the sum of these values does not exceed 100% (61 + 2 + 2 + 5 + 26 + 2 + 2 = 100), the elements are positioned side by side on the same line.


***Figure 5-2:*** *Flexible elements with paddings and margins*

**Do It Yourself:** Replace the rules in the responsiveall.css file with the code in Listing 5-8. Include the image myimage.jpg in the document's folder. Open the document in your browser. You should see something similar to Figure 5-2.

# Box-sizing

Every time the total area occupied by an element is calculated, the browser gets the final number with the formula: size + margin + padding + borders. Therefore, if we set the **width** property to 100 pixels, **margin** to 20 pixels, **padding** to 10 pixels and **border** to 1 pixel, the total horizontal area occupied by the element will be **100 + 40 + 20 + 2 = 162** pixels (notice that we had to double the values of the **margin**, **padding** and **border** in the formula because we assumed the same values were assigned to the left and right side of the box). This means that every time we declare the size of an element with the **width** property, we have to remember that the area occupied by the element on the page may be larger. This is particularly troublesome when the sizes are defined in percentage, or we try to combine percentage with other types of units like pixels. CSS includes the following property to modify this behavior.

**box-sizing**—This property allows us to change how the size of an element is calculated and force browsers to include the padding and border in the size declared by the **width** property. The possible values are **content-box** and **border-box**.

By default, the value of the **box-sizing** property is set to **content-box**, which means the browser will add the values of the **padding** and **border** properties to the size specified by **width**. If we assign to the property the value **border-box**, the browser includes the padding and the border as part of the width, so we can do things like combining widths defined in percentage with paddings in pixels, as in the following example.

***Listing 5-9:*** *Including padding and border in the size of the element*

```
* {
  margin: 0px;
```

```
  padding: 0px;
}
section {
  float: left;
  width: 65%;
  padding: 20px;
  margin-right: 5%;
  background-color: #999999;
  box-sizing: border-box;
}
aside {
  float: left;
  width: 30%;
  padding: 20px;
  background-color: #CCCCCC;
  box-sizing: border-box;
}
.clearfloat {
  clear: both;
}
```

In Listing 5-9, we declare the **box-sizing** property with the value **border-box** for both elements, **<section>** and **<aside>**. Because of this, we no longer have to consider the padding to calculate the size of the elements (65 + 5 + 30 = 100).

**Do It Yourself:** Replace the rules in the responsiveall.css file with the code in Listing 5-9. Open the document in your browser. You should see something similar to Figure 5-2, but this time the paddings do not change when you resize the window because they were declared with a fixed value of 20 pixels.

# Fixed and Flexible

In the previous example, we have seen how to define flexible boxes with fixed paddings, but there are times when we have to combine entire

columns with fixed and flexible values. Again, this is very easy to do with the Flexible Box Model. We have to create a flexible container, declare the size of the elements we want to be flexible with the **flex** property and those we want to have a fixed size with the **width** property (see Chapter 4, Listing 4-37). But the Traditional Box Model was not prepared to deal with these kinds of tasks. Fortunately, there are a few techniques we can implement to achieve this purpose. The most popular alternative requires us to declare a padding on the flexible column to make room for the fixed column and a negative margin to displace the fixed column to the space left by the padding. The following are the rules we need to declare a flexible width for the **<section>** element and a fixed width for the **<aside>** element of our document.

***Listing 5-10:*** *Declaring flexible and fixed columns*

```
* {
  margin: 0px;
  padding: 0px;
}
section {
  float: left;
  width: 100%;
  padding-right: 260px;
  margin-right: -240px;
  box-sizing: border-box;
}
section > div {
  padding: 20px;
  background-color: #999999;
}
aside {
  float: left;

  width: 240px;
  padding: 20px;
  background-color: #CCCCCC;
  box-sizing: border-box;
}
```

```
.clearfloat {
  clear: both;
}
```

The rules in Listing 5-10 assign a width of 100% and a right padding of 260 pixels to the **<section>** element. Because we use the **box-sizing** property to include the padding in the value of **width**, the element's content will occupy only the left side of the element, leaving an empty space on the right where we can place our fixed column. Finally, to move the column created by the **<aside>** element over this place, we declare a negative margin of 240 pixels on the right side of the **<section>** element.

> **The Basics:** In this example, we float both columns to the left, which means that they are going to be placed side by side on the same line. In cases like this, we can create a gap between the columns by increasing the value of the **padding** property. In the example of Listing 5-10, we declared a padding of 260 pixels and a margin of 240 pixels, which means that the fixed column will move only 240 pixels to the left, leaving a space of 20 pixels in between.

Because we are using the padding to make room for the right column, we have to assign the normal padding and backgrounds to the ** ** element inside the **<section>** element that represents the column. This not only allows us to add padding to the content but it also assigns the background color only to the area occupied by the content, differentiating the two columns on the screen.


***Figure 5-3:*** *Flexible and Fixed columns*

**Do It Yourself:** Replace the rules in the responsiveall.css file with the code in Listing 5-10. Open the document in your browser. Drag one side of the window to change its size. You should see the left column expanding or shrinking and the right column always of the same size (240 pixels).

If we want to place the fixed column on the left side instead of the right side of the page, the trick is the same, but we have to declare the left column below the right column in the code (as declared in Listing 5-6) and then set their positions with the **float** property, as in the following example.

***Listing 5-11:*** *Moving the fixed column to the left*

```
* {
  margin: 0px;
  padding: 0px;
}
section {
  float: right;
  width: 100%;
  padding-left: 260px;
  margin-left: -240px;
  box-sizing: border-box;
}
section > div {
  padding: 20px;
  background-color: #999999;
}
aside {
  float: left;
  width: 240px;
  padding: 20px;
  background-color: #CCCCCC;
  box-sizing: border-box;
}
.clearfloat {
  clear: both;
```

}

The **<section>** element is declared first in our document, but because we assign the value **right** to its **float** property, it is displayed on the right side of the screen. The position of the element in the code makes sure that it is not going to be drawn on top of the **<aside>** element, and the value of the **float** property moves it to the side we choose. The rest of the code is similar to the previous example, except that this time we have to modify the padding and margin of the **<section>** element on the left side instead of the right.


***Figure 5-4:*** *Fixed column on the left*

**The Basics:** When you move an element to a new position by assigning a negative margin to the next element, the browser draws the first element behind the second element, and therefore the first element does not receive any event, such as the user clicking on a link. To make sure that the fixed column remains visible and accessible, we have to declare it in the code below the flexible column. This is why, to move the column created by the **<aside>** element to the left, we did not modify our document. The column is put into place by the value of the **float** property.

If we want to declare two fixed columns on the sides and a flexible column in the middle using this same trick, we have to group the columns inside a container and then apply the properties to the elements twice, inside and outside the container. In the following document, we group two **<section>** elements inside a ** ** element identified with the name

**container** to contain the left and center columns. The two columns are identified with the names **columnleft** and **columncenter**.

***Listing 5-12:*** *Creating a flexible document with two fixed columns*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>This text is the title of the document</title>
  <meta charset="utf-8">
  <meta name="description" content="This is an HTML5 document">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <div id="container">
    <section id="columncenter">
      <div>
        <article>
          <h1>Title First Post</h1>
          <img src="myimage.jpg">
        </article>
      </div>
    </section>
    <section id="columnleft">
      <div>
        <h1>Options</h1>
        <p>Option 1</p>
        <p>Option 2</p>
      </div>
    </section>
    <div class="clearfloat"></div>
  </div>
  <aside>
    <div>
      <h1>Personal Info</h1>
      <p>Quote from post one</p>
```

```
      <p>Quote from post two</p>
    </div>
  </aside>
</body>
</html>
```

The CSS code for this document is simple. The **container** and **columncenter** elements have to be declared as flexible, so we have to define the right padding and margin of the container to make room for the **<aside>** element on the right, and then define the padding and margin for the **columncenter** element to make room for the fixed column on the left.

***Listing 5-13:*** *Defining two fixed columns on the sides*

```
* {
  margin: 0px;
  padding: 0px;
}
#container {
  float: left;
  width: 100%;
  padding-right: 260px;
  margin-right: -240px;
  box-sizing: border-box;
}
aside {
  float: left;
  width: 240px;
  padding: 20px;
  background-color: #CCCCCC;
  box-sizing: border-box;
}
#columncenter {
  float: right;
  width: 100%;
  padding-left: 220px;
  margin-left: -200px;
  box-sizing: border-box;
```

```
}
#columncenter > div {
  padding: 20px;
  background-color: #999999;
}
#columnleft {
  float: left;
  width: 200px;
  padding: 20px;
  background-color: #CCCCCC;
  box-sizing: border-box;
}
.clearfloat {
  clear: both;
}
```

Notice that the **columnleft** element was declared after the **columncenter** element in the code. This is to make sure that **columnleft** is drawn over **columncenter** and it is accessible. As we did before, the positions of these elements are defined by the **float** property. In consequence, the **columnleft** element is positioned on the left side of the page with a fixed size of 200 pixels, the **columncenter** element is positioned at the center with a flexible width, and the **<aside>** element is positioned on the right with a fixed width of 240 pixels.


***Figure 5-5:*** *Fixed columns on both sides*

**Do It Yourself:** Create a new HTML file with the document of Listing 5-12 and a CSS file called mystyles.css with the code in Listing 5-13. Include the myimage.jpg file in the same folder. Open the document in

your browser and change the size of the window. You should see something similar to Figure 5-5, with the columns on the sides always at the same size.

# Text

Another aspect of the design we have to adapt to different devices is the text. When we declare a font with a fixed size, like 14 pixels, the text is always shown at that size, regardless of the device. On a desktop computer it looks fine, but on small screens, a text of that size could be unreadable. To adjust the size of the font, we can use the standard font size. This is a size determined by the browser according to the device on which it is running. For example, on desktop computers, the size of the font by default is usually 16 pixels. To make sure that the text on our web page is readable, we can define its size relative to this value. CSS offers the following units for this purpose.

- **em**—This unit represents a measure relative to the size of the font assigned to the element. It takes decimal numbers. A value of 1 is equal to the font's current size.
- **rem**—This unit represents a measure relative to the size of the font assigned to the root element (the **<body>** element). It takes decimal numbers. A value of 1 is equal to the font's current size.

Fonts are defined the same way we did before, but the **px** unit must be replaced with the **em** unit to declare the size of the font relative to the current size. For example, the following Style Sheet defines different font sizes for the **<h1>** and **<p>** elements in our document using **em** units (the rules were defined considering the document of Listing 5-6).

***Listing 5-14:*** *Declaring relative font sizes*

```
* {
  margin: 0px;
  padding: 0px;
}
section {
```

```
  float: left;
  width: 61%;
  padding: 2%;
  margin-right: 5%;
  background-color: #999999;
}
aside {
  float: left;
  width: 26%;
  padding: 2%;
  background-color: #CCCCCC;
}
.clearfloat {
  clear: both;
}
article h1 {
  font: bold 2em Georgia, sans-serif;
}
aside h1 {
  font: bold 1.2em Georgia, sans-serif;
}
aside p {
  font: 1em Georgia, sans-serif;
}
```

The **em** units replace the **px** units, and therefore the browser is responsible for calculating the size of the text in pixels from the size of the font currently assigned to the element. To calculate this value, the browser multiplies the number of **em** by the current size of the standard font. For example, the rule **article h1** in Listing 5-14 assigns a size of **2em** to the **<h1>** elements inside the **<article>** elements. This means that on a desktop computer the text in these elements will have a size of 32 pixels (16 * 2).


***Figure 5-6:*** *Relative font size*

**Do It Yourself:** The example of Listing 5-14 assumes that you are working with the document of Listing 5-6. Replace the rules in the responsiveall.css file created for this document with the code in Listing 5-14 and open the document in your browser. You should see something similar to Figure 5-6.

The **em** unit defines the size relative to the current size of the font in the element. This is because elements inherit properties from their containers and therefore the size of their font may be different from the standard set by the browser. This means that if we modify the size of the font in the element's container, it will affect the element's font size. For example, the following rules assign a font size to the **<article>** element and another for the **<h1>** elements inside.

***Listing 5-15:*** *Declaring relative sizes for elements and containers*

```
article {
  font: bold 1.5em Georgia, sans-serif;
}
article h1 {
  font: bold 2em Georgia, sans-serif;
}
```

The first rule assigns a font size of **1.5em** to the **<article>** element. After the property is applied, the content of the **<article>** element, including the content of the **<h1>** element inside, will have a size 1.5 times bigger than

the standard size (24 pixels in a desktop computer). The **<h1>** element inherits this value, so when the next rule is applied, the size of the font is not calculated from the standard size but from the size established by the container (16 * 1.5 * 2 = 48). The result is shown in Figure 5-7.


***Figure 5-7:*** *Font size relative to the container*

If we want to change this behavior and force the browser to calculate the size of the font always from the standard value, we can use **rem** units. These units are relative to the root element instead of the current element, and therefore the values are always multiplied against the same base value.

***Listing 5-16:*** *Declaring relative sizes for the elements with* rem *units*

```
article {
  font: bold 1.5rem Georgia, sans-serif;
}
article h1 {
  font: bold 2rem Georgia, sans-serif;
}
```

If we apply the rules in Listing 5-16 to the example of Listing 5-14, the browser calculates the size of the font from the standard value and shows the article's title only two times bigger, regardless of the size declared for the container, as illustrated in Figure 5-6.

**The Basics:** You may also use **em** units instead of percentages to set the size of the elements. When you use **em** units, the size is determined by the size of the font instead of the container. This technique is used to

create Elastic Layouts. For more information, visit our website and follow the links for this chapter.

# Images

By default, images are shown in their original size, which means that they do not adapt to the space available unless we declare it explicitly. Turning a fixed image into a flexible image is very simple; we have to declare its size in percentage. For instance, the following rule sets the width of the **<img>** elements inside the **<article>** elements of our document to 100%, so the images will always be as wide as their container.

***Listing 5-17:*** *Adapting the size of the images*

```
article img {
  width: 100%;
}
```

**Do It Yourself:** Add the rule in Listing 5-17 to the Style Sheet created for the previous example and open the document in your browser. You should see the image expanding or shrinking to fit the column as you change the size of the window.

Assigning a percentage to the image's width forces the browser to calculate the size of the image according to the size of its container (in our document, this is the **<article>** element). We can declare lower values than 100%, but the size of the image will always be proportional to the size of its container, which means that if the container is expanded, the size of the image could be bigger than the original. If we want to establish limits to how much the image can expand or shrink, we can use the **max-width** and **min-width** properties. We have seen these properties before applied to the Flexible Box Model, but they are also used in the Traditional Box Model to declare a limit for a flexible element. For example, if we want the image to be expanded until it reaches its original size, we can use the **max-width** property.

***Listing 5-18:*** *Setting a maximum size for the images*

```
article img {
  max-width: 100%;
}
```

The rule in Listing 5-18 lets the image expand until it reaches its original size. The image will be as wide as its container unless the container is bigger than the image's original size.


***Figure 5-8:*** *Image with a maximum width*

**Do It Yourself:** Replace the rule in Listing 5-17 in your Style Sheet with the rule in Listing 5-18. Open the document in your browser and change the size of the window. You should see the image expanding until it reaches its original size, as shown in Figure 5-8.

Besides adapting the image to the space available, a responsive website also requires the images to be replaced when the conditions change. There are at least two situations in which this becomes necessary; either there is not enough space available for the original image to be shown, like when a logo has to be replaced with a simplified version, or the pixel density of the screen is different and we need to show an image with more or less resolution. No matter the reason, HTML offers the following elements to select the image to be shown.

**<picture>**—This element is a container that allows us to specify multiple sources for the same **<img>** element.

**<source>**—This element defines a possible source for the **<img>** element. It may include the attributes **media**, to specify the Media Query

the resource is associated with, and **srcset**, to specify the path of the images we want to declare as resources.

The **<picture>** element is a container that must include one or more **<source>** elements to specify the possible sources for the image and an **<img>** element at the end to load and display the selected image on the screen. The following document illustrates how to replace the logo of a website with a simpler version on devices with small screens using these elements.

***Listing 5-19:*** *Selecting the image with the* <picture> *element*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>This text is the title of the document</title>
  <meta charset="utf-8">
  <meta name="description" content="This is an HTML5 document">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
  <header>
    <picture>
      <source media="(max-width: 480px)" srcset="logosmall.jpg">
      <img src="logo.jpg">
    </picture>
  </header>
</body>
</html>
```

In the document of Listing 5-19, we declare only one resource and associate it with the Media Query **max-width: 480px**. When the browser reads this document, it first processes the **<source>** element and then assigns the resource to the **<img>** element if the viewport complies with the Media Query. If no match is found, the image specified by the **src** attribute of the **<img>** element is displayed. This means that every time the viewport is bigger than 480 pixels the logo.jpg image is loaded and displayed.


***Figure 5-9:*** *Original logo*

If the viewport's width is less or equal than 480 pixels, the browser assigns the image specified by the **<source>** element to the **<img>** element and the logosmall.jpg image is displayed.


***Figure 5-10:*** *Logo for small screens*

**Do It Yourself:** Create a new HTML file with the document of Listing 5-19. Download the images logo.jpg and logosmall.jpg from our website and move them to the document's folder. Open the document in your browser and change the size of the window. Depending on the current size, you should see something similar to Figure 5-9 or Figure 5-10.

Although this is probably the most common scenario in which an image has to be replaced with another of different size or content, there is one more condition we must consider when delivering images to the users. Since the introduction of Retina displays by Apple in 2010, screens with higher pixel density became available. Higher density means more pixels per inch, which translates into sharper images. But if we want to take advantage of this feature, we have to provide images with higher resolution for these screens. For instance, if we have an image of 300 by 200 pixels it will look fine in screens with normal density, but devices with higher density will have to stretch the image to occupy all the pixels available in the same area. If we want this image to look right in Retina displays of a scale of 2 (twice the number of pixels per area), we have to replace it with the same image with twice the resolution (600 x 400 pixels).

Media Queries already consider this situation with the **resolution** keyword. The keyword requires the value to be declared as an integer in **dppx** units. A value of 1 means a standard resolution. Currently, Retina

displays come in scales of 2 and 3. The following document declares a source for a display with a scale of 2.

***Listing 5-20:*** *Selecting images for different resolutions*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>This text is the title of the document</title>
  <meta charset="utf-8">
  <meta name="description" content="This is an HTML5 document">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <section>
    <div>
      <article>
        <h1>Title First Post</h1>
        <picture>
          <source media="(resolution: 2dppx)"
srcset="myimage@2x.jpg">
          <img src="myimage.jpg">
        </picture>
      </article>
    </div>
  </section>
  <aside>
    <div>
      <h1>Personal Info</h1>
      <p>Quote from post one</p>
      <p>Quote from post two</p>
    </div>
  </aside>
  <div class="clearfloat"></div>
</body>
</html>
```

If this document is opened on a screen with a standard density of pixels, the browser shows the image myimage.jpg, but when the device has a Retina display with a scale of 2, the browser assigns the image myimage@2x.jpg to the **<img>** element and shows it on the screen.

As illustrated by the example of Listing 5-20, the process is simple. We have to create two images, one with normal resolution and another with higher resolution for devices with Retina displays, and then declare the sources available with **<source>** elements. But there is a problem. If we do not specify the size of the images, they are shown in their original size, and therefore the image with high resolution will be twice the size of the image with normal resolution. For this reason, every time we work with low and high-resolution images, we have to declare their sizes explicitly. For instance, the following rules set the width of the image to be 100% the size of its container, which will assign the same size to both images.

***Listing 5-21:*** *Declaring the size of an image with high resolution*

```
* {
  margin: 0px;
  padding: 0px;
}
section {
  float: left;
  width: 61%;
  padding: 2%;
  margin-right: 5%;
  background-color: #999999;
}
aside {
  float: left;
  width: 26%;
  padding: 2%;
  background-color: #CCCCCC;
}
.clearfloat {
  clear: both;
}
```

```
article img {
  width: 100%;
}
```

This is the same we have done before, but this time the browser displays images of different resolution depending on the device. Figure 5-11 shows what the document looks like on a Retina display (we have added the text "2X" at the bottom of the myimage@2x.jpg file to differentiate the high-resolution image from the low-resolution image).

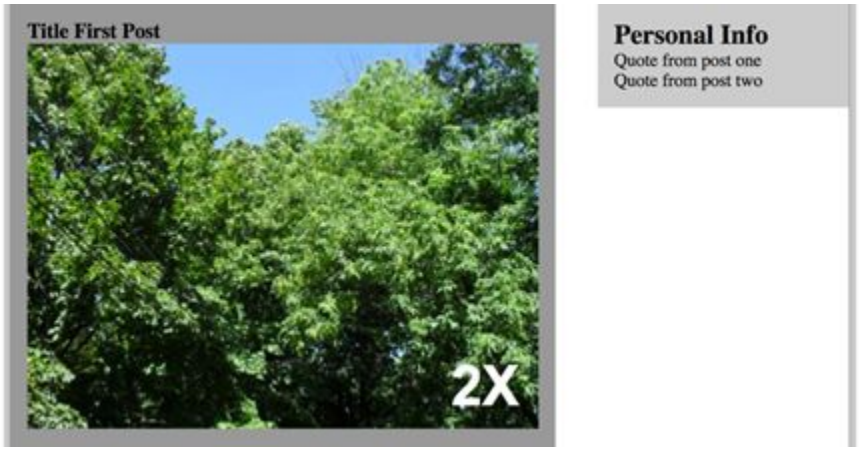


***Figure 5-11:*** *High-resolution image shown in Retina displays*

As we did in a previous example, we can use the **max-width** property to limit the width of the image to its original size (see Listing 5-18), but because this time we are working with two images of different sizes, we cannot declare the value of this property in percentage. Instead, we have to use the exact width we want to be the maximum. In our example, the image in the myimage.jpg file has a width of 365 pixels, so this is the value we have to use.

***Listing 5-22:*** *Declaring the maximum size of an image with high resolution*

```
article img {
  width: 100%;
  max-width: 365px;
}
```

Now, independently of the image selected by the browser, they are always shown with a maximum size of 365 pixels.


***Figure 5-12:*** *Limitations for high-resolution images*

**Do It Yourself:** Create a new HTML file with the document of Listing 5-20 and a CSS file called mystyles.css with the code in Listing 5-21. Download the images myimage.jpg and myimage@2x.jpg from our website and move them to the document's folder. Open the document in your browser. If your computer includes a Retina display, you should see the image with the text "2X" at the bottom, as illustrated in Figure 5-11. Update the Style Sheet with the rule in Listing 5-22 and refresh the page. You should see the image expanding until it reaches its original size (365 pixels).

**The Basics:** By convention, a file containing the high-resolution image must use the same name than the file with the image in low resolution, but followed by the @ character, the scale value, and the letter x, as in @2x or @3x. This is why we assigned the name myimage@2x.jpg to the high-resolution image of our example.

As well as the images introduced with the **<img>** element, background images can also be replaced, but in this case, we do not need to use an HTML element because we can do everything from CSS using the **background** or the **background-image** properties. Therefore, the image is not defined in the document but in the Style Sheet. The document only has to include the element to which we want to assign the background.

***Listing 5-23:*** *Including the element to show the background image*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>This text is the title of the document</title>
  <meta charset="utf-8">
  <meta name="description" content="This is an HTML5 document">
  <meta name="keywords" content="HTML, CSS, JavaScript">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <header>
    <div id="logo"></div>
  </header>
</body>
</html>
```

The header of this document contains a ** ** element identified with the name **logo** that we are going to use to show the logo of our website. As we did before, to change the value of a property in different conditions, we have to define a value by default and then declare Media Queries to change it according to the circumstances.

***Listing 5-24:*** *Updating the background image*

```
* {
  margin: 0px;
  padding: 0px;
}
#logo {
  width: 275px;
  height: 60px;
  background: url("logo.jpg") no-repeat;
}
@media (max-width: 480px) {
  #logo {
    width: 60px;
    background: url("logosmall.jpg") no-repeat;
```

```
  }
}
```

The Style Sheet of Listing 5-24 defines the size of the ** ** element to be equal to the size of the image we want to show and then assigns the logo.jpg image as its background (see Figure 5-9). If the viewport's width is 480 pixels, the Media Query adjusts the size of the ** ** element to fit the size of the small logo and assigns the logosmall.jpg image to the background (see Figure 5-10). The effect produced by this code is exactly the same achieved with the **<picture>** element in the document of Listing 5-19, except this time the image is not presented with the **<img>** element but assigned as the background of the element.

**Do It Yourself:** Create a new HTML file with the document of Listing 5-23 and a CSS file called mystyles.css with the code in Listing 5-24. Download the images logo.jpg and logosmall.jpg from our website and move them to the document's folder. Open the document in your browser and change the size of the window. Depending on the current size, you should see something similar to Figure 5-9 or Figure 5-10.

# Real-life Application

Creating a responsive website demands the combination of all the techniques we have just studied. Some have to be applied several times, and usually, more than one Media Query has to be declared to adapt the design to multiple devices. But where the breakpoints are established and how the techniques are implemented depend on our website's design and the box model we use. To illustrate how to create a website with Responsive Web Design and the Traditional Box Model, we are going to use the document introduced in Chapter 4, Listing 4-20. The basic structure of this document includes a header, a navigation bar, two columns created with a **<section>** and an **<aside>** element, and a footer. The first thing we have to do is to define the styles by default and make these elements flexible.

As mentioned before, to turn the elements flexible in the Traditional Box Model, we have to define their widths with values in percentage. The following are the styles by default required by the document of Chapter 4,

Listing 4-20. Unlike what we did in Chapter 4, this time the sizes are declared in percentage, and we use the **max-width** property to specify a maximum width for the content of the page.

***Listing 5-25:*** *Defining flexible elements with the Traditional Box Model*

```
* {
  margin: 0px;
  padding: 0px;
}
#logoheader {
  width: 96%;
  height: 150px;
  padding: 0% 2%;
  background-color: #0F76A0;
}
#logoheader > div {
  max-width: 960px;
  margin: 0px auto;
  padding-top: 45px;
}
#logoheader h1 {
  font: bold 54px Arial, sans-serif;
  color: #FFFFFF;
}
#mainmenu {
  width: 96%;
  height: 50px;
  padding: 0% 2%;
  background-color: #9FC8D9;
  border-top: 1px solid #094660;
  border-bottom: 1px solid #094660;
}
#mainmenu > div {
  max-width: 960px;
  margin: 0px auto;
}
#mainmenu li {
```

```
  display: inline-block;
  height: 35px;
  padding: 15px 10px 0px 10px;
  margin-right: 5px;
}
#mainmenu li:hover {
  background-color: #6FACC6;
}
#mainmenu a {
  font: bold 18px Arial, sans-serif;
  color: #333333;
  text-decoration: none;
}
main {
  width: 96%;
  padding: 2%;
  background-image: url("background.png");
}
main > div {
  max-width: 960px;
  margin: 0px auto;
}
#mainarticles {
  float: left;
  width: 65%;
  padding-top: 30px;
  background-color: #FFFFFF;
  border-radius: 10px;
}
#asideinfo {
  float: right;
  width: 29%;
  padding: 2%;
  background-color: #E7F1F5;
  border-radius: 10px;
}
#asideinfo h1 {
```

```
  font: bold 18px Arial, sans-serif;
  color: #333333;
  margin-bottom: 15px;
}
.clearfloat {
  clear: both;
}
article {
  position: relative;
  padding: 0px 40px 20px 40px;
}
article time {
  display: block;
  position: absolute;
  top: -5px;
  left: -70px;
  width: 80px;
  padding: 15px 5px;

  background-color: #094660;
  box-shadow: 3px 3px 5px rgba(100, 100, 100, 0.7);
  border-radius: 5px;
}
.daynumber {
  font: bold 36px Verdana, sans-serif;
  color: #FFFFFF;
  text-align: center;
}
.dayname {
  font: 12px Verdana, sans-serif;
  color: #FFFFFF;
  text-align: center;
}
article h1 {
  margin-bottom: 5px;
  font: bold 30px Georgia, sans-serif;
}
```

```
article p {
  font: 18px Georgia, sans-serif;
}
figure {
  margin: 10px 0px;
}
figure img {
  max-width: 98%;
  padding: 1%;
  border: 1px solid;
}
#logofooter {
  width: 96%;
  padding: 2%;
  background-color: #0F76A0;
}
#logofooter > div {
  max-width: 960px;
  margin: 0px auto;
  background-color: #9FC8D9;
  border-radius: 10px;
}
.footersection {
  float: left;
  width: 27.33%;
  padding: 3%;
}
.footersection h1 {
  font: bold 20px Arial, sans-serif;
}
.footersection p {
  margin-top: 5px;
}
.footersection a {
  font: bold 16px Arial, sans-serif;
  color: #666666;
  text-decoration: none;
```

```
}
```

The rules in Listing 5-25 turn the structural elements flexible, so the website adapts to the space available, but when the screen is too small, the design breaks, some elements are partially displayed, and the content becomes illegible, as shown in Figure 5-13.


***Figure 5-13:*** *Responsive website with only flexible elements*

**Do It Yourself:** Create a new HTML file with the document of Listing 4-20. Create a CSS file called mystyles.css and include the rules in Listing 5-25. Open the document in your browser. At this point, the elements expand or shrink according to the space available, but the page looks awful when the viewport is too small (see Figure 5-13). Next, we will add the Media Queries to the Style Sheet to correct this situation.

Changes in the design have to be introduced gradually. For example, in our design, when the size of the viewport is 1120 pixels or less, the **<time>** element that we use to represent the date the article was posted falls outside the window. This tells us that our design needs a breakpoint at 1120 pixels to move this element to a different position or reorganize its content. In this case, we have decided to move the date back inside the area occupied by the **<article>** element.

***Listing 5-26:*** *Repositioning the* <time> *element*

```
@media (max-width: 1120px) {
  article time {
```

```
    position: static;
    width: 100%;
    padding: 0px;
    margin-bottom: 10px;

    background-color: #FFFFFF;
    box-shadow: 0px 0px 0px;
    border-radius: 0px;
  }
  .daynumber {
    display: inline-block;
    font: bold 14px Verdana, sans-serif;
    color: #999999;
    padding-right: 5px;
  }
  .dayname {
    display: inline-block;
    font: bold 14px Verdana, sans-serif;
    color: #999999;
  }
  article h1 {
    margin-bottom: 0px;
  }
}
```

The first thing we have to do to reposition the date is to restore the positioning mode for the **<time>** element. The rules by default introduced to the Style Sheet in Listing 5-25 give the element an absolute position to move it to the right of the area occupied by the **<article>** element, but when the screen is not big enough to show it in that position, we have to move it back to its natural place in the document, below the **<h1>** element. This is done by assigning the value **static** to the **position** property (**static** is the positioning mode by default). Now, the **<time>** element is positioned below the article's title, but we still have to allocate the day's number and name on the same line. In this example, we have decided to turn the elements into **inline-block** elements, so they will be located next to each other on the same line. The result is shown in Figure 5-14.


***Figure 5-14:*** *The date is displaced below the article's title with Media Queries*

**Do It Yourself:** Add the Media Query defined in Listing 5-26 at the end of your Style Sheet and refresh the page in your browser. Reduce the size of the window to see how the **<time>** element is modified.

Another point where the design should be modified is when the two columns become too small to display their content properly. Depending on the characteristics of the content, we can choose to hide it, replace it with something else, or rearrange the columns. In this case, we have decided to turn the two-column design into a one-column design by moving the **<aside>** element below the **<section>** element. There are a few different ways to achieve this. For instance, we can assign the value **none** to the **float** property to prevent the elements from floating to the sides, or define the width of the elements as 100% or **auto** and let the browser take care of placing them one per line. For our example, we have decided to move to a one-column design when the viewport is 720 pixels or less using the second option.

***Listing 5-27:*** *Rearranging the columns*

```
@media (max-width: 720px) {
  #mainarticles {
    width: 100%;
  }
  #asideinfo {
    width: 90%;
```

```
    padding: 5%;
    margin-top: 20px;
  }
}
```

Every time the document is displayed on a screen of a size of 720 pixels or less, the user will see the content organized in just one column.


***Figure 5-15:*** *One-column design*

**Do It Yourself:** Add the Media Query defined in Listing 5-27 at the end of your Style Sheet and refresh the page in your browser. When the viewport's width is 720 pixels or less, you should see the content in only one column, as illustrated in Figure 5-15.

At this point, the document looks good on computers and tablets, but there are a few more changes we have to make to adapt it to the small screens provided by smartphones. When this page is shown in a viewport of 480 pixels or less, the options in the menu may not have enough space to be displayed on a single line, and the footer may look crowded. One way to solve these issues is to list the items inside these sections one per line.

***Listing 5-28:*** *Creating a mobile menu*

```
@media (max-width: 480px) {
  #logoheader > div {
    text-align: center;
  }
```

```
  #logoheader h1 {
    font: bold 46px Arial, sans-serif;
  }
  #mainmenu {
    width: 100%;
    height: 100%;
    padding: 0%;
  }
  #mainmenu li {
    display: block;
    margin-right: 0px;
    text-align: center;
  }
  .footersection {
    width: 94%;
    text-align: center;
  }
}
```

With the rules in Listing 5-28, we modify the elements to force the browser to show them one per line and center their content. The first two rules center the content of the **logoheader** element and assign a new size to the page's title to make it look better on small screens. Next, we define the size of the **mainmenu** element (the container of the menu) to have the maximum width possible and a height determined by its content (**height: 100%**). We also declare the **<li>** elements inside the **mainmenu** element as Block elements to display the options of the menu one per line. Finally, the widths of the three sections inside the footer are extended to force the browser to show them one per line. Figure 5-16 illustrates how these changes affect some of the elements.


***Figure 5-16:*** *Mobile menu*

**Do It Yourself:** Add the Media Query defined in Listing 5-28 at the end of your Style Sheet and refresh the page in your browser. Reduce the size of the window to see how the options in the menu and the sections in the footer adapt to the space available.

After these rules are applied, the footer looks fine, but the options of the menu at the top of the screen displace the relevant content down the page, forcing the user to scroll the page to see it. A better alternative, implemented by most developers nowadays, is to replace the menu with a button and show the options only after the button is pressed. For this purpose, we have to add an image to the document that will take the place of the menu when the viewport's width is 480 pixels or less.

***Listing 5-29:*** *Adding a menu button for small screens*

```
<nav id="menuicon">
  <img src="iconmenu.png" width="35" height="28" id="menu-img">
</nav>
<nav id="mainmenu">
  <div>
    <ul>
      <li><a href="index.html">Home</a></li>
      <li><a href="photos.html">Photos</a></li>
      <li><a href="videos.html">Videos</a></li>
      <li><a href="contact.html">Contact</a></li>
    </ul>
```

```
  </div>
</nav>
```

The first modification we have to introduce to our Style Sheet is a rule that hides the **menuicon** element because we only want to show it on small screens. There are two ways to do it; we can define the **visibility** property with the value **hidden** or declare the display mode as **none** with the **display** property. The first option does not show the element, but the element still occupies an area on the page, while the second option tells the browser to display the page as if the element was not included in the document, so this is the one we have to implement for our menu.

***Listing 5-30:*** *Hiding the menu button*

```
#menuicon {
  display: none;
  width: 95%;
  height: 38px;
  padding: 12px 2% 0px 3%;
  background-color: #9FC8D9;
  border-top: 1px solid #094660;
  border-bottom: 1px solid #094660;
}
```

The next step is to show the button and hide the menu when the viewport's width is 480 pixels or less. The following are the modifications we need to introduce at this breakpoint.

***Listing 5-31:*** *Replacing the menu with the button*

```
@media (max-width: 480px) {
  #mainmenu {
    display: none;
    width: 100%;
    height: 100%;
    padding: 0%;
  }
  #mainmenu li {
    display: block;
```

```
    margin-right: 0px;
    text-align: center;
  }
  #menuicon {
    display: block;
  }
  .footersection {
    width: 94%;
    text-align: center;
  }
  #logoheader > div {
    text-align: center;
  }
}
```

By assigning the value **none** to the **display** property of the **mainmenu** element we make the menu disappear. If the viewport's width is 480 pixels or less, the **menuicon** element and its content are shown instead.


***Figure 5-17:*** *Menu button*

**Do It Yourself:** Add a **<nav>** element identified with the name **menuicon** on top of the existent **<nav>** element in your document, as shown in Listing 5-29. Add the rule in Listing 5-30 at the top of your Style Sheet. Update the Media Query for the 480px breakpoint with the code in Listing 5-31. Open the document in your browser and reduce the size of the window. You should see the menu button replacing the menu when the viewport's width is 480 pixels or less.

Now, we have a menu that adapts to the space available, but the button is not responsive. To show the menu when the user presses or clicks the button, we have to program our document to detect the action. These actions are called *events* and are managed by JavaScript code. We will study JavaScript and events in Chapter 6, but the process we have to perform in this example is fairly straightforward. We have to turn the **mainmenu** element visible when the button is pressed by the user. The following is a possible implementation.

***Listing 5-32:*** *Displaying the menu when the button is pressed*

```
<script>
  var visible = false;
  function initiate() {
    var elem = document.getElementById("menu-img");
    elem.addEventListener("click", showMenu);
  }
  function showMenu() {
    var elem = document.getElementById("mainmenu");
    if (!visible) {
      elem.style.display = "block";
      visible = true;
    } else {
      elem.style.display = "none";
      visible = false;
    }
  }
  window.addEventListener("load", initiate);
</script>
```

As we will see later, one way to insert JavaScript code into an HTML document is with the **<script>** element. This element is usually placed inside the document's head (the **<head>** element), but it can also go anywhere we want in the document.

JavaScript code, like any other programming code, is composed of a series of instructions that are processed sequentially. The code in Listing 5-32 gets a reference to the **menu-img** element and assigns a handler for the

**click** event to this element. Then, when the element is clicked, it changes the value of the **display** property of the **mainmenu** element depending on the current condition. If the menu is not visible, it makes it visible, and vice versa (We will explain how this code works in Chapter 6). Figure 5-18 shows what we see when the button is clicked.

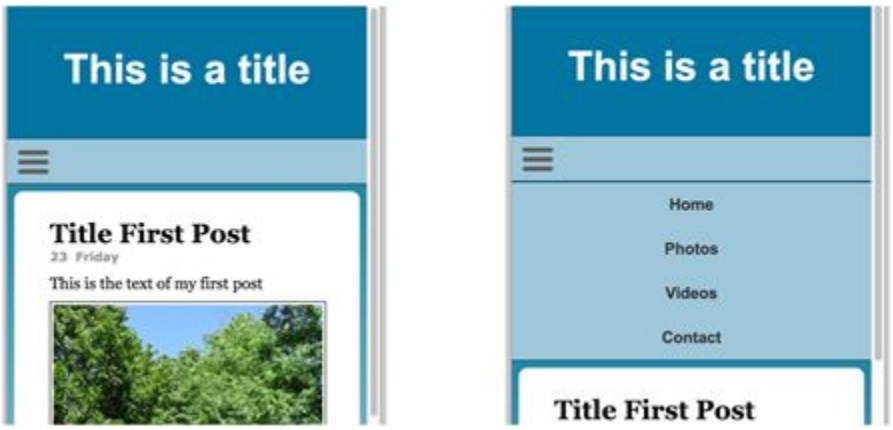


***Figure 5-18:*** *Menu displayed with JavaScript code*

**Do It Yourself:** Add the code in Listing 5-32 inside the **<head>** element and after the **<link>** element in your document. Refresh the page in your browser. Click on the button to open the menu. You should see something similar to Figure 5-18.

Up to now, we have been working with the Traditional Box Model. Although this is currently the preferred model because of its compatibility with old browsers, we also have the option to implement Responsive Web Design with the Flexible Box Model. To illustrate how to develop a responsive website with this model, we are going to use the document introduced in Chapter 4, Listing 4-52.

This example assumes that we have included in the document the **<nav>** element added in Listing 5-29 and the **<script>** element with the JavaScript code introduced in Listing 5-32, but this code has to be modified to make it work with this model. The value assigned to the **display** property to make the menu appear when the button is clicked has to be **flex** instead of **block** because now we are working with flexible containers.

***Listing 5-33:*** *Displaying the menu as a flexible container*

```
<script>
  var visible = false;
  function initiate() {
    var elem = document.getElementById("menu-img");
    elem.addEventListener("click", showMenu);
  }
  function showMenu() {
    var elem = document.getElementById("mainmenu");
    if (!visible) {
      elem.style.display = "flex";
      visible = true;
    } else {
      elem.style.display = "none";
      visible = false;
    }
  }
  window.addEventListener("load", initiate);
</script>
```

The Style Sheet is very similar than the one we have created for the Traditional Box Model; the only difference is that we have to build flexible containers and declare flexibility with the **flex** property. The Following are the styles by default for the whole document.

***Listing 5-34:*** *Designing a responsive document with the Flexible Box Model*

```
* {
  margin: 0px;
  padding: 0px;
}
#logoheader {
  display: flex;
  justify-content: center;
  width: 96%;
  height: 150px;
  padding: 0% 2%;
  background-color: #0F76A0;
```

```
}
#logoheader > div {
  flex: 1;
  max-width: 960px;
  padding-top: 45px;
}
#logoheader h1 {
  font: bold 54px Arial, sans-serif;
  color: #FFFFFF;
}
#mainmenu {
  display: flex;
  justify-content: center;
  width: 96%;
  height: 50px;
  padding: 0% 2%;
  background-color: #9FC8D9;
  border-top: 1px solid #094660;
  border-bottom: 1px solid #094660;
}
#mainmenu > div {
  flex: 1;
  max-width: 960px;
}
#mainmenu li {
  display: inline-block;
  height: 35px;
  padding: 15px 10px 0px 10px;
  margin-right: 5px;
}
#mainmenu li:hover {
  background-color: #6FACC6;
}
#mainmenu a {
  font: bold 18px Arial, sans-serif;
  color: #333333;
  text-decoration: none;
```

```
}
#menuicon {
  display: none;
  width: 95%;
  height: 38px;
  padding: 12px 2% 0px 3%;
  background-color: #9FC8D9;
  border-top: 1px solid #094660;
  border-bottom: 1px solid #094660;
}
main {
  display: flex;
  justify-content: center;
  width: 96%;
  padding: 2%;
  background-image: url("background.png");
}
main > div {
  display: flex;
  flex: 1;
  max-width: 960px;
}
#mainarticles {
  flex: 1;
  margin-right: 20px;
  padding-top: 30px;
  background-color: #FFFFFF;
  border-radius: 10px;
}
#asideinfo {
  flex: 1;
  max-width: 280px;
  padding: 2%;
  background-color: #E7F1F5;
  border-radius: 10px;
}
#asideinfo h1 {
```

```
  font: bold 18px Arial, sans-serif;
  color: #333333;
  margin-bottom: 15px;
}
article {
  position: relative;
  padding: 0px 40px 20px 40px;
}
article time {
  display: block;
  position: absolute;
  top: -5px;
  left: -70px;
  width: 80px;
  padding: 15px 5px;
  background-color: #094660;
  box-shadow: 3px 3px 5px rgba(100, 100, 100, 0.7);
  border-radius: 5px;
}
.daynumber {
  font: bold 36px Verdana, sans-serif;
  color: #FFFFFF;
  text-align: center;
}
.dayname {
  font: 12px Verdana, sans-serif;
  color: #FFFFFF;
  text-align: center;
}
article h1 {
  margin-bottom: 5px;
  font: bold 30px Georgia, sans-serif;
}
article p {
  font: 18px Georgia, sans-serif;
}
figure {
```

```
  margin: 10px 0px;
}
figure img {
  max-width: 98%;
  padding: 1%;
  border: 1px solid;
}
#logofooter {
  display: flex;
  justify-content: center;
  width: 96%;
  padding: 2%;
  background-color: #0F76A0;
}
#logofooter > div {
  display: flex;
  flex: 1;
  max-width: 960px;
  background-color: #9FC8D9;
  border-radius: 10px;
}
.footersection {
  flex: 1;
  padding: 3%;
}
.footersection h1 {
  font: bold 20px Arial, sans-serif;
}
.footersection p {
  margin-top: 5px;
}
.footersection a {
  font: bold 16px Arial, sans-serif;
  color: #666666;
  text-decoration: none;
}
```

The graphic design for this document is the same we have created before, so we have to use the same breakpoints. Again, when the viewport's width is 1120 pixels or less, we have to move the **<time>** element below the article's title. Because in both models the **<time>** element is positioned with absolute values, this Media Query does not change.

***Listing 5-35:*** *Moving the* <time> *element*

```
@media (max-width: 1120px) {
  article time {
    position: static;
    width: 100%;
    padding: 0px;
    margin-bottom: 10px;

    background-color: #FFFFFF;
    box-shadow: 0px 0px 0px;
    border-radius: 0px;
  }
  .daynumber {
    display: inline-block;
    font: bold 14px Verdana, sans-serif;
    color: #999999;
    padding-right: 5px;
  }
  .dayname {
    display: inline-block;
    font: bold 14px Verdana, sans-serif;
    color: #999999;
  }
  article h1 {
    margin-bottom: 0px;
  }
}
```

Now, we have to turn the two-column design into a one-column design when the viewport's width is 720 pixels or less. Because we no longer want the columns to share the same line but to be placed one on top of another,

we have to declare the container as a Block element. Once this is done, extending the elements to the sides is easy, we have to give them a size of 100% (notice that the **<aside>** element has a maximum width of 280 pixels by default, so we also have to declare the value of the **max-width** property as 100% to extend the elements to the edges of the container).

***Listing 5-36:*** *Moving from a two-column to a one-column design*

```
@media (max-width: 720px) {
  main > div {
    display: block;
  }
  #mainarticles {
    width: 100%;
    margin-right: 0px;
  }
  #asideinfo {
    width: 90%;
    max-width: 100%;
    padding: 5%;
    margin-top: 20px;
  }
}
```

At the last breakpoint, we have to modify the menu bar to show the menu button instead of the options and declare the container in the footer as a Block element to move one section on top of another.

***Listing 5-37:*** *Adapting the menu and the footer*

```
@media (max-width: 480px) {
  #logoheader > div {
    text-align: center;
  }
  #logoheader h1 {
    font: bold 46px Arial, sans-serif;
  }
  #mainmenu {
    display: none;
```

```
    width: 100%;
    height: 100%;
    padding: 0%;
  }
  #mainmenu li {
    display: block;
    margin-right: 0px;
    text-align: center;
  }
  #menuicon {
    display: block;
  }
  #logofooter > div {
    display: block;
  }
  .footersection {
    width: 94%;
    text-align: center;
  }
}
```

With the code in Listing 5-37, the Style Sheet is ready. The final design is exactly the same as the one achieved with the Traditional Box Model, but this time using the Flexible Box Model. The Flexible Box Model is a vast improvement over the Traditional Box Model and can simplify the creation of responsive websites, allowing us to modify the order in which the elements are presented and facilitating the combination of flexible and fixed elements, but it is not supported by all the browsers in the market. Some developers are already using this model or implementing some of its properties, but most websites are still developed with the Traditional Box Model.

**Do It Yourself:** Create a new HTML file with the document of Listing 4-52 (see Flexible Box Model in Chapter 4). Add the **<nav>** element introduced in Listing 5-29 and the **<script>** element in Listing 5-33 to the document as explained in the previous example. Create a new CSS file called mystyles.css and copy inside the codes in Listings 5-34, 5-35,

5-36 and 5-37. Download the files myimage.jpg and iconmenu.png from our website and move them to the document's folder. Open the document in your browser and change the size of the window to see how the elements adapt to the space available.

# Chapter 6 - JavaScript

## 6.1 Introduction to JavaScript

HTML and CSS are languages that provide instructions to tell the browser how to organize and visualize a document and its content, but the interaction with the user and the system is limited to a small set of predefined responses. We can create a form with input fields and buttons, but HTML only provides functionality to send the information entered by the user to the server or clear the form. Something similar happens with CSS; we can build instructions (rules) with pseudo-classes like **:hover** to apply a different set of properties when the user moves the mouse over an element, but if we want to perform a custom task, such as modifying the styles of multiple elements at once, we have to load a new Style Sheet that already presents those changes. With the purpose of modifying elements dynamically, performing custom operations, or responding to user interaction, browsers include a third language called *JavaScript*.

JavaScript is a programming language used to process information and manipulate documents. Like any other programming language, JavaScript provides instructions that are executed in sequential order to tell the system what to do (perform an arithmetic operation, assign a new value to an element, etc.). When the browser finds this type of code in our document, it executes the instructions at once, and any changes performed to the document are shown on the screen.

## Implementing JavaScript

Following the same approach than CSS, JavaScript code may be incorporated into the document with three different techniques; the code may be inserted into an element (Inline), embedded in the document, or loaded from external files. The Inline technique takes advantage of special attributes that describe an event, such as the click of the mouse. To make an element respond to an event using this technique, all we have to do is to add the corresponding attribute with the code to handle it.

***Listing 6-1:*** *Defining inline JavaScript*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
</head>
<body>
  <section>
    <p onclick="alert('You clicked me!')">Click Me</p>
    <p>You Can't Click Me</p>
  </section>
</body>
</html>
```

The **onclick** attribute added to the **<p>** element in Listing 6-1 is saying something like "when someone clicks on this element execute this code", and the code is (in this case) the instruction **alert()**. This is a predefined JavaScript instruction called *function*. What this function does is to ask the browser to display a popup window with the value provided between parentheses. When the user clicks on the area occupied by the **<p>** element, the browser executes the **alert()** function and shows a popup window on the screen with the message "You clicked me!".


***Figure 6-1:*** *Popup window generated by the* alert() *function*

**Do It Yourself:** Create a new HTML file with the code in Listing 6-1. Open the document in your browser and click on the text "Click Me" (the **onclick** attribute affects the entire element, not only the text, so you can also click on the rest of the area occupied by the element to execute the code). You should see a popup window with the message "You clicked me!", as shown in Figure 6-1.

**The Basics:** JavaScript includes multiple predefined functions, and you can also create your own. We will learn more about functions and predefined functions later in this chapter.

The **onclick** attribute is one of a series of attributes provided by HTML to respond to events. The list of attributes available is extensive, but they may be organized into groups depending on their purpose. For instance, the following are the most frequently used attributes associated with the mouse.

**onclick**—This attribute handles the **click** event. This event is fired when the user clicks the mouse's left button. HTML offers two more attributes called **ondblclick** (the user double clicks the mouse's left button) and **oncontextmenu** (the user clicks the mouse's right button).

**onmousedown**—This attribute handles the **mousedown** event. This event is fired when the user pushes down the mouse's left or right buttons.

**onmouseup**—This attribute handles the **mouseup** event. This event is fired when the user releases the mouse's left button.

**onmouseenter**—This attribute handles the **mouseenter** event. This event is fired when the mouse enters the area occupied by the element.

**onmouseleave**—This attribute handles the **mouseleave** event. This event is fired when the mouse leaves the area occupied by the element.

**onmouseover**—This attribute handles the **mouseover** event. This event is fired when the mouse moves over the element or any of its children.

**onmouseout**—This attribute handles the **mouseout** event. This event is fired when the mouse moves off the element or off any of its children.

**onmousemove**—This attribute handles the **mousemove** event. This event is fired every time the mouse is over the element and it moves.

**onwheel**—This attribute handles the **wheel** event. This event is fired every time the mouse's wheel is rotated.

The following are the attributes available to respond to events coming from the keyboard. They apply to elements that receive input from the user,

such as the **<input>** and **<textarea>** elements.

**onkeypress**—This attribute handles the **keypress** event. This event is fired when the element is focused, and a key is pressed.

**onkeydown**—This attribute handles the **keydown** event. This event is fired when the element is focused, and a key is pressed down.

**onkeyup**—This attribute handles the **keyup** event. This event is fired when the element is focused, and a key is released.

There are also two important attributes associated with the document.

**onload**—This attribute handles the **load** event. This event is fired when the document or a resource has finished loading.

**onunload**—This attribute handles the **unload** event. This event is fired when a resource has been unloaded.

Event attributes are included in an element depending on when we want the code to be executed. If we want to respond to the click of the mouse, we have to include the **onclick** attribute, as we did in Listing 6-1, but if we want to perform a task when the mouse passes over an element, we have to include the **onmouseover** or **onmousemove** attributes. Because multiple events may occur, in some cases even at the same time, we can declare more than one attribute per element. For instance, the following document includes a **<p>** element with two attributes, **onclick** and **onmouseout**, that include their own JavaScript code. If the user clicks on the element, a popup window is displayed with the message "You clicked me!", but if the user moves the mouse outside the area occupied by the element a different popup window is shown with the message "Do not leave me!"

***Listing 6-2:*** *Implementing multiple event attributes*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
</head>
```

```
<body>
  <section>
    <p onclick="alert('You clicked me!')" onmouseout="alert('Do not leave me!')">Click Me</p>
  </section>
</body>
</html>
```

**Do It Yourself:** Update your HTML file with the code in Listing 6-2. Open the document in your browser and move the mouse over the area occupied by the **<p>** element. If you move the mouse out of the area, you should see a popup window with the message "Do not leave me!".

Events are produced not only by the user but also by the browser. A useful event fired by the browser is **load**. This event is fired when resources have finished loading, and therefore it is frequently used to execute JavaScript code after the document and its content have been loaded.

***Listing 6-3:*** *Responding to the* load *event*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
</head>
<body onload="alert('Welcome!')">
  <section>
    <h1>My Website</h1>
    <p>Welcome to my website</p>
  </section>
</body>
</html>
```

The document of Listing 6-3 shows a popup window to welcome the user after it is completely loaded. The browser first loads the content of the document, and when it is over, it calls the **alert()** function and shows the message on the screen.

**IMPORTANT:** Events are critical in web development. Besides those studied in this chapter, there are dozens more available to control a variety of processes, from playing a video to checking the progress of a task. We will learn more about events later in this chapter, and study the rest of the events available in more practical situations in later chapters.

**The Basics:** When you test the code in Listing 6-3 in your browser, you will see that the popup window is shown before the content of the document appears on the screen. This is because the document is loaded in an internal structure of objects called DOM and later reconstructed on the screen from these objects. We will learn more about the DOM and how to access HTML elements from JavaScript later in this chapter.

Event attributes are useful when we want to test code or implement a feature quickly, but they are not suitable for large applications. To work with extensive codes and customized functions, we have to group the codes together with the **<script>** element. The **<script>** element acts exactly like the **<style>** element for CSS, organizing the code in one place and affecting the rest of the elements in the document using references.

***Listing 6-4:*** *Embedding JavaScript in the document*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    alert('Welcome!');
  </script>
</head>
<body>
  <section>
    <p>Hello</p>
  </section>
</body>
</html>
```

The **<script>** element and its content may be located anywhere in the document, but it is usually placed inside the header, as in this example. This way, when the browser loads the file, it reads the content of the **<script>** element, it executes the code right away, and then continues processing the rest of the document.

**Do It Yourself:** Update your HTML file with the code in Listing 6-4 and open the document in your browser. You should see a popup window with the message "Welcome!" as soon as the document is loaded. Because the **alert()** function stops the execution of the code, the content of the document is not shown on the screen until we press the OK button.

Embedding JavaScript in the document with the **<script>** element may be practical when we have a small set of instructions, but JavaScript code can grow very fast. If we use the same code in more than one document, we will have to maintain different versions of the same program and the browsers will have to download the same code again with each document requested by the user. A better alternative is to include the JavaScript code in a single file and then load that file from the documents that require it. This way, only the documents that need that set of instructions will include the file, and the browser will have to download the file only once (browsers keep the files in a cache in case they are later requested by other documents from the same website). For this purpose, the **<script>** element includes the **src** attribute. With this attribute, we can declare the path to the JavaScript file and write all our code inside that file. The **<script>** element in the following example loads the JavaScript code from an external file called mycode.js.

***Listing 6-5:*** Introducing JavaScript code from *an external file*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script src="mycode.js"></script>
</head>
```

```
<body>
  <section>
    <p>Hello</p>
  </section>
</body>
</html>
```

JavaScript files, like HTML and CSS files, are text files and therefore they are created with a text editor or the professional editors we have recommended in Chapter 1. We may assign to these files any name we want, but by convention, they have to include the .js extension. Inside, the code must be declared exactly as we would do between the **<script>** tags. For instance, the following is the code we have to introduce in the mycode.js file to reproduce the previous example.

***Listing 6-6:*** Creating a JavaScript file (mycode.js)

```
alert("Welcome!");
```

**Do It Yourself:** Update your HTML file with the code in Listing 6-5. Create a new file with the name mycode.js and the JavaScript code in Listing 6-6. Open the document in your browser. You should see a popup window with the message "Welcome!" as soon as the document is loaded.

**IMPORTANT:** Besides the **src** attribute, there are two more attributes available for the **<script>** element: **async** and **defer**. These are Boolean attributes that indicate how and when the code will be executed. If **async** is declared, the code is executed asynchronously (while the document is being processed). If the **defer** attribute is declared, the code is executed after the document has been processed.

**The Basics:** It is recommended to finish each line of code with a semicolon to make sure that the browser does not have any trouble identifying the end of the instruction.

# Variables

Of course, JavaScript is more than popup windows showing messages to alert the user. The language can perform numerous tasks, from calculating complex algorithms to processing the content of a document. Each one of these tasks involves the manipulation of values, and this is why the most important feature of JavaScript, as in any other programming language, is the capacity to store data in memory.

The memory of a computer or mobile device is like a honeycomb with millions and millions of consecutive cells available to store information. These cells have limited space, and therefore the combination of several cells is necessary to store large amounts of data. Because of the complexity of this structure, programming languages incorporate the concept of *variables* to facilitate the identification of each value stored in memory. Variables are names assigned to a cell or a group of cells where the data is going to be stored. For example, if we want to store the value 5 in memory, we need to know where that number was stored to be able to retrieve it later. Creating a variable allows us to identify that space in memory with a name and use that name later to read the value or store a new one.

Variables in JavaScript are declared with the **var** keyword followed by the name we want to give to the variable. If we want to store a value in the space of memory assigned by the system to the variable, we have to include the character = (equal) followed by the value, as in the following example.

***Listing 6-7:*** *Declaring a variable in JavaScript*

```
var mynumber = 2;
```

The instruction in Listing 6-7 creates the variable **mynumber** and stores the value **2** in the space of memory reserved by the system for this variable. When this code is executed, the browser reserves a space in memory, stores the number **2** inside, creates a reference to that space, and finally assigns that reference to the name **mynumber**.

After the value is assigned to the variable, every time the variable is referenced (the name **mynumber** is mentioned), the system reads the memory and returns the number **2**, as illustrated by the following example.

***Listing 6-8:*** *Using the content of a variable*

```
var mynumber = 2;
alert(mynumber);
```

As we already mentioned, the instructions in a JavaScript program are executed by the browser one by one in sequential order. So when the browser reads the code in Listing 6-8, it executes the instructions from top to bottom. The first instruction asks the browser to create a variable called **mynumber** and assign the value **2** to it. After this task is completed, the browser executes the next instruction on the list. This instruction asks the browser to display a popup window with the current value stored in the **mynumber** variable.


***Figure 6-2:*** *Popup window displaying the value of a variable*

**Do It Yourself:** Update your mycode.js file with the code in Listing 6-8. Open the document of Listing 6-5 in your browser. You should see a popup window with the value **2**.

Variables are called variables because they are not constant. We can change their values any time we want, and that is, in fact, their most important characteristic.

***Listing 6-9:*** *Assigning a new value to a variable*

```
var mynumber = 2;
mynumber = 3;
alert(mynumber);
```

In Listing 6-9, after the variable is declared, a new value is assigned to it. Now the **alert()** function shows the number **3** (the value **2** was replaced with the value **3** by the second instruction). There is no need for the use of the **var** keyword this second time, just the name of the variable and the = sign are required to assign the new value.

The value stored in a variable may be assigned to another variable. For instance, the following code creates two variables called **mynumber** and

**yournumber** and assigns the value stored in the **mynumber** variable to the **yournumber** variable. The value shown by the popup window is the number **2**.

***Listing 6-10:*** *Assigning the value of a variable to another variable*

```
var mynumber = 2;
var yournumber = mynumber;
alert(yournumber);
```

In a more practical situation, we would probably use the value of the variable to perform an operation and assign the result back to the same variable.

***Listing 6-11:*** *Performing an operation with the value stored in a variable*

```
var mynumber = 2;
mynumber = mynumber + 1;  // 3
alert(mynumber);
```

In this example, a value of **1** is added to the current value of **mynumber** and the result is assigned to the same variable. This is the same as performing 2 + 1, with the difference that when using a variable instead of a number its value may change at any time.

**The Basics:** The characters at the end of the second instruction are considered to be a comment and therefore they are not processed as part of the instruction. Comments are added to the code as references or reminders for the developer. They may be declared using two slashes for single-line comments (// **comment**) or combining a slash and a star character for multi-line comments (/* **comment** */). Everything after the two slashes or in between the /* and */ characters is ignored by the browser.

Besides the **+** operator, JavaScript also includes the operators **-** (subtraction), ***** (multiplication), / (division), and **%** (remainder). These operators may be used in a single operation between two values or combined with multiple values to perform more complex arithmetic operations.

***Listing 6-12:*** *Performing complex operations*

```
var mynumber = 2;
mynumber = mynumber * 25 + 3;  // 53
alert(mynumber);
```

Arithmetic operations are executed following an order of precedence determined by the operators. Multiplication and division have precedence over addition and subtraction. This means that the multiplications and divisions are performed before the additions and subtractions. In Listing 6-12, the current value of the **mynumber** variable is multiplied by 25, and then the value 3 is added to the result. If we want to control the precedence, we can isolate the operations between parentheses. For example, the following code performs the addition and then the multiplication, producing a different result.

***Listing 6-13:*** *Controlling precedence in the operation*

```
var mynumber = 2;
mynumber = mynumber * (25 + 3);  // 56
alert(mynumber);
```

Performing an operation on the variable's current value and assigning the result back to the same variable is very common in computer programming. JavaScript offers the following operators to simplify this task.

- ++ is shorthand for **variable = variable + 1**.
- **--** is shorthand for **variable = variable - 1**.
- += is shorthand for **variable = variable + number**.
- **-=** is shorthand for **variable = variable - number**.
- ***=** is shorthand for **variable = variable * number**.
- **/=** is shorthand for **variable = variable / number**.

With these operators, we can easily perform operations on the value of a variable and assign the result back to the same variable. A common use of these operators is to create counters by incrementing or decrementing the value of a variable in one or more units. For example, the ++ operator adds

the value 1 to the variable's current value every time the instruction is executed.

***Listing 6-14:*** *Incrementing the value of a variable*

```
var mynumber = 0;
mynumber++;
alert(mynumber);  // 1
```

If the value of the variable must be incremented by more than one unit, we can use the += operator. This operator adds the value specified in the instruction to the variable's current value and stores the result back in the variable.

***Listing 6-15:*** *Incrementing the value of a variable by a specific value*

```
var mynumber = 0;
mynumber += 5;
alert(mynumber);  // 5
```

The process generated by the code in Listing 6-15 is straightforward. After the value 0 is assigned to the **mynumber** variable, the system reads the second instruction, gets the variable's current value, adds 5 to that value, and stores the result back in **mynumber**.

An interesting operator is the remainder operator. This operator returns the amount left by a division between two numbers.

***Listing 6-16:*** *Calculating the remainder*

```
var mynumber = 11 % 3;  // 2
alert(mynumber);
```

The operation assigned to the **mynumber** variable in Listing 6-16 produces the result 2. The system divides 11 by 3 and finds a quotient of 3. Then, to get the remainder, it calculates 11 minus the multiplication of 3 times the quotient (11 - (3 * 3) = 2).

The remainder operator is used frequently to determine whether a value is odd or even. If we calculate the remainder of an integer number divided by 2, we get a result according to its parity (odd or even). If the number is

even, the remainder is 0, but if the number is odd, the remainder is 1 (or -1 for negative numbers).

***Listing 6-17:*** *Determining the parity of a number*

```
var mynumber = 11;
alert(mynumber % 2);   // 1
```

The code in Listing 6-17 calculates the remainder of the current value of the **mynumber** variable. The value returned is 1, which means that the value is odd. Notice that this time we perform the operation between the parentheses of the **alert()** function. Every time an operation is included inside an instruction, the browser first performs the operation and then executes the instruction with the result, so an operation may be provided every time a value is required.

> **Do It Yourself:** Update your mycode.js with the example you want to try and open the document in your browser. Change the numbers assigned to the variables and perform more complex operations to see the different results produced by JavaScript and get familiar with the language's operators.

# Strings

In previous examples, we stored numbers, but variables can also store other types of values, including text. To assign text to a variable, we have to enclose the characters in quotation marks (single or double quotes).

***Listing 6-18:*** *Assigning a string of characters to a variable*

```
var mytext = "Hello World!";
alert(mytext);
```

The code in Listing 6-18 creates a variable called **mytext** and stores a string of characters in it. When the first instruction is executed, the system reserves a space in memory long enough to store the string, creates the variable, and stores the text. Every time we read the **mytext** variable, we get back the text "Hello World!" (without the quotes).


***Figure 6-3:*** *Popup window displaying a string stored in a variable*

The space reserved in memory by the system to store the string of characters depends on the size of the text (how many characters it contains), but the system is prepared to extend that space if longer values are assigned later to the variable. A common situation in which longer texts are assigned to a variable is when we add more characters at the beginning or the end of the variable's current value. The texts may be concatenated with the + operator.

***Listing 6-19:*** *Concatenating text*

```
var mytext = "My name is ";
mytext = mytext + "John";
alert(mytext);
```

The code in Listing 6-19 adds the text "John" at the end of the text "My name is ". The final value of the **mytext** variable is "My name is John". If we want to add the text at the beginning, we have to invert the operation.

***Listing 6-20:*** *Adding text at the beginning of the value*

```
var mytext = "John";
mytext = "My name is " + mytext;
alert(mytext);
```

If instead of text we try to concatenate a string of text and a number, the number is converted into a string and added to the current value. The following code produces the string "The number is 3".

***Listing 6-21:*** *Concatenating strings and numbers*

```
var mytext = "The number is " + 3;
alert(mytext);
```

This is important when we have a string that contains a number, and we want to add another number to it. Because JavaScript considers the current value to be a string, the number is turned into a string, and the values are not added but concatenated.

***Listing 6-22:*** *Concatenating numbers*

```
var mytext = "20" + 3;
alert(mytext);  // "203"
```

> **The Basics:** The result of the code in Listing 6-22 is not 23 but the string "203". The system converts the number 3 into a string and concatenates the strings instead of adding the numbers. Later, we will learn how to extract numbers from strings to perform operations with their values.

Strings may contain any character we want, and this includes single and double quotes. If the quotes included in the text are different from those used to define the string, the quotes are treated like any other character, but if the quotes are the same, the system does not know where the string ends. To solve this problem, JavaScript provides the escape character \. For example, if the string was declared with single quotes, we have to escape the single quotes inside the text.

***Listing 6-23:*** *Escaping characters*

```
var mytext = 'They don\'t make it anymore';
alert(mytext);  // "They don't make it anymore"
```

JavaScript offers several escape characters for different purposes. The most frequently used are **\n** to generate a new line and **\r** to return the cursor to the beginning of the line. These two characters are usually implemented together to divide the text into multiple lines, as in the following example.

***Listing 6-24:*** *Generating new lines*

```
var mytext = "The good life is one inspired by love\r\n";
mytext = mytext + "and guided by knowledge"
alert(mytext);
```

The code in Listing 6-24 begins by assigning a string to the **mytext** variable that includes the New Line and Carriage Return characters. In the next instruction, we add a second text at the end of the variable's current value, but because of the escape characters, the strings are displayed in the popup window in two lines.

**The Basics:** Strings are declared as objects in JavaScript, and therefore they include methods to perform operations on their characters. We will study objects, **String** objects, and how to implement their methods later in this chapter.

# Booleans

Another type of values we can store in variables are Booleans. There are only two Boolean values available: **true** and **false**. They are useful when we only need to determine the current state of a condition. For example, if our application needs to know whether a value inserted in a form is valid or not, we can keep track of this condition with a Boolean variable.

***Listing 6-25:*** *Declaring a Boolean variable*

```
var valid = true;
alert(valid);
```

The purpose of these variables is to simplify the process of identifying the state of a condition. If we use an integer value to indicate a state, we have to remember which numbers we decided to use for the valid and the invalid states. Using Boolean values, we only have to check whether the value is equal to **true** or **false**.

**IMPORTANT:** Booleans become useful when we use them with instructions that allow us to perform a task or repetitive tasks depending on a condition. We will study Conditionals and Loops later in this chapter.

# Arrays

Variables can also store several values at once in a structure called *arrays*. Arrays may be created using a simple notation that includes the values separated by commas and between square brackets. The values are later identified by an index, starting from 0.

***Listing 6-26:*** *Creating arrays*

```
var myarray = ["red", "green", "blue"];
alert(myarray[0]);  // "red"
```

In Listing 6-26, we create an array called **myarray** with three values, the strings "red", "green" and "blue". JavaScript automatically assigns the index **0** to the first value, **1** to the second value, and **2** to the third value. To retrieve this information, we have to mention the index every time we want to read each value in the array. This is done by writing the index number within square brackets and after the name of the variable. For example, to get the first value of **myarray**, we have to write the instruction **myarray[0]**, as we did in our example.

Notice that the **alert()** function can display not only single values but also entire arrays. If we want to see all the values included in an array, we just have to specify the array's name.

***Listing 6-27:*** *Displaying the array's values*

```
var myarray = ["red", "green", "blue"];
alert(myarray);  // "red,green,blue"
```

Arrays, like common variables, can take any type of value we want. For example, we can create an array like the one in Listing 6-27 combining numbers and strings.

***Listing 6-28:*** *Using different types of values*

```
var myarray = ["red", 32, "HTML5 is awesome!"];
alert(myarray[1]);  // 32
```

**Do It Yourself:** Replace the code in your mycode.js file with the code in Listing 6-28 and open the document of Listing 6-5 in your browser. Change the index provided to the **alert()** function to see every value of the array (remember that the indexes start from 0).

If we try to read a value in an index that was not yet defined, JavaScript returns the value **undefined**. This value is used by the system to report that the value we are trying to access does not exist, but we can also assign it to an array to declare that we do not have the value for that position.

***Listing 6-29:*** *Declaring a value as undefined*

```
var myarray = ["red", undefined, 32];
alert(myarray[1]);  // undefined
```

A better way to tell the system that there is no value available in a certain position of the array is using another special value called **null**. The difference between the values **undefined** and **null** is that **undefined** indicates that the variable was declared but no value was assigned to it, while **null** indicates that there is a value but it is null.

***Listing 6-30:*** *Declaring a* null *value*

```
var myarray = ["red", 32, null];
alert(myarray[2]);  // null
```

Of course, we can also perform operations on the values of an array and store the result, as we did before with single variables.

***Listing 6-31:*** *Working with values in arrays*

```
var myarray = [64, 32];
myarray[1] = myarray[1] + 10;
alert("The new value is " + myarray[1]);  // "The new value is 42"
```

Arrays operate exactly like single variables, with the exception that we have to mention the index every time we want to use them. With the instruction **myarray[1] = myarray[1] + 10** we tell the JavaScript interpreter to read the current value of **myarray** at index **1** (32), add 10 to

that value, and store the result in the same array at the same index. In the end, the value of **myarray[1]** is 42.

Arrays can take any type of values, so it is possible to declare arrays of arrays. These types of arrays are called *multidimensional arrays*.

***Listing 6-32:*** *Defining multidimensional arrays*

```
var myarray = [[2, 45, 31], [5, 10], [81, 12]];
```

The code in Listing 6-32 creates an array of arrays of integers (notice that the values are declared inside another array). To access these values, we have to declare the indexes of each level between square brackets, one after another. The following example returns the first value (index 0) of the second array (index 1). The instruction reads the array at index 1 and then reads the number in that array at index 0.

***Listing 6-33:*** *Accessing multidimensional arrays*

```
var myarray = [[2, 45, 31], [5, 10], [81, 12]];
alert(myarray[1][0]);  // 5
```

If we want to erase one of the values, we can declare it as **undefined** or **null**, as we did before, or declare it as an empty array assigning square brackets with no values inside.

***Listing 6-34:*** *Assigning an empty array as the value of another array*

```
var myarray = [[2, 45, 31], [5, 10], [81, 12]];
myarray[1] = []
alert(myarray[1][0]);  // undefined
```

Now, the value shown in the popup window is **undefined**, because there is no element at the positions **[1][0]**. Of course, this may also be used to empty any type of array.

***Listing 6-35:*** *Assigning an empty array to a variable*

```
var myarray = [2, 45, 31];
myarray = []
alert(myarray[1]);  // undefined
```

**The Basics:** Like strings, Arrays are also declared as objects in JavaScript, and therefore they include methods to perform operations on their values. We will study objects, Array objects, and how to implement their methods later in this chapter.

# Conditionals and Loops

Up to this point, we have been writing instructions in a sequence, one after another. In this kind of program, the system executes each instruction once, and in the order they were presented. It begins with the first one and goes on until it reaches the end of the list. The purpose of Conditionals and Loops is to break this sequential flow. Conditionals allow us to execute one or more instructions only when a certain condition is met, and Loops let us execute a block of code (a group of instructions) over and over again until a condition is met. JavaScript provides a total of four instructions to process code according to conditions determined by the programmer: **if**, **switch**, **for** and **while**.

The simplest way to check a condition is with the **if** instruction. The **if** instruction checks an expression and processes a set of instructions if the condition set by the expression is true. The instruction requires the keyword **if** followed by the condition between parentheses and the instructions we want to execute if the condition is true between braces.

***Listing 6-36:*** *Checking a condition with* if

```
var myvariable = 9;
if (myvariable < 10) {
  alert("The number is less than 10");
}
```

In the code in Listing 6-36, the value 9 is assigned to **myvariable**, and then using the **if** instruction we compare the variable with the number **10**. If the value of the variable is less than **10**, the **alert()** function shows a message on the screen.

The operator used in this example to compare the value of the variable against 10 is called *comparison operator*. The following are the comparison operators available in JavaScript.

- == checks whether the value on the left is equal to the value on the right.
- **!=** checks whether the value on the left is different than the value on the right.
- > checks whether the value on the left is greater than the value on the right.
- < checks whether the value on the left is less than the value on the right.
- >= checks whether the value on the left is greater or equal than the value on the right.
- <= checks whether the value on the left is less or equal than the value on the right.

After a condition is evaluated, it returns a logical value true or false. This allows us to work with conditions as values and combine them to create complex conditions. JavaScript offers the following logical operators for this purpose.

- **!** (logical NOT) toggles the state of the condition. If the condition is true, it returns false, and vice versa.
- **&&** (logical AND) checks two conditions and returns true if both are true.
- || (logical OR) checks two conditions and returns true if one or both are true.

The logical operator **!** inverts the state of the condition. If the condition evaluates to true, the final state will be false, and the instructions between braces will not be executed.

***Listing 6-37:*** *Inverting the result of the condition*

```
var myvariable = 9;
if (!(myvariable < 10)) {
  alert("The number is less than 10");
}
```

The code in Listing 6-37 does not show any message on the screen. The value 9 is still less than 10, but because we alter the condition with the **!** operator, the final result is false, and the **alert()** function is not executed. Notice that we put the condition between parentheses to work with the state returned by the condition and not the values the condition is comparing. Because of the parentheses, the condition is first evaluated, and then the state returned is inverted by the **!** operator.

The **&&** (AND) and || (OR) operators work a little bit different. They calculate the final result based on the results of the conditions involved. The **&&** (AND) operator returns true only if the conditions on both sides are true, and the || (OR) operator returns true if one or both conditions are true. For example, the following code executes the **alert()** function only when the age is less than 21 and the value of the **smart** variable is equal to "YES" (because we use the **&&** operator, both conditions have to be true for the overall condition to be true).

***Listing 6-38:*** *Checking multiple conditions with logical operators*

```
var smart = "YES";
var age = 19;
if (age < 21 && smart == "YES") {
  alert("John is allowed");
}
```

If we assume that our example only considers two values for the **smart** variable, "YES" and "NO", we can turn it into a Boolean variable. Because Booleans are already logical values, we do not need to compare them to anything. The following code reproduces the previous example using a Boolean variable.

***Listing 6-39:*** *Using Boolean values as conditions*

```
var smart = true;
var age = 19;
if (age < 21 && smart) {
  alert("John is allowed");
}
```

JavaScript is very flexible regarding conditions. The language is capable of determining a condition based on the value of any variable. For example, a variable with an integer number will return false if the value is 0 or true if the value is different than 0.

***Listing 6-40:*** *Using integer numbers to set a condition*

```
var age = 0;
if (age) {
  alert("John is allowed");
}
```

The code in the **if** instruction of Listing 6-40 is not executed because the value of the **age** variable is 0 and therefore the state of the condition is considered to be false. If we store a different value into this variable, the condition will be true, and the message will be shown on the screen.

Variables with empty strings also return false. The following example checks whether a string was assigned to a variable and only shows its value if the string is not empty.

***Listing 6-41:*** *Using strings to set a condition*

```
var name = "John";
if (name) {
  alert(name + " is allowed");
}
```

**Do It Yourself:** Replace the code in your mycode.js file with the code in Listing 6-41 and open the document of Listing 6-5 in your browser. The popup window should show the message "John is allowed". Assign an empty string to the **name** variable. The **if** instruction now considers the condition to be false and no message is shown on the screen.

There are times when instructions have to be executed for either state of the condition (true or false). JavaScript includes the **if else** instruction to help in this situation. The instructions to be executed are declared in two blocks of code delimited by braces. The block preceded by **if** is executed when the condition is true and the block preceded by **else** is executed otherwise.

***Listing 6-42:*** *Checking two conditions with* if else

```
var myvariable = 21;
if (myvariable < 10) {
  alert("The number is less than 10");
} else {
  alert("The number is 10 or larger");
}
```

In this example, the code considers two conditions: when the number is less than **10** and when the number is equal to or greater than **10**. If we need to check multiple conditions, instead of the **if else** instructions we can use the **switch** instruction. This instruction evaluates an expression (usually just a variable), compares the result with multiple values, and executes the instructions corresponding to the value that matches the expression. The syntax includes the **switch** keyword followed by the expression between parentheses. The possible values are listed using a keyword called **case**, as in the following example.

***Listing 6-43:*** *Checking a value with the* switch *instruction*

```
var myvariable = 8;
switch(myvariable) {
  case 5:
    alert("The number is five");
    break;
  case 8:
    alert("The number is eight");
    break;
  case 10:
    alert("The number is ten");
    break;
  default:
    alert("The number is " + myvariable);
}
```

In Listing 6-43, the **switch** instruction evaluates the variable **myvariable**, gets its value, and then compares it with the value in every

case. If the value is **5**, for example, the control is transferred to the first **case,** and the **alert()** function shows the text "The number is five" on the screen. If this first **case** does not match the value of the variable, then the next case is evaluated, and so on. If no case matches the value, the instructions in the case labeled **default** are executed.

In JavaScript, once a match is found, the instructions in the case are executed along with the instructions in the following cases. This is the default behavior but usually not what we need in our code. For this reason, JavaScript provides the **break** instruction. To stop the system from executing the instructions of every case after the match, we have to include the **break** instruction at the end.

The **switch** and **if** instructions are useful but perform a simple task; they evaluate an expression, execute a block of instructions according to the result, and return the control to the main code afterward. There are certain situations in which this is not sufficient. We could have to execute the instructions several times for the same condition or evaluate the condition again after a process is over. There are two instructions we can use in these situations: **for** and **while**.

The **for** instruction executes the code between braces while a condition is true. It uses the syntax **for(initialization; condition; increment)**. The first parameter establishes the initial values for the loop; the second parameter is the condition to be checked, and the last parameter is an instruction that determines how the initial values evolve in every cycle.

***Listing 6-44:*** *Creating a loop with the* for *instruction*

```
var total = 0;
for (var f = 0; f < 5; f++) {
  total += 10;
}
alert("The total is: " + total);  // "The total is: 50"
```

In the code in Listing 6-44, we declare a variable called **f** to control the loop and assign the number **0** as its initial value. The condition in this example checks whether the value of the variable **f** is less than 5. If this is true, the code between braces is executed. After that, the JavaScript interpreter executes the last parameter of the **for** instruction, which adds 1 to the current value of **f** (**f++**), and then checks the condition again (in every

cycle **f** is incremented by 1). If the condition is still true, the instructions are executed one more time. This process continues until **f** reaches a value of **5**, which makes the condition false (5 is not less than 5), and the loop is interrupted.

Inside the **for** loop in Listing 6-44, we add the value 10 to the current value of the **total** variable. Loops are frequently used in this manner, to make the value of a variable evolve according to previous results. For example, we can use a **for** loop to add all the values of an array.

***Listing 6-45:*** *Iterating over the values of an array*

```
var total = 0;
var list = [23, 109, 2, 9];
for (var f = 0; f < 4; f++) {
  total += list[f];
}
alert("The total is: " + total);  // "The total is: 143"
```

To read all the values of an array, we have to create a loop that goes from the index of the initial value to a value that matches the index of the last value in the array. In this case, the **list** array contains four elements, and therefore the corresponding indexes go from 0 to 3. The loop reads the value at index 0, adds it to the current value of the **total** variable, and then moves to the next value in the array until the value of **f** is equal to 4 (there is no value at index 4). In the end, all the values of the array are added to the **total** variable, and the result is shown on the screen.

> **The Basics:** In the example of Listing 6-45, we are able to set up the loop because we know the number of values inside the array, but this is not always the case. Sometimes you will not have this information during development, either because the array is created when the page is loaded or with values inserted or selected by the user. To work with dynamic arrays, JavaScript offers the **length** property. The property returns the number of values in an array. We will study this property and **Array** objects later in this chapter.

The **for** instruction is useful when we can determine certain prerequisites, such as the initial values for the loop or how those values will

evolve. When this information is unclear, we can use the **while** instruction. The **while** instruction only requires the declaration of the condition in parentheses and the code to be executed between braces. The loop runs until the condition is false.

***Listing 6-46:*** *Using the* while *instruction*

```
var counter = 0;
while(counter < 100) {
  counter++;
}
alert("The value is: " + counter);  // "The value is: 100"
```

The example of Listing 6-46 is straightforward. The instruction between braces is executed while the value of the **counter** variable is less than 100. This means that the loop will be executed 100 times (when the value of **counter** is 99, the instruction is executed one more time and therefore the variable's final value is 100).

If the first evaluation of the condition returns false (e.g., the value of **counter** is already greater than 99) the code between braces is never executed. If we want the instructions to be executed at least once, regardless of the result of the condition, we can use a different implementation of the **while** loop called **do while**. The **do while** instruction first executes the instructions between braces and then checks the condition, which guarantees that the instructions are going to be executed at least once. The syntax is similar; we have to precede the braces with the **do** keyword and place the **while** keyword with the condition at the end.

***Listing 6-47:*** *Using the* do while *instruction*

```
var counter = 150;
do {
  counter++;
} while(counter < 100);
alert("The value is: " + counter);  // "The value is: 151"
```

In Listing 6-47, the value of the **counter** variable is already greater than 99, but because we are using the **do while** loop, the instruction between

braces is executed once, and therefore the final value of **counter** is 151 (150 + 1 = 151).

# Control Transfer Instructions

Loops sometimes must be interrupted, regardless of the state of the condition. JavaScript offers multiple instructions to break the execution of loops and conditionals. The following are the most frequently used.

**continue**—This instruction interrupts the current cycle and moves to the next. The system ignores the rest of the instructions in the loop after this instruction is executed.

**break**—This instruction interrupts the loop. All the instructions and any pending cycles are ignored after this instruction is executed.

The **continue** instruction is applied when we do not want to execute the rest of the instructions between braces, but we want to keep the loop running.

***Listing 6-48:*** *Jumping to the next cycle of the loop*

```
var list = [2, 4, 6, 8];

var total = 0;
for (var f = 0; f < 4; f++) {
  var number = list[f];
  if (number == 6) {
    continue;
  }
  total += number;
}
alert("The total is: " + total);  // "The total is: 14"
```

The **if** instruction inside the **for** loop in Listing 6-48 compares the value of **number** with the value 6. If the number currently produced by the loop is 6, the **continue** instruction is executed, the last instruction inside the loop is ignored, and the loop moves on to the next value in the

**list** array. In consequence, all the values of the array are added to the **total** variable except the number 6.

Unlike **continue**, the **break** instruction interrupts the loop completely, delegating the control to the instruction after the loop.

***Listing 6-49:*** *Interrupting the loop*

```
var list = [2, 4, 6, 8];

var total = 0;
for (var f = 0; f < 4; f++) {
  var number = list[f];
  if (number == 6) {
    break;
  }
  total += number;
}
alert("The total is: " + total);  // "The total is: 6"
```

Again, the **if** instruction in Listing 6-49 compares the value of **number** with the value 6, but this time it executes the **break** instruction when a match is found. If the number currently produced by the loop is 6, the **break** instruction is executed, and the loop is over, no matter how many values are left in the array. In consequence, only the values located before the number 6 are added to the value of the **total** variable.

# 6.2 Functions

Functions are blocks of code identified with a name. The difference between functions and the blocks of code used with loops and conditionals discussed earlier is that there is no condition to satisfy; the instructions inside a function are executed every time the function is called. Functions are called (executed) by writing the name followed by parentheses. This call may be performed from anywhere in the code and every time necessary, which completely breaks the sequential processing of a program. Once a function is called, the execution of the program continues with the instructions inside the function and only returns back to the section of the code that called the function when the execution of the function is over.

# Declaring Functions

Functions are declared using the keyword **function**, the name followed by parentheses, and the code between braces. To call a function (execute it), we have to declare its name with a pair of parentheses at the end, as shown below.

***Listing 6-50:*** *Declaring and executing functions*

```
function showMessage() {
  alert("I am a function");
}
showMessage();
```

Functions are declared first and then executed. The code in Listing 6-50 declares a function called **showMessage()** and then calls it one time. As with variables, the JavaScript interpreter reads the function, stores its content in memory, and assigns a reference to the name of the function. When we call the function by its name, the interpreter checks the reference and reads the function from memory. This allows us to call the function as many times as we want, as in the following example.

***Listing 6-51:*** *Processing data with functions*

```
var total = 5;
```

```
function calculateValues(){
  total = total * 2;
}
for(var f = 0; f < 10; f++){
  calculateValues();
}
alert("The total is: " + total);  // "The total is: 5120"
```

The code in Listing 6-51 combines different instructions already studied. First, it declares a variable and assigns the value **5** to it. Then, a function called **calculateValues()** is declared (but not executed). Next, a **for** instruction is used to create a loop that will run as long as the value of the variable **f** is less than 10. The instruction inside the loop calls the **calculateValues()** function, so the function is executed in every cycle. Every time the function is executed, the value of **total** is multiplied by 2, doubling the current value.

# Scope

In JavaScript, the instructions that are outside of a function are considered to be in a global namespace. This is the space where we write the instructions until a function, or another data structure is defined. Variables defined in the global namespace have global scope, and therefore they can be used anywhere in the code, but those declared inside functions have local scope, which means that they can only be used inside the function in which they were declared. This is another advantage of functions; they are special places where we can store information that will not be accessible from other parts of the code. This segregation helps us avoid duplicates that may lead to mistakes, such as overwriting the value of a variable when the old value was still required by the application.

The following example illustrates how the scopes are defined and what we should expect when we access the variables of one scope from another.

***Listing 6-52:*** *Declaring global and local variables*

```
var globalVariable = 5;
function myfunction(){
```

```
  var localVariable = "The value is ";
  alert(localVariable + globalVariable);  // "The value is 5"
}
myfunction();
alert(localVariable);
```

The code in Listing 6-52 declares a function called **myfunction()** and two variable, one in the global namespace called **globalVariable** and another inside the function called **localVariable**. When the function is executed, the code concatenates the variables **localVariable** and **globalVariable** and shows the final string on the screen. Because **globalVariable** is a global variable, it is accessible from inside the function and therefore its value is added to the string. But when we try to display the value of **localVariable** outside the function, an error is returned (the popup window is not shown). This is because **localVariable** was defined inside the function and therefore it is not accessible from the global namespace.

> **The Basics:** Browsers report JavaScript errors in a hidden console. If you want to see the errors produced by your code, you have to open this console from the options in the browser's menu (Developer Tools in Google Chrome). We will learn more about consoles and how to manage errors at the end of this chapter.

Because variables with different scopes are considered to be different variables, two variables with the same name, one defined in the global namespace and another inside a function, will be considered two separate variables (they are assigned different spaces in memory).

***Listing 6-53:*** *Declaring two variables with the same name*

```
var myvariable = 5;
function myfunction(){
  var myvariable = "This is a local variable";
  alert(myvariable);
}
myfunction();
alert(myvariable);
```

In the code in Listing 6-53, we declare two variables called **myvariable**, one in the global namespace and another inside the function **myfunction()**. There are also two **alert()** functions in the code, one inside **myfunction()** and another in the global namespace at the end of the code. Both **alert()** functions show the content of **myvariable**, but they reference different variables. The variable inside the function is referencing a space in memory that contains the string "This is a local variable", while the variable in the global namespace is referencing a space in memory containing the value 5.

**The Basics:** Global variables can also be created from functions. Omitting the **var** keyword when a variable is declared inside a function will be enough to set that variable as global.

Global variables are useful when functions have to share common values, but because they are accessible from every function in the code, there is always the possibility of accidentally overwriting their values from somewhere else in the same code, or even from other codes (every JavaScript code included in the document shares the same global namespace). Therefore, using global variables from functions is not a good idea. As an alternative, JavaScript allows us to send values to a function when the function is called.

To receive a value, the function must include a name between the parentheses. These names are called *parameters*. When the function is executed, these parameters are turned into variables that we can read inside the function to access the values.

***Listing 6-54:*** *Sending a value to a function*

```
function myfunction(value) {
  alert(value);
}
myfunction(5);
```

In Listing 6-54, we no longer use global variables. The value to be processed is sent to the function when it is called and received by the function through its parameter. When the function is called, the value between the parentheses of the call (5) is assigned to the **value** variable

created by the function to receive this value, and this variable is read inside the function to show its value on the screen.

> **The Basics:** The names declared between parentheses in the function to receive values are called *parameters*. On the other hand, the values specified in the call are called *attributes*. In these terms, we can say that a function's call has attributes that are sent to the function and received by its parameters.

The advantage of using functions is that we can execute their instructions as many times as we need, and because we can send different values in every call, the result produced by each call will be different. The following example executes **myfunction()** two times, but each time it sends a different value to be processed.

***Listing 6-55:*** *Calling the function with different values*

```
function myfunction(value) {
  alert(value);
}
myfunction(5);
myfunction(25);
```

The interpreter executes the first call with the value 5 and, when the execution of the function is over, it calls it again with the value 25. In consequence, two popup windows are opened, one with the value 5 and another with the value 25.

In this and the previous examples, we sent values to the function, but we can also send the current value of a variable.

***Listing 6-56:*** *Sending the value of a variable to a function*

```
var counter = 100;
function myfunction(value) {
  alert(value);
}
myfunction(counter);
```

This time we include the variable **counter** in the call instead of a number. The interpreter reads this variable and sends the current value to the function. The rest of the process is the same; the function receives the value, assigns it to the **value** variable, and shows it on the screen.

Functions can also receive multiple values. All we have to do is to declare the values and the parameters separated by commas.

***Listing 6-57:*** *Sending multiple values to a function*

```
var counter = 100;
var items = 5;
function myfunction(value1, value2) {
  var total = value1 + value2;
  alert(total);  // 105
}
myfunction(counter, items);
```

In Listing 6-57, we add the values received by the function and show the result on the screen, but there are times when the result is required outside the function. To send values from the function to the global namespace, JavaScript includes the **return** instruction. This instruction determines which value must be returned to the code that called the function.

If we want to process the value returned by a function, we have to assign that function to a variable. The interpreter first executes the function and then assigns the value returned by the function to the variable, as in the following example.

***Listing 6-58:*** *Returning values from functions*

```
var counter = 100;
var items = 5;
function myfunction(value1, value2) {
  var total = value1 + value2;
  return total;
}
var result = myfunction(counter, items);
alert(result);
```

The code in Listing 6-58 defines the same **myfunction()** function used before, but this time the value produced by the function is not shown on the screen but returned with the **return** instruction. Back to the global namespace, the value returned by the function (**17**) is assigned to the variable **result**, and the content of that variable is shown on the screen.

Notice that at the beginning of this code we declare the variable **myresult** but do not assign any value to it. This is perfectly acceptable and even recommended. It is good to declare all the variables we are going to work with at the beginning, to avoid confusion and be able to identify each one of them later from other parts of the code.

The **return** instruction finishes the execution of the function. Any instruction declared after a value is returned will not be executed. For this reason, the **return** instruction is usually placed at the end of the function, but this is not mandatory. We can return a value anywhere in the code if we have conditions to satisfy. For example, the following function returns the result of the addition of two values if the total is greater than 100. Otherwise, it returns the value 0.

***Listing 6-59:*** *Returning different values from a function*

```
var counter = 100;
var items = 5;
function myfunction(value1, value2) {
  var total = value1 + value2;
  if (total > 100) {
    return total;
  } else {
    return 0;
  }
}
var result = myfunction(counter, items);
alert(result);
```

# Anonymous Functions

Another way to declare a function is to use an anonymous function. Anonymous functions are functions without a name or identifier (hence

"anonymous"). They can be passed as values to other functions or assigned to variables. When an anonymous function is assigned to a variable, the name of the variable is the one we use later to call the function, as in the following example.

***Listing 6-60:*** *Declaring anonymous functions*

```
var myfunction = function(value) {
  value = value * 2;
  return value;
};
var total = 2;
for (var f = 0; f < 10; f++) {
  total = myfunction(total);
}
alert("The total is " + total);  // "The total is 2048"
```

In Listing 6-60, we declare an anonymous function that receives a value, multiplies that value times 2, and returns the result. Because the function is assigned to a variable, we can use the name of that variable to call the function, so after the definition of the function, we create a **for** loop that calls **myfunction()** multiple times with the current value of the **total** variable. The instruction in the loop assigns the value returned by the function back to the **total** variable, doubling its value in every cycle.

Anonymous functions can be executed right away adding parentheses at the end of their declaration. This is useful when we want to assign the result of a complex operation to a variable. The function performs the operation and returns the result. In this case, it is the value returned by the function the one assigned to the variable and not the function itself.

***Listing 6-61:*** *Executing anonymous functions*

```
var myvalue = function(value) {
  value = value * 2;
  return value;
}(35);
alert("The value is " + myvalue);  // "The value is 70"
```

The anonymous function in Listing 6-61 is defined and executed as soon as the instruction is processed by the interpreter. The function receives the value 35, multiplies that value times 2, and returns the result, which is assigned to the **myvalue** variable.

**The Basics:** Anonymous functions are extremely useful in JavaScript because they allow us to construct complicated programming patterns that are necessary to build professional applications. We will see practical applications of these types of functions and some of the patterns available in JavaScript in later chapters.

# Standard Functions

Besides the functions we can create ourselves, we also have access to functions predefined by JavaScript. These functions perform processes that simplify complex tasks. The following are the most frequently used.

**isNaN(**value**)**—This function returns **true** if the value between parentheses is not a number.

**parseInt(**value**)**—This function turns a string with a number into an integer value that we can process in arithmetic operations.

**parseFloat(**value**)**—This function turns a string with a number into a floating-point value that we can process in arithmetic operations.

**encodeURIComponent(**value**)**—This function encodes a string. It is used to codify characters in a string that might present a conflict when the string is included in a URL.

**decodeURIComponent(**value**)**—This function decodes a string.

Standard functions are global functions that we can call from anywhere in the code. We have to call them as we do with any other function, with the value we want to be processed between parentheses.

***Listing 6-62:*** *Checking whether a value is a number or not*

```
var myvalue = "Hello";
if (isNaN(myvalue)) {
```

```
  alert(myvalue + " is not a number");
} else {
  alert(myvalue + " is a number");
}
```

The **isNaN()** function returns a Boolean value, so we can use it to set a condition. The interpreter first calls the function and then executes the blocks of code defined by the **if else** instructions depending on the result. In this case, the value of the variable is a string of letters, so the **isNaN()** function returns **true** and the message "Hello is not a number" is shown on the screen.

The **isNaN()** function returns **false** not only when the variable contains a number but also when it contains a string with a number. This means that we cannot use the value in an arithmetic operation because it could be a string and the process will not produce the result we are expecting. To make sure that the value can be included in an operation, we have to convert it to a numeric value. For this purpose, JavaScript offers two functions: **parseInt()** for integers and **parseFloat()** for floating-point numbers.

***Listing 6-63:*** *Converting a string into a number*

```
var myvalue = "32";
if (isNaN(myvalue)) {
  alert(myvalue + " is not a number");
} else {
  var number = parseInt(myvalue);
  number = number * 10;
  alert("The number is: " + number);  // "The number is 320"
}
```

The code in Listing 6-63 checks whether the value in the **myvalue** variable is a number or not, as we did before, but this time the value is converted into a numeric value if a number is found. After the value is extracted from the string, we use it to perform a multiplication and then show the result on the screen.

Another useful standard function is **encodeURIComponent()**. With this function we can prepare a string to be included in a URL. The problem with URLs is that they provide special meaning to some characters, like ? or &,

as we have seen in previous chapters (see Figure 2-43). Because users are not aware of these restrictions, we have to codify the strings before to include them in a URL every time they are provided by the user or come from an unreliable source.

***Listing 6-64:*** *Encoding a string to include it in a URL*

```
var name = "John Doe";
var encodedName = encodeURIComponent(name);
var myURL = "http://www.example.com/contact.html?name=" +
encodedName;
alert(myURL);
```

The code in Listing 6-64 adds the value of the **name** variable to a URL. In this example we assume that the value of the variable was defined by the user, so we encode it with the **encodeURIComponent()** function to make sure the final URL is valid. The function analyzes the string and replaces every conflictive character with hexadecimal numbers that represent the character preceded by the % character. In this case, the only character that requires codification is the space between the first and last names. The final URL is http://www.example.com/contact.html?name=John%20Doe.

# 6.3 Objects

Objects are units of information capable of containing other variables (called *properties*) as well as functions (called *methods*). Because objects store values along with functions, they are like independent programs that communicate with each other to perform common tasks.

The idea behind objects in computer programming is to simulate the role of objects in real life. A real object has properties and performs actions. For instance, a person has a name and an address, but it can also walk and talk. The characteristics and functionality are part of the person and is the person who defines how is going to walk and what is going to say. By organizing our code in this manner, we can create independent processing units that can perform tasks and have all the information they need to do it. For instance, we can create an object that controls a button, shows its title and performs a task when the button is pressed. Because all the information to present and manage the button is stored inside the object, the rest of the code does not need to know how to do it. As long as we are aware of the methods provided by the object and the values returned, the code inside the object may be updated or entirely replaced without affecting the rest of the program.

Being able to create independent processing units, duplicate those units as many times as necessary, and modify their values to adapt them to the circumstances, are the main advantages introduced by objects and the reason why Object-Oriented Programming has become the most popular programming paradigm available. JavaScript was created around the concept of objects, and therefore understanding objects is necessary to understand the language and its capabilities.

# Declaring Objects

There are different ways of declaring objects in JavaScript; the simplest is to use literal notation. The object is declared as any other variable using the **var** keyword and the properties and methods that define the object are declared between braces using a colon after the name and commas to separate each declaration.

***Listing 6-65:*** *Creating objects*

```
var myobject = {
  name: "John",
  age: 30
};
```

In Listing 6-65, we declare the object **myobject** with two properties: **name** and **age**. The value of the property **name** is "John" and the value of the property **age** is 30.

Unlike variables, we cannot access the properties' values using only their names; we also have to specify the object they belong to using dot notation or square brackets.

***Listing 6-66:*** *Accessing properties*

```
var myobject = {
  name: "John",
  age: 30
};
var message = "My name is " + myobject.name + "\r\n";
message += "I'm " + myobject["age"] + " years old";
alert(message);
```

In Listing 6-66, we use both notations to access the values of the object's properties and create the message we are going to show on the screen. The use of any notation is irrelevant, except in some special circumstances. For instance, when we need to access the property through the value of a variable, we have to use square brackets.

***Listing 6-67:*** *Accessing properties using variables*

```
var propertyName = "name";
var myobject = {
  name: "John",
  age: 30
};
alert(myobject[propertyName]);  // "John"
```

In Listing 6-67, we couldn't have accessed the property with dot notation (**myobject.propertyName**) because the interpreter would have tried to find a property called **propertyName** that does not exist. Using square brackets, the variable is resolved first, and then the object is accessed with its value ("name") instead of its name.

It is also necessary to access a property using square brackets when its name is considered invalid for a variable (it includes invalid characters, such as a space, or begins with a number). In the following example, the object includes a property declared with a string as a name. We are allowed to declare names of properties as strings, but because this particular string contains a space, the code **myobject.my age** will return an error, so we have to use square brackets to access the property.

***Listing 6-68:*** *Accessing properties with invalid names*

```
var myvariable = "name";
var myobject = {
  name: "John",
  'my age': 30
};
alert(myobject['my age']);  // 30
```

Besides reading the values of properties, we can also set new properties or modify them using both notations. In the next example, we change the value of the **name** property and add a new property to our object called **job**.

***Listing 6-69:*** *Updating values and adding new properties to the object*

```
var myobject = {
  name: "John",
  age: 30
};
myobject.name = "Martin";
myobject.job = "Programmer";
alert(myobject.name + " " + myobject.age + " " + myobject.job);
```

Objects can also contain other objects. In the following example, we assign an object to the property of another object.

***Listing 6-70:*** *Creating objects inside objects*

```
var myobject = {
  name: "John",
  age: 30,
  bike: {
    model: "Susuki",
    year: 1981
  }
};
alert(myobject.name + " owns a " + myobject.bike.model);
```

The **myobject** object in the code in Listing 6-70 includes a property called **bike** which value is another object with the properties **model** and **year**. If we want to access these properties, we have to indicate the name of the object they belong to (**bike**) and the name of the object that object belongs to (**myobject**). The names are concatenated with dot notation in the order they were included in the hierarchy. For example, the **model** property is inside the **bike** property that in turn is inside the **myobject** property, so if we want to read the value of the **model** property, we have to write **myobject.bike.model**.

# Methods

As we mentioned before, objects can also include functions. Functions inside objects are called *methods*. Methods have the same syntax as properties; they require a colon after the name and commas to separate each declaration, but instead of values they are assigned anonymous functions.

***Listing 6-71:*** *Declaring and executing methods*

```
var myobject = {
  name: "John",
  age: 30,
  showinfo: function() {
    var message = "Name: " + myobject.name + "\r\n";
    message += "Age: " + myobject.age;
    alert(message);
```

```
  },
  changename: function(newname) {
    myobject.name = newname;
  }
};
myobject.showinfo();  // "Name: John Age: 30"
myobject.changename("Jonathan");
myobject.showinfo();  // "Name: Jonathan Age: 30"
```

In this example, we add two methods to the object: **showinfo()** and **changename()**. The **showinfo()** method shows a popup window with the values of the **name** and **age** properties, and the **changename()** method assigns the value received by its parameter to the **name** property. These are two independent methods that work on the same properties, one reads the current values, and the other assigns new values. To execute the methods, we use the dot notation and parentheses after the name, as we do with functions.

Like functions, methods can also return values. In the following example, we modify the **changename()** method to return the old name after replacing it by the new one.

***Listing 6-72:*** *Returning values from methods*

```
var myobject = {
  name: "John",
  age: 30,
  showinfo: function() {
    var message = "Name: " + myobject.name + "\r\n";
    message += "Age: " + myobject.age;
    alert(message);
  },
  changename: function(newname) {
    var oldName = myobject.name;
    myobject.name = newname;
    return oldName;
  }
};
var old = myobject.changename("Jonathan");
```

```
alert("Old name was: " + old);  // "Old name was: John"
```

The new **changename()** method stores the current value of the **name** property in a temporary variable called **oldName** to be able to return the previous name after the new one is assigned to the property.

# The this Keyword

In the last examples, we mentioned the name of the object every time we wanted to modify its properties from the methods. Although this works, it is not a good practice. Because the name of the object is determined by the name of the variable to which the object is assigned, it may change. Also, as we will see later, JavaScript allows us to create multiple objects from the same definition or create new objects from other objects, which produce different objects that share the same definition. To make sure that we are referencing the object we are working with from the methods, JavaScript includes the **this** keyword. The **this** keyword is used instead of the object's name to reference the object the instruction belongs to. The following code reproduces the previous example, but this time we use the **this** keyword instead of the object's name to reference the properties. The results are the same as before.

***Listing 6-73:*** *Referencing the object's properties with the* this *keyword*

```
var myobject = {
  name: "John",
  age: 30,
  showinfo: function() {
    var message = "Name: " + this.name + "\r\n";
    message += "Age: " + this.age;
    alert(message);
  },
  changename: function(newname) {
    var oldName = this.name;
    this.name = newname;
    return oldName;
  }
```

```
};
var old = myobject.changename("Jonathan");
alert("Old name was: " + old);  // "Old name was: John"
```

**IMPORTANT:** Every time you want to access properties and methods from inside the object, you should use the **this** keyword to reference the object, but if you try to do the same from outside the object, you will be referencing the JavaScript's global object. The **this** keyword references the object in which the instruction is being executed. This is why, in the code in Listing 6-73 we only use the **this** keyword inside the methods of the **myobject** object, but the instructions in the global namespace use the name of the object instead.

# Constructors

With literal notation, we can create single objects, but if we want to create copies of that object with the same properties and methods, we have to use constructors. A constructor is an anonymous function that defines a new object and returns it, creating copies of the object (also called *instances*), each with its own properties, methods, and values.

***Listing 6-74:*** *Using a constructor to create an object*

```
var constructor = function() {
  var obj = {
    name: "John",
    age: 30,
    showname: function() {
      alert(this.name);
    },
    changename: function(newname) {
      this.name = newname;
    }
  };
  return obj;
};
var employee = constructor();
```

```
employee.showname();  // "John"
```

In Listing 6-74, an anonymous function is assigned to the **constructor** variable. Inside the function, an object is created and returned by the **return** instruction. Finally, the object returned by the function is stored in the **employee** variable and its **showname()** method is executed.

Using constructors, we can create new objects dynamically. For instance, we can set initial values for the properties by sending the values to the function when the object is constructed.

***Listing 6-75:*** *Sending initial values to a constructor*

```
var constructor = function(initialname) {
  var obj = {
    name: initialname,
    age: 30,
    showname: function() {
      alert(this.name);
    },
    changename: function(newname) {
      this.name = newname;
    }
  };
  return obj;
};
var employee = constructor("John");
employee.showname();  // "John"
```

The purpose of a constructor is to work as an objects factory. The following example illustrates how to create multiple objects from the same constructor.

***Listing 6-76:*** *Using constructors to create multiple objects*

```
var constructor = function(initialname) {
  var obj = {
    name: initialname,
    age: 30,
    showname: function() {
```

```
    alert(this.name);
   },
   changename: function(newname) {
    this.name = newname;
   }
  };
  return obj;
};
var employee1 = constructor("John");
var employee2 = constructor("Robert");
var employee3 = constructor("Arthur");
alert(employee1.name + ", " + employee2.name + ", " +
employee3.name);
```

Although the objects created from a constructor share the same properties and methods, they are stored in different places in memory, and therefore they store different values. Every time the **constructor** function is called, a new object is created, and therefore we can assign different values to each one of them. In Listing 6-76, we have created three objects: **employee1**, **employee2**, and **employee3**, and the values "John", "Robert", and "Arthur" were assigned to their **name** properties. When we read the **name** property of any of these objects, we get different values depending on the object the property belongs to (The **alert()** function at the end of the code shows the message "John, Robert, Arthur").

**The Basics:** Every new object created from a constructor is stored in a different space in memory and therefore has its own properties and values, but you can also assign the same object to multiple variables. If you want to make sure that a variable is not referencing the same object, you can compare the variables with the special operators **===** and **!==**. JavaScript also includes the **is()** method inside a global object called **Object** that can be used to know if two variables are referencing the same object (e.g., **Object.is(object1, object2)**).

An advantage of using constructors to create objects is the possibility to define private properties and methods. All the objects we have created have public properties and methods, which means that they can be accessed and

modified from outside the objects. To avoid this situation and make the properties and methods available only for the object that created them, we have to define them as private using a technique called *closure*.

As we explained before, variables created in the global namespace are accessible from anywhere in the code, while variables created inside functions are only accessible from the functions in which they were created. What we did not mention is that functions, and therefore methods, always keep a link to the space where they were created, and therefore they are always bound to the variables declared in that space. When we return an object from a constructor, its methods can still access the function's variables, even when they are no longer in the same scope, and therefore these variables become accessible only to the object.

***Listing 6-77:*** *Defining private properties*

```
var constructor = function() {
  var name = "John";
  var age = 30;
  var obj = {
    showname: function() {
       alert(name);
    },
    changename: function(newname) {
      name = newname;
    }
  };
  return obj;
};
var employee = constructor();
employee.showname();  // "John"
```

The code in Listing 6-77 is exactly the same from the previous example except that instead of declaring **name** and **age** as properties of the object we declare them as variables of the function that is returning the object. The object returned remembers these variables and is the only one that has access to them. There is no way to modify the value of those variables from other instructions in the code other than to use the methods of the object returned by the function (in this case, **showname()** and **changename()**).

This is the reason why the procedure is called *closure*. The function is closed, and its space is no longer accessible, but we keep an element bonded to it (an object in our example).

Notice that the methods in this example access the variables without the **this** keyword. This is because the values are now stored in variables defined by the function and not in properties defined by the object.

# The new Operator

With literal notation and constructors, we have all we need to create single objects or multiple objects from the same definition. But to preserve consistency with other Object-Oriented Programming languages, JavaScript offers a third option. This is a special kind of constructor that works with an operator called **new**. The object is defined by a function, and then the function is called with the **new** operator to create an object from that definition.

***Listing 6-78:*** *Creating objects with the* new *operator*

```
function Myobject(){
  this.name = "John";
  this.age = 30;
  this.showname = function(){
    alert(this.name);
  };
  this.changename = function(newname){
    this.name = newname;
  };
}
var employee = new Myobject();
employee.showname();  // "John"
```

These types of constructors require the object's properties and methods to be identified with the **this** keyword, but except for the use of this keyword, the definition of these constructors and those studied before is the same. As with other constructors, we can also provide initialization values.

***Listing 6-79:*** *Defining initial values for the object*

```
function Myobject(initialname, initialage){
  this.name = initialname;
  this.age = initialage;
  this.showname = function(){
    alert(this.name);
  };
  this.changename = function(newname){
    this.name = newname;
  };
}
var employee = new Myobject("Robert", 55);
employee.showname();  // "Robert"
```

# Inheritance

An important characteristic of objects is that we can create them from other objects. When objects are created from other objects, they inherit their properties and methods. Inheritance in JavaScript (how objects get the same properties and methods from other objects) is done through prototypes. An object does not inherit properties and methods directly from another object; it does it from the object's prototype. Working with prototypes can be very confusing, but JavaScript includes the **create()** method to simplify our job. This method is part of a global object predefined by JavaScript called **Object**. It uses an existing object as prototype of a new one, so we can create objects from other objects without worrying about how their properties and methods are shared among them.

***Listing 6-80:*** *Creating objects from other objects*

```
var myobject = {
  name: "John",
  age: 30,
  showname: function(){
    alert(this.name);
  },
  changename: function(newname){
```

```
    this.name = newname;
  }
};
var employee = Object.create(myobject);
employee.changename('Robert');
employee.showname();  // "Robert"
myobject.showname();  // "John"
```

The code in Listing 6-80 creates the **myobject** object using literal notation and then calls the **create()** method to create a new object based on the **myobject** object. The **create()** method only requires the name of the object that is going to be the prototype of the new one, and it returns a new object that we can assign to a variable for further use. In this example, the new object is first created with **Object.create()** and then assigned to the variable **employee**. Once we have the new object, we can update its values. Using the **changename()** method, we change the name of **employee** to "Robert" and then show the value of the **name** property of each object on the screen.

This code creates two independent objects, **myobject** and **employee**, with their own properties, methods and values, but linked through a prototype chain. The new **employee** object is not just a copy; it is an object that keeps a link to the **myobject**'s prototype. When changes are introduced to this prototype, the objects down the chain inherit those changes.

***Listing 6-81:*** *Adding a new method to the prototype*

```
var myobject = {
  name: "John",
  age: 30,
  showname: function(){
    alert(this.name);
  },
  changename: function(newname){
    this.name = newname;
  }
};
var employee = Object.create(myobject);
employee.age = 24;
```

```
myobject.showage = function(){
  alert(this.age);
};
employee.showage();  // 24
```

In Listing 6-81, a method called **showage()** is added to the prototype object (**myobject**). Because of the prototype chain, this new method is now accessible from the other instances as well. When we call the **showage()** method of **employee** at the end of the code, the interpreter first looks for the method in **employee** and continues searching up the prototype chain until it finds it in **myobject**. When the method is finally found, it shows the value 24 on the screen. This is because, even when the **showage()** method belongs to **myobject**, the **this** keyword in this method points to the object we are working with (**employee**). Because of the prototype chain, the **showage()** method can be invoked from **employee**, and because of the **this** keyword, the value of the **age** property shown on the screen is the one we assigned before to **employee**.

The **create()** method is as simple as it is powerful. It takes an object and turns it into the prototype of a new one. This allows us to build a chain of objects where each one of them inherits properties and methods from its predecessor.

***Listing 6-82:*** *Testing the prototype chain*

```
var myobject = {
  name: "John",
  age: 30,
  showname: function(){
    alert(this.name);
  },
  changename: function(newname){
    this.name = newname;
  }
};
var employee1 = Object.create(myobject);
var employee2 = Object.create(employee1);
var employee3 = Object.create(employee2);
```

```
employee2.showage = function(){
  alert(this.age);
};
employee3.age = 24;
employee3.showage();  // 24
```

In Listing 6-82, the existence of the prototype chain is demonstrated adding the method **showage()** to **employee2**. Now, **employee2** and **employee3** (as well as any object created later based on these two objects) have access to this method, but because inheritance works down the chain, not up, the method is not available for the **employee1** object.

**IMPORTANT:** If you get used to working with the **create()** method, you will not need to access and modify the object's prototypes. In fact, the **create()** method should be the way we work with JavaScript, creating objects without getting into the complexity of their prototypal nature. However, prototypes are the essence of the language, and in some circumstances, we cannot avoid them. To learn more about prototypes and how to work with objects, visit our website and follow the links for this chapter.

# 6.4 Standard Objects

Objects are like wrappers of code, and JavaScript takes advantage of this feature extensively. In fact, almost everything in JavaScript is an object. For instance, the numbers and strings we assign to variables are turned into objects by the JavaScript interpreter in the background. Every time we assign a new value to a variable, we are assigning an object that contains that value.

Because the values we store in variables are of different types, there are different kinds of objects available to represent them. For example, if the value is a string, the object created to store it is of type **String**. When we assign a string to a variable, JavaScript creates a **String** object, stores the string inside the object, and assigns that object to the variable.

If we want, we can create JavaScript objects directly using their constructors. There are different constructors available depending on the type of value we want to store. The following are the most frequently used.

**Number(**value**)**—This constructor creates objects to store numerical values. It can take a number and also a string with a number. If the value specified by the attribute cannot be converted into a number, the constructor returns the special value **NaN** (Not a Number).

**String(**value**)**—This constructor creates objects to store strings of characters. It can take a string or any value that can be converted into a string, such as a number.

**Boolean(**value**)**—This constructor creates objects to store Boolean values. It can take the values **true** and **false**. If other values are provided, they are converted into a Boolean value. If the value is omitted or is equal to 0, **NaN**, **null**, **undefined**, or an empty string, the value stored in the object is **false**. Otherwise, it is **true**.

**Array(**values**)**—This constructor creates objects to store arrays. If multiple values are provided, the constructor creates an array with those values, but if only one value is provided, and that value is an integer, the constructor creates an array with the number of elements indicated by the integer, and the values stored at each index are **undefined**.

These constructors work with the **new** operator to create new objects. The following example stores a number.

***Listing 6-83:*** *Creating numbers with a constructor*

```
var value = new Number(5);
alert(value);  // 5
```

The advantage of using constructors is that they can create the value from values of different types. For example, we can get a number from a string.

***Listing 6-84:*** *Creating numbers from strings*

```
var value = new Number("5");
alert(value);  // 5
```

The string provided to the **Number()** constructor in Listing 6-84 is converted into a numeric value by the constructor and stored in a **Number** object. Because JavaScript takes care of converting these objects into primitive values and vice versa, we can perform arithmetic operations on the value stored in the object, as we do with any other numeric value.

***Listing 6-85:*** *Performing arithmetic operations with objects*

```
var value = new Number("5");
var total = value * 35;
alert(total);  // 175
```

The **Array()** constructor operates in a different way. If we provide multiple values, the array is created as if we had declared it with square brackets.

***Listing 6-86:*** *Creating an array with a constructor*

```
var list = new Array(12, 35, 57, 8);
alert(list);  // 12,35,57,8
```

On the other hand, if we only provide one value and that value is an integer, the constructor uses that number to determine the length of the array and creates an array of that size with **undefined** values.

***Listing 6-87:*** *Creating an empty array with a constructor*

```
var list = new Array(2);
alert(list[0] + " - " + list[1]);  // undefined - undefined
```

> **The Basics:** Whether to assign values directly or use these constructors is up to you. JavaScript can tell the difference between a value and an object, but it is usually irrelevant, and most of the time both types of values operate the same way.

# String Objects

Converting values into objects allows the language to provide all the functionality necessary to manipulate the values within the objects. The following are the most frequently used properties and methods included in **String** objects.

**length**—This property returns an integer representing the number of characters in the string.

**toLowerCase()**—This method converts the characters in the string to lowercase letters.

**toUpperCase()**—This method converts the characters in the string to uppercase letters.

**trim()**—This method removes whitespace characters, such as spaces or tabs, on both sides of the string. JavaScript also includes the methods **trimLeft()** and **trimRight()** to clean the string on only one side.

**substr(**start, length**)**—This method returns a new string with characters from the original. The **start** attribute indicates the position of the first character to read, and the **length** attribute determines how many characters we want to include. If the length is not specified, the method returns all the characters to the end of the string.

**substring(**start, end**)**—This method returns a new string with characters from the original. The **start** and **end** attributes are integer

values that determine the position of the first and last characters to include.

**split(**separator, limit**)**—This method splits the string into parts and returns an array with the substrings. The **separator** attribute indicates the character at which the string is going to be cut, and the **limit** attribute is an integer that determines the maximum number of cuts. If no limit is specified, the string is cut every time the separator is found.

**startsWith(**value**)**—This method returns **true** if the string specified by the **value** attribute is found at the beginning of the string.

**endsWith(**value**)**—This method returns **true** if the string specified by the **value** attribute is found at the end of the string.

**includes(**search, position**)**—This method searches for the value of the **search** attribute in the string and returns **true** or **false** according to the result. The **search** attribute is the string we want to search, and the **position** attribute determines the index at which we want to begin the search. If the **position** attribute is not specified, the search starts from the beginning of the string.

**indexOf(**value, position**)**—This method returns the index at which the string specified by the **value** attribute is found for the first time. The **position** attribute determines the index at which we want to begin the search. If the **position** attribute is not specified, the search starts from the beginning of the string. It returns the value -1 if no match is found.

**lastIndexOf(**value, position**)**—This method returns the index at which the string specified by the **value** attribute is found for the first time. Unlike **indexOf()**, this method performs the search backward, from the end of the string. The **position** attribute determines the index at which we want to begin the search. If the **position** attribute is not specified, the search starts from the end of the string. It returns the value -1 if no match is found.

**replace(**expression, replacement**)**—This method replaces the part of the string that matches the value of the **expression** attribute with the string specified by the **replacement** attribute. The **expression** attribute can be specified as the string we want to replace or as a Regular Expression to search for a string with a particular format.

Strings are stored as arrays of characters, so we can access each character using square brackets, as we do with any other array. JavaScript includes the **length** property to count the number of characters in a string.

***Listing 6-88:*** *Counting the number of characters in a string*

```
var text = "Hello World";
var message = "The text is " + text.length + " characters long";
alert(message);  // "The text is 11 characters long"
```

Because strings are arrays of characters, we can iterate through the characters with a loop. In the following example, we add a space character between letters.

***Listing 6-89:*** *Iterating through the characters of a string*

```
var text = "Hello World";
var message = "";
for (var f = 0; f < text.length; f++) {
  message += text[f] + " ";
}
alert(message);  // "H e l l o  W o r l d"
```

The **length** property returns the number of characters in the string, but the indexes start counting from 0, so we have to create a loop that goes from 0 to the previous value of the **length** property to get all the indexes of the characters in the string. Using these indexes, the **for** loop in Listing 6-89 reads every character using square brackets and adds them to the current value of the **message** variable along with a space character. In consequence, we get a string with all the characters in the original string separated by a space.

This example adds a space character after every character in the string, which means that the final string ends with a space. **String** objects include the **trim()** method to remove these unwanted space characters.

***Listing 6-90:*** *Removing space characters*

```
var text = "Hello World";
var message = "";
for (var f = 0; f < text.length; f++) {
```

```
  message += text[f] + " ";
}
message = message.trim();
alert(message);  // "H e l l o   W o r l d"
```

The ability to access each character of a string allows us to achieve more interesting effects. We just have to detect the position of the character we want to manipulate and then perform the changes we want. The following example adds dots between the letters of each word, but not between the words.

***Listing 6-91:*** *Processing a string*

```
var text = "Hello World";
var message = "";
var previous = "";

for (var f = 0; f < text.length; f++) {
  if (message != "") {
    previous = text[f - 1];
    if (previous != " " && text[f] != " ") {
      message += ".";
    }
  }
  message += text[f];
}
alert(message);  // "H.e.l.l.o W.o.r.l.d"
```

The code in Listing 6-91 checks whether the character we are currently reading and the one we have read in the previous cycle are not space characters before adding a dot "." to the current value of the **message** variable. This way, the dots are inserted only between letters and not at the end or beginning of the words.

Because of the complexity of some of the processes performed on strings, JavaScript includes several methods to simplify our work. For instance, we can replace all the characters in a string with uppercase letters by calling the **toUpperCase()** method.

***Listing 6-92:*** *Converting a string to uppercase*

```
var text = "Hello World";
var message = text.toUpperCase();
alert(message);  // "HELLO WORLD"
```

Sometimes we do not need to work with the whole text but rather with a few words or characters. **String** objects offer the **substr()** and **substring()** methods to copy a piece of text from a string. The **substr()** method copies the set of characters that begins at the index specified by the first attribute. A second attribute can also be provided to determine how many characters we want to include from the initial position.

***Listing 6-93:*** *Copying a set of characters*

```
var text = "Hello World";
var word = text.substr(0, 5);
alert(word);  // "Hello"
```

The **substr()** method in Listing 6-93 copies a total of 5 characters starting with the character at index 0. In return, we get the string "Hello". If we do not specify the number of characters to include, the method returns characters until it reaches the end of the string.

***Listing 6-94:*** *Copying all the characters from a specific index to the end of the string*

```
var text = "Hello World";
var word = text.substr(6);
alert(word);  // "World"
```

The **substr()** method can also take negative values. When a negative index is specified, the method counts backward from the end of the string. The following code copies the same characters as in the previous example.

***Listing 6-95:*** *Referencing characters with negative indexes*

```
var text = "Hello World";
var word = text.substr(-5);
alert(word);  // "World"
```

The **substring()** method works slightly different than the **substr()** method. This method takes two values to determine the first and last characters we want to copy, but it does not include the last character.

***Listing 6-96:*** *Copying characters between two indexes*

```
var text = "Hello World";
var word = text.substring(6, 8);
alert(word);  // "Wo"
```

The **substring()** method in Listing 6-96 copies the characters from the index 6 to 7 of the string (the character at the last index is not included). These are the positions where the characters "Wo" are located and that is the string we get in return.

If we need to extract the words in a string, we can use the **split()** method. This method cuts the string into pieces and returns an array with the values. The method requires a value with the character we want to use as a separator, so if we use a space character, we can split the string into words.

***Listing 6-97:*** *Getting words from a string*

```
var text = "Hello World";
var words = text.split(" ");
alert(words[0] + " / " + words[1]);  // "Hello / World"
```

The **split()** method in Listing 6-97 creates two strings with the words "Hello" and "World" and returns an array with these values, so we can read them as we do with the values of any other array.

Sometimes we do not know where the characters we want to modify are located or if the string even contains those characters. There are several methods available in **String** objects to search for a string. What we use depends on what we want to achieve. For example, the **startsWith()** and **endsWith()** methods search for a string at the beginning and end of the string and return **true** if the string is found.

***Listing 6-98:*** *Searching for a string at the beginning of another string*

```
var text = "Hello World";
if (text.startsWith("He")) {
  alert("The text begins with 'He'");
```

```
}
```

Because these methods return Boolean values, we can use them as the condition of an **if** instruction. In the code in Listing 6-98, we search for the string "He" at the beginning of the **text** variable and show a message in case of success. In this example, the string is found, the method returns **true**, and therefore the message is shown on the screen.

If the text we are looking for can be found in any part of the string, not only at the beginning or the end, we can use the **includes()** method. Like the previous methods, the **includes()** method searches for a string and returns **true** in case of success, but the search is performed in the whole string.

***Listing 6-99:*** *Searching for a string inside another string*

```
var text = "Hello World";
if (text.includes("ll")) {
  alert("The text includes double L");
}
```

So far, we have checked whether one or more characters are found in a string, but sometimes we need to know where those characters were found. There are two methods we can use for this purpose: **indexOf()** and **lastIndexOf()**. Both methods return the index where the first match is found, but the **indexOf()** method searches from the beginning of the string and the **lastIndexOf()** method does it from the end.

***Listing 6-100:*** *Finding the location of a string inside another string*

```
var text = "Hello World";
var index = text.indexOf("World");
alert("The word is at index " + index);  // "The word is at index 6"
```

Knowing the positions of the characters we are looking for, we could program a small piece of code that replaces those characters with a different set, but this is not necessary. There is a method included in **String** objects called **replace()** specifically designed for this purpose. This is a complex method that may process multiple values and work with Regular Expressions to perform the search, but we can also use it with simple

strings. The following example replaces the word "World" with the word "Planet".

***Listing 6-101:*** *Replacing a string with another string*

```
var text = "Hello World";
var newText = text.replace("World", "Planet");
alert(newText);  // "Hello Planet"
```

# Array Objects

As we already mentioned, arrays are also objects in JavaScript, and they include their own properties and methods to manipulate their values. The following are the most frequently used.

**length**—This property returns an integer representing the number of values in the array.

**push(**values**)**—This method adds one or more values at the end of the array and returns the array's new length. There is a counterpart called **unshift()**, which adds the values at the beginning of the array.

**pop()**—This method removes the last value in the array and returns it. There is a counterpart called **shift()**, which removes the first value in the array.

**concat(**array**)**—This method concatenates the array with the array specified by the attribute and returns a new array with the result. The original arrays are not modified.

**splice(**index, remove, values**)**—This method adds and removes values from an array and returns a new array with the elements removed if any. The **index** attribute is the index at which we are going to introduce the modifications, the **remove** attribute is the number of values we want to remove from the index, and the **values** attribute is the list of values separated by commas that we want to add to the array from the index. If we only want to add values, the **remove** attribute may be specified as 0, and if we only want to remove values, we can ignore the third attribute.

**slice(**start, end**)**—This method copies the values at the positions indicated by the attributes into a new array. The **start** and **end** attributes indicate the indexes of the first and last value to copy. The last value is not included in the new array.

**indexOf(**value, position**)**—This method returns the index at which the value specified by the **value** attribute is found for the first time. The **position** attribute determines the index at which we want to begin the search. If the **position** attribute is not specified, the search starts from the beginning of the array. It returns the value -1 if no match is found.

**lastIndexOf(**value, position**)**—This method returns the index at which the value specified by the **value** attribute is found for the first time. Unlike **indexOf()**, this method performs the search backward, from the end of the array. The **position** attribute determines the index at which we want to begin the search. If the **position** attribute is not specified, the search starts from the end of the array. It returns the value -1 if no match is found.

**filter(**function**)**—This method sends the values of the array to a function one by one and returns a new array with all the values approved by the function. The **function** attribute is a reference to a function or an anonymous function in charge of validating the values. The function receives three values: the value to be checked, its index, and a reference to the array. After processing the value, the function must return a Boolean value indicating if it is valid or not.

**every(**function**)**—This method sends the values of the array to a function one by one and returns **true** when all the values are approved by the function. The **function** attribute is a reference to a function or an anonymous function in charge of checking the values. The function receives three values: the value to be checked, its index, and a reference to the array. After processing the value, the function must return a Boolean value indicating if it is valid or not. There is a similar method available called **some()** that returns **true** if at least one value is approved by the function.

**join(**separator**)**—This method creates a string with all the values in the array. The **separator** attribute specifies the character or string of characters we want to include between values.

**reverse()**—This method reverses the order of the values in the array.

**sort(**function**)**—This method sorts the values in the array. The **function** attribute is a reference to a function or an anonymous function in charge of comparing the values. The function receives two values from the array and must return a Boolean value indicating their order. If the attribute is not specified, the method sorts the elements alphabetically in ascending order.

**map(**function**)**—This method sends the values of the array to a function one by one and creates a new array with the values returned by the function. The **function** attribute is a reference to a function or an anonymous function in charge of processing the values. The function receives three values: the value to be processed, its index, and a reference to the array.

As well as **String** objects, **Array** objects offer the **length** property to get the number of values in the array. The implementation is the same; we have to read the property on the array to get the value in return.

***Listing 6-102:*** *Getting the number of values in the array*

```
var list = [12, 5, 80, 34];
alert(list.length);  // 4
```

Using this property, we can iterate through the array with a **for** loop, as we did with strings before (see Listing 6-89). The value returned by the property is used to define the condition of the loop.

***Listing 6-103:*** *Iterating through the array*

```
var list = [12, 5, 80, 34];
var total = 0;
for (var f = 0; f < list.length; f++) {
  total += list[f];
}
alert("The total is: " + total);  // "The total is: 131"
```

In the code in Listing 6-103, the value returned by the property is 4, so the loop goes from 0 to 3. Using these values inside the loop, we can read

the array's values and process them.

Although we can iterate through the array to read and process the values one by one, **Array** objects offer other methods to access the values. For example, if we want to process only some of the values in the array, we can get a copy with the **slice()** method.

***Listing 6-104:*** *Creating an array from the values of another array*

```
var list = [12, 5, 80, 34];
var newList = list.slice(1, 3);
alert(newList);  // 5,80
```

The **slice()** method returns a new array with the values between the index specified by the first attribute and the one before the index specified by the second attribute. In Listing 6-104, the method accesses the values in the indexes 1 and 2, returning the numbers 5 and 80.

If we want to examine the values before they are included in the new array, we have to use a filter. For this purpose, **Array** objects offer the **filter()** method. This method sends each value to a function and includes the value in the new array only when the function returns **true**. In the following example, we return **true** when the value is less than or equal to 50. As a result, the new array contains all the values in the original array except the value 80.

***Listing 6-105:*** *Filtering the values of an array*

```
var list = [12, 5, 80, 34];
var newList = list.filter(function(value) {
  if (value <= 50) {
    return true;
  } else {
    return false;
  }
});
alert(newList);  // 12, 5, 34
```

In the function provided to the **filter()** method in Listing 6-105, we compare the value from the array against the number 50 and return **true** or **false** depending on the condition, but conditions already return a Boolean

value when they are evaluated, so we can return the condition itself and simplify the code.

***Listing 6-106:*** *Returning a condition to filter the elements*

```
var list = [12, 5, 80, 34];
var newList = list.filter(function(value) {
  return value <= 50;
});
alert(newList);  // 12, 5, 34
```

Similar to **filter()** are the methods **every()** and **some()**. These methods check the values with a function, but instead of returning an array with the values validated by the function, they return the values **true** or **false**. The **every()** method returns **true** if all the values are validated, and the **some()** method returns **true** if at least one of the values is validated. The following example uses the **every()** function to check that all the values in the array are less than or equal to 100.

***Listing 6-107:*** *Checking the values of an array*

```
var list = [12, 5, 80, 34];
var valid = list.every(function(value) {
  return value <= 100;
});
if (valid) {
  alert("The values are not greater than 100");
}
```

If all we want is to put all the values in a string to show them to the user, we can call the **join()** method. This method creates a string with the array's values separated by a character or a string of characters. The following example creates a string with a hyphen between the values.

***Listing 6-108:*** *Creating a string with the values of an array*

```
var list = [12, 5, 80, 34];
var message = list.join("-");
alert(message);  // "12-5-80-34"
```

Another way to access the values of an array is with the **indexOf()** and **lastIndexOf()** methods. These methods search for a value and return the index of the first match found. The **indexOf()** method starts searching from the beginning of the array and the **lastIndexOf()** method starts searching from the end.

***Listing 6-109:*** *Getting the index of a value in an array*

```
var list = [12, 5, 80, 34, 5];
var index = list.indexOf(5);
alert("The value " + list[index] + " is at index " + index);
```

The code in Listing 6-109 looks for the value 5 in the array and returns the index 1. This is because the method only searches for the first match. If we want to find all the values that match the value we are looking for, we have to create a loop to perform multiple searches, as in the following example.

***Listing 6-110:*** *Searching for multiple values in an array*

```
var list = [12, 5, 80, 34, 5];
var search = 5;
var last = 0;
var counter = 0;

while (last >= 0) {
  var last = list.indexOf(5, last);
  if (last != -1) {
    last++;
    counter++;
  }
}
alert("There is a total of " + counter + " values " + search);
```

The **indexOf()** method can take a second attribute that specifies the location at which we want to begin the search. Using this attribute, we can tell the method not to look at locations where a value was already found. In Listing 6-110, we use the **last** variable for this purpose. This variable stores the index of the last value found by the **indexOf()** method. At first, the

variable is initialized with the value 0, which means that the method will begin the search from the index 0 of the array, but as soon as a search is performed, the variable is updated with the index returned by the **indexOf()** method, moving the starting point for the next search to a new location. The loop keeps going until the **indexOf()** method returns -1 (no matches are found).

Besides establishing the new location with the **last** variable, the loop also increments the value of the **counter** variable to keep track of the number of matches found. After the process is over, the code shows a message on the screen to communicate this information ("There is a total of 2 values 5").

Up to this point, we have been working with the same original array, but arrays can be extended or reduced. Adding values to an array is as simple as calling the **push()** or **unshift()** methods. The **push()** method adds the value at the end of the array, and the **unshift()** method adds it at the beginning.

***Listing 6-111:*** *Adding values to an array*

```
var list = [12, 5, 80, 34];
list.push(100);
alert(list);  // 12,5,80,34,100
```

To add multiple values, we have to specify them separated by commas.

***Listing 6-112:*** *Adding multiple values to an array*

```
var list = [12, 5, 80, 34];
list.push(100, 200, 300);
alert(list);  // 12,5,80,34,100,200,300
```

Another way to add multiple values to an array is with the **concat()** method. This method concatenates two arrays and returns a new array with the result (the original arrays are not modified).

***Listing 6-113:*** *Concatenating two arrays*

```
var list = [12, 5, 80, 34];
var newList = list.concat([100, 200, 300]);
alert(newList);  // 12,5,80,34,100,200,300
```

Removing values is also easy with the **pop()** and **shift()** methods. The **pop()** method removes the value at the end of the array and the **shift()** method removes it at the beginning.

***Listing 6-114:*** *Removing values from an array*

```
var list = [12, 5, 80, 34];
list.pop();
alert(list);  // 12,5,80
```

The previous methods add or remove values from the beginning or the end of the array. If we want to have more flexibility, we can use the **splice()** method. This method adds or removes values from any location in the array. The first attribute of the **splice()** method specifies the index where we want to start removing and the second attribute determines how many elements will be removed. For example, the following code removes 2 values from index 1.

***Listing 6-115:*** *Removing values from an array*

```
var list = [12, 5, 80, 34];
var removed = list.splice(1, 2);
alert("Values remaining: " + list);  // 12,34
alert("Values removed: " + removed);  // 5,80
```

With this method, we can also add new values to an array in specific positions. The values must be specified after the first two attributes separated by commas. The following example adds the values 24, 25, and 26 at index 2. Because no values are going to be removed, we declare the value 0 for the second attribute.

***Listing 6-116:*** *Adding values to an array in a specific position*

```
var list = [12, 5, 80, 34];
list.splice(2, 0, 24, 25, 26);
alert(list);  // 12,5,24,25,26,80,34
```

**Array** objects also include methods to order the values in the array. For example, the **reverse()** method sorts the array in reverse order.

***Listing 6-117:*** *Sorting the values of an array in reverse order*

```
var list = [12, 5, 80, 34];
list.reverse();
alert(list);  // 34,80,5,12
```

A better method to sort an array is called **sort()**. This method can take a function to decide in which order the values are going to be placed. If no function is provided, the method sorts the array alphabetically in ascending order.

***Listing 6-118:*** *Sorting the values in alphabetical order*

```
var list = [12, 5, 80, 34];
list.sort();
alert(list);  // 12,34,5,80
```

Notice that the values in the last example are ordered alphabetically, not numerically. If we want the method to consider the numerical order or achieve a different type of arrangement, we have to provide a function. The function receives two values from the array and must return a Boolean value to indicate their order. For instance, to sort the values in ascending order, we have to return **true** if the first value is greater than the second value or **false** otherwise.

***Listing 6-119:*** *Sorting the values in numerical order*

```
var list = [12, 5, 80, 34];
list.sort(function(value1, value2) {
  return value1 > value2;
});
alert(list);  // 5,12,34,80
```

A useful method included in **Array** objects is **map()**. With this method, we can process the values of the array one by one and create a new array with the results. Like previous methods, this method sends the values to a function, but instead of a Boolean value, the function must return the value we want to store in the new array. For example, the following code multiplies each value times 2 and returns the results.

***Listing 6-120:*** *Processing the values and storing the results*

```
var list = [12, 5, 80, 34];
var newList = list.map(function(value) {
  return value * 2;
});
alert(newList);  // 24,10,160,68
```

# Date Objects

Managing dates is a complicated task, not only because they are composed of several values that represent different things but also because the values are dependent on one another. If one value is increased over its limit, it affects the rest of the values on the date. And there are different limits for each component. The limit for minutes and seconds is 60, but the limit for hours is 24, and every month has a different number of days. Dates also have to comply with different time zones, daylight saving times, etc. To simplify the work for developers, JavaScript defines an object called **Date**. The **Date** object stores a date and takes care of keeping the values within their range. The date in these objects is stored in milliseconds, which allows us to perform operations between dates, calculate intervals, etc. JavaScript offers the following constructor to create **Date** objects.

**Date(**value**)**—This constructor creates a value in milliseconds to represent a date based on the value provided by the attribute. The **value** attribute may be specified as a string or as the components of a date separated by commas in the order year, month, day, hours, minutes, seconds, milliseconds. If no value is provided, the constructor creates a date from the current date in the system.

The date stored in these objects is represented by a value in milliseconds calculated since January 1st, 1970. Because the value is not familiar to users, **Date** objects offer the following methods to get the components of the date, such as the year or the month.

**getFullYear()**—This method returns an integer representing the year (a 4 digits value).

**getMonth()**—This method returns an integer representing the month (a value from 0 to 11).
**getDate()**—This method returns an integer representing the day of the month (a value from 1 to 31).
**getDay()**—This method returns an integer representing the day of the week (a value from 0 to 6).
**getHours()**—This method returns an integer representing the hours (a value from 0 to 23).
**getMinutes()**—This method returns an integer representing the minutes (a value from 0 to 59).
**getSeconds()**—This method returns an integer representing the seconds (a value from 0 to 59).
**getMilliseconds()**—This method returns an integer representing the milliseconds (a value from 0 to 999).
**getTime()**—This method returns an integer representing the milliseconds from January 1st, 1970 to the date.

The **Date** objects also include the following methods to modify the components of the date.

**setFullYear(**year**)**—This method sets the year of the date (a 4 digits value). It can also take values to set the month and day.
**setMonth(**month**)**—This method sets the month of the date (a value from 0 to 11). It can also take a value to set the day.
**setDate(**day**)**—This method sets the day of the date (a value from 1 to 31).
**setHours(**hours**)**—This method sets the hours of the date (a value from 0 to 23). It can also take values to set the minutes and seconds.
**setMinutes(**minutes**)**—This method sets the minutes of the date (a value from 0 to 59). It can also take a value to set the seconds.
**setSeconds(**seconds**)**—This method sets the seconds of the date (a value from 0 to 59). It can also take a value to set the milliseconds.
**setMilliseconds(**milliseconds**)**—This method sets the milliseconds of the date (a value from 0 to 999).

The **Date** object also offers methods to turn the date into a string.

**toString()**—This method turns a date into a string. The value in the string is expressed in American English and the format "Wed Jan 04 2017 22:32:48 GMT-0500 (EST)".

**toDateString()**—This method turns a date into a string, but it only returns the date part of the value. The value in the string is expressed in American English and the format "Wed Jan 04 2017".

**toTimeString()**—This method turns a date into a string, but it only returns the time part of the value. The value in the string is expressed in American English and the format "23:21:55 GMT-0500 (EST)".

Every time we need a date, we have to create a new object with the **Date()** constructor. If we do not provide a date, the constructor creates the **Date** object with the current date in the system.

***Listing 6-121:*** *Creating a* Date *object*

```
var mydate = new Date();
alert(mydate);  // "Wed Jan 04 2017 20:51:17 GMT-0500 (EST)"
```

The date may be declared with a string containing a date expressed in human-readable format. The constructor takes care of converting the string into milliseconds.

***Listing 6-122:*** *Creating a* Date *object from a string*

```
var mydate = new Date("January 20 2017");
alert(mydate);  // "Fri Jan 20 2017 00:00:00 GMT-0500 (EST)"
```

Because browsers interpret the string in different ways, instead of a string, it is recommended to create the date declaring the values of the components separated by commas. The following example creates a **Date** object with the date 2017/02/15 12:35.

***Listing 6-123:*** *Creating a* Date *object from date components*

```
var mydate = new Date(2017, 1, 15, 12, 35, 0);
```

```
alert(mydate);  // "Wed Feb 15 2017 12:35:00 GMT-0500 (EST)"
```

The values must be declared in the order Year, Month, Day, Hours, Minutes, and Seconds. Notice that the month is represented by a number from 0 to 11, so we had to declare the value 1 in the constructor to represent the month of February.

> **The Basics:** Dates are represented with a value in milliseconds, but every time you show them on the screen with the **alert()** function the browser automatically turns them into strings with a format the user can understand. This is only done by the browser when the date is going to be presented to the user with predefined functions like **alert()**. If we want to format the date in our code, we can extract the components and include them in a string, as we will see later, or call the methods provided by **Date** objects for this purpose. The **toString()** method creates a string with the whole date, the **toDateString()** method includes only the date components, and the **toTimeString()** method includes only the time.

Because dates are stored in milliseconds, every time we want to process the values of their components, we have to retrieve them with the methods provided by the object. For example, to get the year of a date, we have to use the **getFullYear()** method.

***Listing 6-124:*** *Reading the components of a date*

```
var today = new Date();
var year = today.getFullYear();
alert("The year is " + year);  // "The year is 2017"
```

The other components are retrieved the same way with their corresponding methods. The only thing we have to consider is that months are represented with values from 0 to 11, so we have to add 1 to the value returned by the **getMonth()** method to get a value our users can understand.

***Listing 6-125:*** *Reading the month*

```
var today = new Date();
var year = today.getFullYear();
var month = today.getMonth() + 1;
```

```
var day = today.getDate();
alert(year + "-" + month + "-" + day);  // "2017-1-5"
```

These methods are also useful for incrementing or decrementing a date. For example, if we want to determine the date 15 days from now, we have to get the current day, add the value 15 to it, and assign the result back to the date.

***Listing 6-126:*** *Incrementing a date*

```
var today = new Date();
alert(today);  // "Thu Jan 05 2017 14:37:28 GMT-0500 (EST)"
today.setDate(today.getDate() + 15);
alert(today);  // "Fri Jan 20 2017 14:37:28 GMT-0500 (EST)"
```

If instead of incrementing or decrementing a date by a period of time we want to calculate the difference between two dates, we have to subtract one date from another.

***Listing 6-127:*** *Calculating an interval*

```
var today = new Date(2017, 0, 5);
var future = new Date(2017, 0, 20);
var interval = future - today;
alert(interval);  // 1296000000
```

The value returned by the subtraction is expressed in milliseconds. From this value, we can extract any component we want. All we have to do is to divide the number by the maximum values of each component. For example, if we want to express the number in seconds, we have to divide it by 1000 (1000 milliseconds = 1 second).

***Listing 6-128:*** *Calculating an interval in seconds*

```
var today = new Date(2017, 0, 5);
var future = new Date(2017, 0, 20);
var interval = future - today;
var seconds = interval / 1000;
alert(seconds + " seconds");  // "1296000 seconds"
```

Of course, seconds are not easy to put into context, but days are. To express the interval in days, we have to keep dividing the number. If we divide milliseconds by 1000 we get seconds, the result divided by 60 gives us minutes, that result divided by 60 again gives us hours, and dividing the result by 24 we get days (24 hours = 1 day).

***Listing 6-129:*** *Calculating an interval in days*

```
var today = new Date(2017, 0, 5);
var future = new Date(2017, 0, 20);
var interval = future - today;
var days = interval / (24 * 60 * 60 * 1000);
alert(days + " days");  // "15 days"
```

Another useful thing we can do with dates is to compare them. Although we can compare **Date** objects directly to one another, some comparator operators do not compare the dates but the objects themselves, so the best way to do it is to first get the values of the dates in milliseconds with the **getTime()** method and then compare those values.

***Listing 6-130:*** *Comparing two dates*

```
var today = new Date(2017, 0, 20, 10, 35);
var future = new Date(2017, 0, 20, 12, 35);
if (future.getTime() == today.getTime()) {
  alert("Equal dates");
} else {
  alert("Different dates");
}
```

The code in Listing 6-130 checks whether two dates are equal or not and displays a message to communicate the result. These dates are the same, except for the time. One date was set with the time 10:35 and the second date was set with the time 12:35, and therefore they are considered different. If we want to compare only the date part of the date without considering the time, we have to reset the time components. For example, we can set the hours, minutes and seconds on both dates to 0 and then the only values left to compare are the year, month and day.

***Listing 6-131:*** *Comparing only the date without the time*

```
var today = new Date(2017, 0, 20, 10, 35);
var future = new Date(2017, 0, 20, 12, 35);
today.setHours(0, 0, 0);
future.setHours(0, 0, 0);

if (future.getTime() == today.getTime()) {
  alert("Equal dates");
} else {
  alert("Different dates");
}
```

In Listing 6-131, before comparing the dates, we reset the hours, minutes and seconds with the **setHours()** method. Now, the dates have their time set to 0, and the comparison operator only has to check whether the date is equal or not. In this case, the dates are the same and the message "Equal dates" is shown on the screen.

> **IMPORTANT:** Beside the methods studied in this chapter, **Date** objects also include methods that take into consideration the locale (the user location and language). You should consider implementing these methods when developing a website or application that could be used by people that speak a language other than English, or when you have to take into account local variations, such as daylight saving times. A JavaScript library that developers often use to simplify working with dates is called MomentJS, available at www.momentjs.com. We will learn more about libraries and how to implement them later in this chapter.

# Math Object

There are objects in JavaScript which function is not to store values but to provide properties and methods to process values from other objects. This is the case of the **Math** object. This object does not include a constructor to create more objects, but it defines several properties and methods that we can access from its definition to get the values of mathematical constants

and perform arithmetic operations. The following are the most frequently used.

**PI**—This property returns the value of PI.

**E**—This property returns the Euler's constant.

**LN10**—This property returns the natural logarithm of 10. The object also includes the properties **LN2** (natural logarithm of 2), **LOG2E** (base 2 logarithm of E), and **LOG10E** (base 10 logarithm of E).

**SQRT2**—This property returns the square root of 2. The object also includes the property **SQRT1_2** (the square root of 1/2).

**ceil(**value**)**—This method rounds a value up to the next integer and returns the result.

**floor(**value**)**—This method rounds a value down to the next integer and returns the result.

**round(**value**)**—This method rounds a value to the nearest integer and returns the result.

**trunc(**value**)**—This method removes the fractional digits of a value and returns the integer.

**abs(**value**)**—This method returns the absolute value of a number (inverts negative values to get a positive number).

**min(**values**)**—This method returns the smallest number of a list of values separated by commas.

**max(**values**)**—This method returns the largest number of a list of values separated by commas.

**random()**—This method returns a random number in a range between 0 and 1.

**pow(**base, exponent**)**—This method returns the result of raising the base to the power of the exponent.

**exp(**exponent**)**—This method returns the result of raising E to the power of the exponent. The object also includes the **expm1()** method, which returns the same result minus 1.

**sqrt(**value**)**—This method returns the square root of a value.

**log10(**value**)**—This method returns the base 10 logarithm of a value. The object also includes the methods **log()** (returns the base E logarithm), **log2()** (returns the base 2 logarithm), and **log1p()** (returns the base E logarithm of 1 plus a number).

**sin(**value**)**—This method returns the sine of a number. The object also includes the methods **asin()** (returns the arcsine of a number), **sinh()** (returns the hyperbolic sine of a number), and **asinh()** (returns the hyperbolic arcsine of a number).

**cos(**value**)**—This method returns the cosine of a number. The object also includes the methods **acos()** (returns the arccosine of a number), **cosh()** (returns the hyperbolic cosine of a number), and **acosh()** (returns the hyperbolic arccosine of a number).

**tan(**value**)**—This method returns the tangent of a number. The object also includes the methods **atan()** (returns the arctangent of a number), **atan2()** (returns the arctangent of a number), **tanh()** (returns the hyperbolic tangent of a number), and **atanh()** (returns the hyperbolic arctangent of a number).

Because no copies are created from this object, we have to read the properties and call the methods from the object itself (e.g., **Math.PI**). Other than that, the application of these methods and values is straightforward, as shown in the following example.

***Listing 6-132:*** *Performing arithmetic operations with the* Math *object*

```
var square = Math.sqrt(4);  // 2
var power = Math.pow(2, 2);  // 4
var maximum = Math.max(square, power);
alert("The maximum value is " + maximum);  // "The maximum value is 4"
```

The code in Listing 6-132 gets the square root of 4, calculates 2 to the power of 2, and then compares the results to get the maximum value using the **max()** method. The following example demonstrates how to calculate a random number.

***Listing 6-133:*** *Getting a random number*

```
var randomnumber = Math.random() * (11 - 1) + 1;
```

```
var value = Math.floor(randomnumber);
alert("The number is: " + value);
```

The **random()** method returns a number between 0 and 1. If we want to get a different range of numbers, we have to multiply this value by the formula **(max - min) + min**, where **min** and **max** are the minimum and maximum values we want to include. In our example, we want to get a random number from 1 to 10, but we had to define the values as 1 and 11 because the maximum value is not included in the range. Also, the number returned by the **random()** method is a floating-point value. If we want to get integers, we have to round the numbers with the **floor()** method.

**The Basics:** The **floor()** and **ceil()** methods are probably two of the most frequently used methods included in the **Math** object. The **floor()** method rounds a number down to the nearest integer, and the **ceil()** method rounds the number up to the nearest integer. For instance, if the number to be processed is 5.86, the **floor()** method will return the value 5 and the **ceil()** method will return the value 6. We will implement these and other **Math** methods in practical situations in later chapters.

# Window Object

Every time the browser is opened or a new tab is initialized, a global object called **Window** is created to reference the browser window and provide some methods and properties to control it. The object is stored in a property of the JavaScript's global object called **window**. Through this property, we can connect with the browser and the document from our code.

The **Window** object includes, in turn, other objects to provide additional information related to the window and the document. The following are some of the properties available to access these objects.

**location**—This property contains a **Location** object with information about the origin of the current document. It can also be used as a property to set or return the URL of the document (e.g., **window.location = "http://www.formasterminds.com"**).

**history**—This property contains a **History** object with properties and methods to manipulate the navigation history. We will study this object in Chapter 19.

**navigator**—This property contains a **Navigator** object with information about the application and the device. We will study some of the properties of this object, such as **geolocation** (used to detect the user's location) in later chapters.

**document**—This property contains a **Document** object, which provides access to the objects that represent the HTML elements in the document.

Besides these valuable objects, the **Window** object also offers its own properties and methods. The following are the most frequently used.

**innerWidth**—This property returns the width of the window in pixels.

**innerHeight**—This property returns the height of the window in pixels.

**scrollX**—This property returns the number of pixels the document has been scrolled horizontally.

**scrollY**—This property returns the number of pixels the document has been scrolled vertically.

**alert(**value**)**—This method shows a popup window on the screen displaying the value between parentheses.

**confirm(**message**)**—This method is similar to **alert()**, but it offers two buttons, OK and Cancel, for the user to choose from. It returns **true** or **false**, according to the user's response.

**prompt(**message**)**—This method shows a popup window with an input field to let the user insert a value. The method returns the value inserted by the user.

**setTimeout(**function, milliseconds**)**—This method executes the function specified in the first attribute after the time in milliseconds specified by the second attribute. The **Window** object also offers the **clearTimeout()** method to cancel the process.

**setInterval(**function, milliseconds**)**—This method is similar to **setTimeout()** but it calls the function repeatedly. The **Window** object

also offers the **clearInterval()** method to cancel the process.

**open(**URL, window, parameters**)**—This method opens a document in a new window. The **URL** attribute is the URL of the document we want to open, the **window** attribute is the name of the window where we want to show the document (if no name is provided or the window does not exist, the document is opened in a new window), and the **parameters** attribute is a list of configuration parameters separated by comma that set the characteristics of the window (e.g., " resizable=no,scrollbars=no"). The **Window** object also offers the **close()** method to close a window opened with this method.

The **Window** object controls aspects of the window, its content, and the metadata associated with it, such as the location of the current document, size, offset, etc. For instance, we can load a new document by changing the value of the location.

***Listing 6-134:*** *Defining a new location*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function doit() {
      window.location = "http://www.formasterminds.com";
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <button type="button" onclick="doit()">Press Button</button>
  </section>
</body>
</html>
```

In the document of Listing 6-134, we declare a function call **doit()** that assigns a new URL to the **location** property of the **Window** object. A call to the function is assigned later to the **onclick** attribute of a **<button>** element to execute the function when the button is pressed.

The **location** property contains a **Location** object with its own properties and methods, but we can also assign a string with a URL directly to the property to define a new location for the content of the window. Once the value is assigned to the property, the browser loads the document in that URL and shows it on the screen.

**Do It Yourself:** Create a new HTML file with the document of Listing 6-134 and open the document in your browser. You should see a title and a button. Press the button. The browser should load the www.formasterminds.com website.

Besides assigning a new URL to the **location** property, we can also work with the location from the methods provided by the **Location** object.

**assign(**URL**)**—This method asks the browser to load the document in the location indicated by the **URL** attribute.

**replace(**URL**)**—This method asks the browser to replace the current document with the document in the location indicated by the **URL** attribute. It differs with the **assign()** method in that it does not add the URL to the browser's history.

**reload(**value**)**—This method asks the browser to reload the current document. It can take a Boolean attribute that determines whether the resource has to be downloaded from the server or it can be loaded from the browser's cache (**true** or **false**).

The following example refreshes the page when the user presses the button. Notice that this time we do not mention the **window** property. The **Window** object is a global object, and therefore the interpreter infers that the properties and methods belong to this object. This is the reason why in previous examples we never called the **alert()** method with the instruction **window.alert()**.

***Listing 6-135:*** *Refreshing the page*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function doit() {
      location.reload();
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <button type="button" onclick="doit()">Press Button</button>
  </section>
</body>
</html>
```

The **Window** object also offers the **open()** method to load new content. In the following example, the www.formasterminds.com website is opened in a new window or tab.

***Listing 6-136:*** *Opening a new window*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function doit() {
      open("http://www.formasterminds.com");
    }
  </script>
</head>
<body>
```

```
  <section>
    <h1>Website</h1>
    <button type="button" onclick="doit()">Press Button</button>
  </section>
</body>
</html>
```

Other important methods of the **Window** object are **setTimeout()** and **setInterval()**. These methods execute an instruction after a certain period of time. The **setTimeout()** method executes the instruction once, and the **setInterval()** method executes the instruction repeatedly until the process is canceled. If instead of one instruction we want to execute several, we can specify a reference to a function. Every time the time runs out, the function is executed.

***Listing 6-137:*** *Using timers to execute functions*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function doit() {
      var today = new Date();
      var time = today.toTimeString();
      alert(time);
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <button type="button" onclick="setTimeout(doit, 5000)">Press
Button</button>
  </section>
</body>
</html>
```

These methods take values in milliseconds. A millisecond is 1000 parts of a second, so if we want to specify the time in seconds, we have to multiply the value times 1000. In our example, we want the **doit()** function to be executed every 5 seconds, and therefore we declare the value 5000 as the time to wait before the function is called.

**IMPORTANT:** Notice that the function is declared without the parentheses. When we want to call a function, we have to declare the parentheses after the name, but when we want to reference the function, we have to omit the parentheses. Because this time we wanted to assign a reference to the function and not the result of its execution, we declared the name without parentheses.

If we need to execute a function over and over again after a period of time, we have to use the **setInterval()** method. This method works exactly like **setTimeout()** but it keeps running until we tell it to stop with the **clearInterval()** method. To identify the method we want to be canceled, we have to store the reference returned by the **setInterval()** method in a variable and use that variable to reference the method later, as in the following example.

***Listing 6-138:*** *Cancelling a timer*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    var seconds = 0;
    var timer = setInterval(doit, 1000);
    function doit() {
      seconds++;
    }
    function cancel() {
      clearInterval(timer);
      alert("Total " + seconds + " seconds");
```

```
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <button type="button" onclick="cancel()">Press Button</button>
  </section>
</body>
</html>
```

The code in Listing 6-138 includes two functions. The **doit()** function is executed every 1 second by the **setInterval()** method and the **cancel()** function is executed when the user presses the button. The purpose of this code is to increase the value of a variable called **seconds** every one second until the user decides to cancel the process and see the total amount accumulated up to that moment.

**Do It Yourself:** Replace the document in your HTML file with the code in Listing 6-138 and open the new document in your browser. Wait a moment and press the button. You should see a popup window with the number of seconds elapsed so far.

**IMPORTANT:** The **setTimeout()** and **setInterval()** methods are required in the construction of small applications and animations. We will review these methods in more practical situations in later chapters.

# Document Object

As mentioned before, almost everything in JavaScript is an object, and this includes the elements in the document. When an HTML document is loaded, the browser creates an internal structure to process it. The structure is called *DOM* (Document Object Model), and it is composed of multiple objects of type **Element** (or more specific types that inherit from **Element**) that represent each element in the document.

The **Element** objects keep a permanent connection with the elements they represent. When an object is modified, its element is modified as well, and the result is shown on the screen. To provide access to these objects and let us work with their properties from our JavaScript code, they are stored in an object called **Document** that is later assigned to the **document** property of the **Window** object.

Among other alternatives, the **Document** object includes the following properties to provide quick access to the **Element** objects that represent the most common elements in the document.

**forms**—This property returns an array with references to all the **Element** objects that represent the **<form>** elements in the document.

**images**—This property returns an array with references to all the **Element** objects that represent the **<img>** elements in the document.

**links**—This property returns an array with references to all the **Element** objects that represent the **<a>** elements in the document.

These properties return an array of objects referencing all the elements of a particular type, but the **Document** object also includes the following methods to access individual objects or to obtain lists of objects based on other parameters.

**getElementById(**id**)**—This method returns a reference to the **Element** object that represents the element identified with the value specified by the attribute (the value assigned to the **id** attribute).

**getElementsByClassName(**class**)**—This method returns an array with references to the **Element** objects that represent the elements identified by the class specified by the attribute (the value assigned to the **class** attribute).

**getElementsByName(**name**)**—This method returns an array with references to the **Element** objects that represent the elements identified by the name specified by the attribute (the value assigned to the **name** attribute).

**getElementsByTagName(**type**)**—This method returns an array with references to the **Element** objects that represent the type of elements

specified by the attribute. The attribute is the keyword that identifies the elements, such as **h1**, **p**, **img**, **div**, etc.

**querySelector(**selector**)**—This method returns a reference to the **Element** object that represents the element that matches the selector specified by the **selector** attribute. The method returns the first element in the document that matches the CSS selector (see Chapter 3 to learn how to build these selectors).

**querySelectorAll(**selectors**)**—This method returns an array with references to the **Element** objects that represent the elements that match the selectors specified by the attribute. One or more selectors may be declared separated by commas.

By accessing the **Element** objects in the DOM, we can read and modify the elements in the document, but before doing it, we must consider that the document is read sequentially by the browser and we cannot reference an element that has not yet been created. The best solution is to execute the JavaScript code only when the **load** event occurs. We have studied this event at the beginning of this chapter. Browsers fire the event after the document is loaded and all the objects in the DOM were created and are accessible. The following example includes the **onload** attribute in the **<body>** element to be able to access a **<p>** element from JavaScript.

***Listing 6-139:*** *Getting a reference to the* Element *object that represents an element*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function initiate() {
      var element = document.getElementById("subtitle");
    }
  </script>
</head>
<body onload="initiate()">
```

```
  <section>
    <h1>Website</h1>
    <p id="subtitle">The best website ever!</p>
  </section>
</body>
</html>
```

The code in Listing 6-139 is not performing any action; it is just getting a reference to the **Element** object that represents the **<p>** element and storing it in the **element** variable as soon as the document is loaded. To do something with the element, we have to work with the object's properties.

Every **Element** object automatically gets properties that reference each attribute in the element they represent. Accessing these properties, we can get or modify the values of the corresponding attributes. For example, if we read the **id** property in the object stored in the **element** variable, we get the string "subtitle".

***Listing 6-140:*** *Reading the element's attributes from JavaScript*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function initiate() {
      var element = document.getElementById("subtitle");
      alert("The id is: " + element.id);  // "The id is: subtitle"
    }
  </script>
</head>
<body onload="initiate()">
  <section>
    <h1>Website</h1>
    <p id="subtitle">The best website ever!</p>
  </section>
</body>
</html>
```

In these examples, we have accessed the element with the **getElementById()** method because the **<p>** element in our document has an **id** attribute, but this may not be the case all the time. If the **id** attribute is not present or we want to get a list of elements that share similar characteristics, we can use the rest of the methods provided by the **Document** object. For instance, we can get a list of all the **<p>** elements in the document with the **getElementsByTagName()** method.

***Listing 6-141:*** *Accessing elements by keyword*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function initiate() {
      var list = document.getElementsByTagName("p");
      for (var f = 0; f < list.length; f++) {
        var element = list[f];
        alert("The id is: " + element.id);
      }
    }
  </script>
</head>
<body onload="initiate()">
  <section>
    <h1>Website</h1>
    <p id="subtitle">The best website ever!</p>
  </section>
</body>
</html>
```

The **getElementsByTagName()** method returns an array with references to all the elements which keyword is equal to the value provided between parentheses. In Listing 6-141, the attribute was defined with the string "p" to obtain a list of all the **<p>** elements in the document. After the references

are fetched, we access each element with a **for** loop and show the value of their **id** attributes on the screen.

If we know the position of the element we want to access, we can provide the index ourselves. In our example, there is only one **<p>** element, so the reference to this element will be at the position 0 in the array.

***Listing 6-142:*** *Accessing an element by keyword*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function initiate() {
      var list = document.getElementsByTagName("p");
      var element = list[0];
      alert("The id is: " + element.id);
    }
  </script>
</head>
<body onload="initiate()">
  <section>
    <h1>Website</h1>
    <p id="subtitle">The best website ever!</p>
  </section>
</body>
</html>
```

A better method to find elements is **querySelector()**. This method searches for an element using a CSS selector. The advantage is that we can exploit the power of CSS selectors to find the right element. In the following example, we use the **querySelector()** method to look for **<p>** elements that are direct children of a **<section>** element.

***Listing 6-143:*** *Finding an element with the* querySelector() *method*

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function initiate() {
      var element = document.querySelector("section > p");
      alert("The id is: " + element.id);
    }
  </script>
</head>
<body onload="initiate()">
  <section>
    <h1>Website</h1>
    <p id="subtitle">The best website ever!</p>
  </section>
</body>
</html>
```

**The Basics:** The **querySelector()** method returns only a reference to the first element it finds. If we want to get a list of all the elements that match the selector, we have to use the **querySelectorAll()** method.

The methods we have just studied look for elements in the entire document, but we can narrow the search by looking inside an element. For this purpose, **Element** objects also include their own versions of methods like **getElementsByTagName()** and **querySelector()**. For example, we can search for **<p>** elements inside **<section>** elements with a particular ID.

***Listing 6-144:*** *Searching for an element inside another element*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function initiate() {
      var mainelement = document.getElementById("mainsection");
```

```
    var list = mainelement.getElementsByTagName("p");
    var element = list[0];
    alert("The id is: " + element.id);
   }
  </script>
</head>
<body onload="initiate()">
  <section id="mainsection">
    <h1>Website</h1>
    <p id="subtitle">The best website ever!</p>
  </section>
</body>
</html>
```

The code in Listing 6-144 gets a reference to the element identified with the string "mainsection" and then calls the **getElementsByTagName()** method to find **<p>** elements inside that element. Because we only have one **<p>** element inside the **<section>** element, we read the reference at index 0 and show the value of its **id** attribute on the screen.

# Element Objects

Getting a reference to access an element and read its attributes may sometimes be useful, but what makes JavaScript a dynamic language is the possibility to modify those elements and the entire document. For this purpose, **Element** objects also contain properties to define and modify the styles of an element and its content. One of these properties is **style**, which contains an object called **Styles** that in turn provides properties for modifying the element's styles.

The names of the styles in JavaScript are not the same as in CSS. There was no consensus on this regard, so although we are able to provide the same values for the properties, we have to learn their names in JavaScript. The following are the most frequently used.

**color**—This property sets the color of the element's content.

**background**—This property sets the element's background styles. We can also work with each style individually using the associated properties **backgroundColor**, **backgroundImage**, **backgroundRepeat**, **backgroundPosition** and **backgroundAttachment**.

**border**—This property sets the styles for the element's border. We can modify each style individually with the properties **borderColor**, **borderStyle**, and **borderWidth**, or affect each border using the associated properties **borderTop** (**borderTopColor**, **borderTopStyle**, and **borderTopWidth**), **borderBottom** (**borderBottomColor**, **borderBottomStyle**, and **borderBottomWidth**), **borderLeft** (**borderLeftColor**, **borderLeftStyle**, and **borderLeftWidth**), and **borderRight** (**borderRightColor**, **borderRightStyle**, and **borderRightWidth**).

**margin**—This property sets the element's margin. We can also use the associated properties **marginBottom**, **marginLeft**, **marginRight**, and **marginTop**.

**padding**—This property sets the element's padding. We can also use the associated properties **paddingBottom**, **paddingLeft**, **paddingRight**, and **paddingTop**.

**width**—This property sets the element's width. There are two associated properties to set the element's maximum and minimum width: **maxWidth** and **minWidth**.

**height**—This property sets the element's height. There are two associated properties to set the element's maximum and minimum height: **maxHeight** and **minHeight**.

**visibility**—This property determines whether the element is visible or not.

**display**—This property defines the type of box used to render the element.

**position**—This property defines the type of positioning used to position the element.

**top**—This property specifies the distance between the element's top margin and the top margin of its container.

**bottom**—This property specifies the distance between the element's bottom margin and the bottom margin of its container.

**left**—This property specifies the distance between the element's left margin and the left margin of its container.

**right**—This property specifies the distance between the element's right margin and the right margin of its container.

**cssFloat**—This property allows an element to float to one side or another.

**clear**—This property recovers the document's normal flow, stopping elements from floating to the left, right, or both sides.

**overflow**—This property specifies how the content that overflows its container's box is going to be shown.

**zIndex**—This property defines an index that determines the position of the element in the **z** axis.

**font**—This property sets the element's font styles. We can also set the styles individually using the associated properties **fontFamily**, **fontSize**, **fontStyle**, **fontVariant** and **fontWeight**.

**textAlign**—This property aligns the text inside the element.

**verticalAlign**—This property aligns inline elements vertically.

**textDecoration**—This property highlights the text with a line. We can also set the styles individually assigning the values **true** or **false** to the properties **textDecorationBlink**, **textDecorationLineThrough**, **textDecorationNone**, **textDecorationOverline**, and **textDecorationUnderline**.

Modifying the values of these properties is straightforward. We have to get a reference to the **Element** object that represents the element we want to modify, as we did in previous examples, and then assign a new value to the property of the **Styles** object we want to change. The only difference with CSS, besides some of the names of the properties, is that the values have to be assigned between quotes.

***Listing 6-145:*** *Assigning new styles to an element*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function initiate() {
      var element = document.getElementById("subtitle");
      element.style.width = "300px";
      element.style.border = "1px solid #FF0000";
      element.style.padding = "20px";
    }
  </script>
</head>
<body onload="initiate()">
  <section>
    <h1>Website</h1>
    <p id="subtitle">The best website ever!</p>
  </section>
</body>
</html>
```

The code in Listing 6-145 assigns a width of 300 pixels, a red border of 1 pixel, and a padding of 20 pixels to the **<p>** element in the document. The result is shown in Figure 6-4.


***Figure 6-4:*** *Styles assigned from JavaScript*

The properties of the **Styles** object are independent of the CSS styles assigned to the document. If we have not previously assigned a value to one of these properties from JavaScript, the value returned is an empty string. To provide some information about the element, **Element** objects include additional properties. The following are the most frequently used.

**clientWidth**—This property returns the width of the element, including the padding.

**clientHeight**—This property returns the height of the element, including the padding.

**offsetTop**—This property returns the number of pixels the element is displaced from the top of its container.

**offsetLeft**—This property returns the number of pixels the element is displaced from the left side of its container.

**offsetWidth**—This property returns the width of the element, including the padding and the border.

**offsetHeight**—This property returns the height of the element, including the padding and the border.

**scrollTop**—This property returns the number of pixels the content of the element is scrolled to the top.

**scrollLeft**—This property returns the number of pixels the content of the element is scrolled to the left.

**scrollWidth**—This property returns the width of the element's content.

**scrollHeight**—This property returns the height of the element's content.

These are read-only properties, but we can get the value we need by reading these properties and then use the properties of the **Styles** object to assign a new one.

***Listing 6-146:*** *Reading CSS styles from JavaScript*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <style>
    #subtitle {
      width: 300px;
      padding: 20px;
```

```
    border: 1px solid #FF0000;
   }
  </style>
  <script>
   function initiate() {
    var element = document.getElementById("subtitle");
    var width = element.clientWidth;
    width = width + 100;
    element.style.width = width + "px";
   }
  </script>
</head>
<body onload="initiate()">
  <section>
   <h1>Website</h1>
   <p id="subtitle">The best website ever!</p>
  </section>
</body>
</html>
```

In this example, we assign the styles of the **<p>** element from CSS and then modify its width from JavaScript. The current width is obtained from the **clientWidth** property, but because this property is read-only, the new value has to be assigned to the **width** property of the **Styles** object (Notice that the value assigned to the property must be a string with the units "px" at the end). After the code is executed, the **<p>** element has a width of 400 pixels.


***Figure 6-5:*** *Styles modified from JavaScript*

It is not common to modify the styles of an element individually, as we did in these examples. Usually, elements are styled by a group of CSS properties assigned through the **class** attribute. As we explained in Chapter

3, these rules are called *classes*. Classes are permanently defined in CSS Style Sheets, but **Element** objects include the following properties to assign a different class to an element and therefore change its styles all at once.

**className**—This property sets or returns the value of the **class** attribute.

**classList**—This property returns an array with the list of the classes assigned to the element.

The array returned by the **classList** property is of type **DOMTokenList**, which includes the following methods to modify the classes on the list.

**add(**class**)**—This method adds another class to the element.

**remove(**class**)**—This method removes a class from the element.

**toggle(**class**)**—This method adds or removes a class depending on the current status. If the class was already assigned to the element, then it is removed. Otherwise, the class is added to the element.

**contains(**class**)**—This method detects whether the class was assigned to the element or not and returns **true** or **false** accordingly.

The easiest way to replace the class of an element is assigning a new value to the **className** property.

***Listing 6-147:*** *Replacing the element's class*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <style>
    .supercolor {
      background: #0099EE;
    }
    .blackcolor {
      background: #000000;
```

```
    }
  </style>
  <script>
    function changecolor() {
      var element = document.getElementById("subtitle");
      element.className = "blackcolor";
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <p id="subtitle" class="supercolor" onclick="changecolor()">The best
website ever!</p>
  </section>
</body>
</html>
```

In the code in Listing 6-147, we have declared two classes: **supercolor** and **blackcolor**. Both classes define the background color of the element. By default, the **supercolor** class is assigned to the **<p>** element, which gives the element a blue background, but when the **changecolor()** function is executed, that class is replaced with the **blackcolor** class and a black color is assigned to the background (notice that this time we execute the function when the user clicks on the element, not when the document finishes loading).

As mentioned in Chapter 3, multiple classes may be assigned to an element. When this is the case, instead of the **className** property it is better to apply the methods of the **classList** property. The following example implements the **contains()** method to detect whether a class was already assigned to an element and adds it or removes it, depending on the current state.

***Listing 6-148:*** *Turning classes on and off*

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <meta charset="utf-8">
  <title>JavaScript</title>
  <style>
    .supercolor {
      background: #000000;
    }
  </style>
  <script>
    function changecolor() {
      var element = document.getElementById("subtitle");
      if (element.classList.contains("supercolor")) {
        element.classList.remove("supercolor");
      } else {
        element.classList.add("supercolor");
      }
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <p id="subtitle" class="supercolor" onclick="changecolor()">The best
website ever!</p>
  </section>
</body>
</html>
```

With the code in Listing 6-148, every time the user clicks on the **<p>** element, the styles change, going from having a background color to no background. The same may be achieved with the **toggle()** method. This method checks the status of the element and adds the class if it was not assigned before or removes it otherwise.

***Listing 6-149:*** *Turning classes on and off with the* toggle() *method*

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <meta charset="utf-8">
  <title>JavaScript</title>
  <style>
    .supercolor {
      background: #000000;
    }
  </style>
  <script>
    function changecolor() {
      var element = document.getElementById("subtitle");
      element.classList.toggle("supercolor");
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <p id="subtitle" class="supercolor" onclick="changecolor()">The best
website ever!</p>
  </section>
</body>
</html>
```

The **toogle()** method simplifies our work. We do not have to check whether the class exists or not, the method does the work for us and adds the class or removes it depending on the current state.

**Do It Yourself:** Create a new HTML file with the document you want to try. Open the document in your browser and click on the area occupied by the **<p>** element. You should see the background of that element changing colors.

Besides the styles of an element, we can also modify its content. The following are the properties and methods provided by **Element** objects for this purpose.

**innerHTML**—This property sets or returns the content of an element.

**outerHTML**—This property sets or returns an element and its content. Unlike **innerHTML**, this property replaces not only the content but also the element.

**insertAdjacentHTML(**location, content**)**—This method inserts content in a location according to the value of the **location** attribute. The values available are **beforebegin** (before the element), **afterbegin** (inside the element, before its first child), **beforeend** (inside the element, after its last child), and **afterend** (after the element).

The easiest way to replace the content of an element is with the **innerHTML** property. By assigning a new value to this property, the current content is replaced with the new one. The following example replaces the content of the **<p>** element with the string "This is my website" when the element is clicked.

***Listing 6-150:*** *Assigning content to an element*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function changecontent() {
      var element = document.getElementById("subtitle");
      element.innerHTML = "This is my website";
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <p id="subtitle" onclick="changecontent()">The best website ever!</p>
  </section>
</body>
</html>
```

The **innerHTML** property is used not only to assign new content but also to read and process the current content. The following example reads the content of an element, adds a string at the end, and assigns the result back to the element.

***Listing 6-151:*** *Modifying the content of an element*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function changecontent() {
      var element = document.getElementById("subtitle");
      var text = element.innerHTML + " We are the best!";
      element.innerHTML = text;
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <p id="subtitle" onclick="changecontent()">The best website ever!</p>
  </section>
</body>
</html>
```

Besides text, the **innerHTML** property can also process HTML code. When HTML code is assigned to these properties, the code is interpreted, and the result is shown on the screen.

***Listing 6-152:*** *Inserting HTML code into the document*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
```

```
  <script>
    function addElement() {
      var element = document.querySelector("section");
      element.innerHTML = "<p>This is a new text</p>";
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <button type="button" onclick="addElement()">Add Content</button>
  </section>
</body>
</html>
```

The code in Listing 6-152 gets a reference to the first **<section>** element in the document and replaces its content with a **<p>** element. From then on, the user will only see the text of the **<p>** element on the screen.

If we do not want to replace the entire content of an element but just add some new content, we can use the **insertAdjacentHTML()** method. This method can add content before or after the current content and also outside the element, depending on the value assigned to the first attribute.

***Listing 6-153:*** *Adding HTML code inside an element*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function addElement() {
      var element = document.querySelector("section");
      element.insertAdjacentHTML("beforeend", "<p>This is a new
text</p>");
    }
  </script>
</head>
```

```
<body>
  <section>
    <h1>Website</h1>
    <button type="button" onclick="addElement()">Add Content</button>
  </section>
</body>
</html>
```

The **insertAdjacentHTML()** method adds content to the document, but without affecting the previous content. When we press the button in the document of Listing 6-153, the JavaScript code adds a **<p>** element below the **<button>** element (before the end of the **<section>** element). The result is shown in Figure 6-6.


***Figure 6-6:*** *Content added to an element*

# Creating Element Objects

When HTML code is added to the document through properties and methods like **innerHTML** or **insertAdjacentHTML()**, the browser parses the document and generates the **Element** objects necessary to represent the new elements. Although this is an accepted procedure to modify the structure of a document, the **Document** object includes methods to work directly with **Element** objects.

**createElement(**keyword**)**—This method creates a new **Element** object of the type specified by the **keyword** attribute.

**appendChild(**element**)**—This method inserts the element represented by an **Element** object as the child of an existent element in the document.

**removeChild(**element**)**—This method removes a child from an element. The attribute must be a reference to the child to be removed.

If our intention is to create a new **Element** object to add an element to the document, we first have to create the object with the **createElement()** method, and then use this object to add the element to the document with the **appendChild()** method.

***Listing 6-154:*** *Creating* Element *objects*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function addElement() {
      var element = document.querySelector("section");
      var newelement = document.createElement("p");
      element.appendChild(newelement);
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <button type="button" onclick="addElement()">Add Element</button>
  </section>
</body>
</html>
```

The code in Listing 6-154 adds a **<p>** element at the end of the **<section>** element, but the element has no content, so we cannot see it on the screen. If we want to define the element's content, we can assign a new value to its **innerHTML** property. The **Element** objects returned by the **createElement()** method are the same as the objects the browser creates automatically to represent the document and therefore we can modify their properties to assign new styles or define their content. The following code assigns content to an **Element** object before adding the element to the document.

***Listing 6-155:*** *Adding content to an* Element *object*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function addElement() {
      var element = document.querySelector("section");
      var newelement = document.createElement("p");
      newelement.innerHTML = "This is a new element";
      element.appendChild(newelement);
    }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <button type="button" onclick="addElement()">Add Element</button>
  </section>
</body>
</html>
```


***Figure 6-7:*** *Element added to the document*

**The Basics:** There is no much difference between adding elements with the **innerHTML** property or these methods, but the **createElement()** method becomes useful when working with APIs that require **Element** objects to process information, such as when we have to process an image or a video that is not going to be shown on the screen but sent to a server or stored on the hard drive. We will learn more about APIs in this

chapter and study practical applications of the **createElement()** method later.

# 6.5 Events

As we have already seen, HTML provides attributes to execute JavaScript code when an event occurs. In recent examples, we have implemented the **onload** attribute to execute a function when the browser finishes loading the document and the **onclick** attribute that executes JavaScript code when the user clicks on the element. What we did not mention is that these attributes, like any other attribute, can be set from JavaScript. This is because, as we have also seen, element's attributes are turned into properties inside **Element** objects, and therefore we can define their values from JavaScript code. For instance, if we want to respond to the **click** event, we have to define the **onclick** property on the element.

***Listing 6-156:*** *Defining event attributes from JavaScript code*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function addClick() {
      var element = document.querySelector("section > button");
      element.onclick = showmessage;
    }
    function showmessage() {
      alert("You pressed the button");
    }
    window.onload = addClick;
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <button type="button">Show</button>
  </section>
</body>
```

</html>

The document of Listing 6-156 does not include any event attribute inside the elements; they are all declared in the JavaScript code. Two attributes are defined in this case, the **onload** attribute of the **Window** object and the **onclick** attribute of the **<button>** element. When the document is loaded, the **load** event is fired, and the **addClick()** function is executed. In this function, we get a reference to the **<button>** element and define its **onclick** attribute to execute the **showmessage()** function when the button is pressed. With this information, the document is ready to work. If the user presses the button, a message is shown on the screen.

**The Basics:** Declaring the **onload** attribute in the **<body>** element or the **Window** object presents no difference whatsoever, but because we should always separate the JavaScript code from the HTML document and from the code it is easier to set up the attribute for the **Window** object, this is the recommended practice.

# The addEventListener() Method

The use of event attributes in HTML elements is not recommended because they defeat the main purpose of HTML5 that is to provide a specific task to each language. HTML should define the structure of the document, CSS its presentation, and JavaScript the functionality. But the definition of these attributes from JavaScript code, as we have done in the previous example, is not completely compatible with all browsers and present some limitations, so it is not recommended either. For these reasons, new methods were defined for the **Window** object to manage and respond to events.

**addEventListener(**event, listener, capture**)**—This method prepares an element to respond to an event. The first attribute is the event's name (without the prefix **on**), the second attribute is a reference to the function that will respond to the event (called *listener*), and the third attribute is a Boolean value that determines whether the event is captured by the element or it propagates to other elements (usually ignored or declared as **false**).

**removeEventListener(**event, listener**)**—This method removes a listener from an element.

The names of the events required by these methods are not the same as the names of the attributes. Actually, the names of the attributes are made by adding the prefix **on** to the real name of the event. For example, the **onclick** attribute represents the **click** event. In the same way, we have the **load** event (**onload**), the **mouseover** event (**onmouseover**), and so on. When we use the **addEventListener()** method to make an element respond to an event, we have to provide the real name of the event between quotes, as in the following example.

***Listing 6-157:*** *Listening to events with the* addEventListener() *method*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function addClick() {
      var element = document.querySelector("section > button");
      element.addEventListener("click", showmessage);
    }
    function showmessage() {
      alert("You pressed the button");
    }
    window.addEventListener("load", addClick);
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <button type="button">Show</button>
  </section>
</body>
</html>
```

The code in Listing 6-157 is the same as the previous example, but now the **addEventListener()** method is used to add event listeners to the **Window** object and the **<button>** element.

# Event Objects

Every function that responds to an event receives an object that contains information about the event. Although some events generate their own specific objects, there is an object called **Event** that is common to every event. The following are some of its properties and methods.

**target**—This property returns a reference to the object that received the event (usually an **Element** object).

**type**—This property returns a string with the name of the event.

**preventDefault()**—This method cancels the event to prevent the system from performing tasks by default (see Chapter 17, Listing 17-3).

**stopPropagation()**—This method stops the propagation of the event to other elements, so only the first element that receives the event can process it (usually applied to elements that overlap and can respond to the same event).

The **Event** object is automatically sent to the function as an attribute, and therefore we have to declare a parameter that will receive this value. The name of the parameter is irrelevant, but by convention it is defined as **e** or **event**. In the following example, we use the **Event** object to identify the element that was clicked by the user.

***Listing 6-158:*** *Using* Event *objects*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function addClicks() {
```

```
    var list = document.querySelectorAll("section > p");
    for (var f = 0; f < list.length; f++) {
      var element = list[f];
      element.addEventListener("click", changecolor);
    }
  }
  function changecolor(event) {
    var element = event.target;
    element.style.backgroundColor = "#999999";
  }
  window.addEventListener("load", addClicks);
 </script>
</head>
<body>
 <section>
  <h1>Website</h1>
  <p>Message number 1</p>
  <p>Message number 2</p>
  <p>Message number 3</p>
 </section>
</body>
</html>
```

The code in Listing 6-158 adds a listener for the **click** event to every **<p>** element inside the **<section>** element of our document, but they are all handled by the same function. To identify which element was clicked from the function, we read the **target** property of the **Event** object. This property returns a reference to the **Element** object that represents the element that was clicked. Using this reference, we modify the element's background. In consequence, every time the user clicks on the area occupied by a **<p>** element, the element's background turns gray.

***Figure 6-8:*** *Only the element that received the event is affected*

The **Event** object is automatically passed to the function when the function is called. If we want to send our own values along with this object, we can handle the event with an anonymous function. The anonymous function only receives the **Event** object, but from inside this function we can call the function that responds to the event with any attributes we want.

***Listing 6-159:*** *Responding to an event with an anonymous function*

```
<script>
  function addClicks() {
    var list = document.querySelectorAll("section > p");
    for (var f = 0; f < list.length; f++) {
      var element = list[f];
      element.addEventListener("click", function(event) {
         var ourvalue = 125;
         changecolor(event, ourvalue);
      });
    }
  }
  function changecolor(event, ourvalue) {
    var element = event.target;
    element.innerHTML = "Value " + ourvalue;
  }
  window.addEventListener("load", addClicks);
</script>
```

The code in Listing 6-159 replaces the code from the previous example. In this code, instead of calling the **changecolor()** function directly, we first execute an anonymous function. This function receives the **Event** object, sets another variable called **ourvalue** with the value 125, and then calls the **changecolor()** function with both values. With these values, the **changecolor()** function modifies the element's content.


***Figure 6-9:*** *The element is modified with the values received by the function*

**Do It Yourself:** Create a new HTML file with the document of Listing 6-158. Open the document in your browser and click on the area occupied by the **<p>** elements. The background color of the element you clicked should change to gray. Update the JavaScript code with the code in Listing 6-159 and open the document again or refresh the page. Click on an element. The content of that element should be replaced with the string "Value 125", as illustrated in Figure 6-9.

In Listing 6-159, the value passed to the **changecolor()** function along with the **Event** object was an absolute value (125), but a problem arises when we try to pass the value of a variable. In that case, because the instructions inside the anonymous function are not processed until the event occurs, the function will contain a reference to the variable instead of its current value. The issue becomes evident when we work with values generated by a loop.

***Listing 6-160:*** *Passing values to a listener*

```
<script>
  function addClicks() {
    var list = document.querySelectorAll("section > p");
    for (var f = 0; f < list.length; f++) {
      var element = list[f];
      element.addEventListener("click", function(event) {
         var ourvalue = f;
         changecolor(event, ourvalue);
      });
    }
  }
  function changecolor(event, ourvalue) {
```

```
    var element = event.target;
    element.innerHTML = "Value " + ourvalue;
  }
  window.addEventListener("load", addClicks);
</script>
```

In this example, instead of the value 125, we assign the **f** variable to the **ourvalue** variable, but because the instruction is not processed until the user clicks on the element, the system assigns a reference of the **f** variable to **ourvalue**, not its value. This means that the system is going to read the **f** variable and assign its value to the **ourvalue** variable only when the **click** event is fired, and by that time the **for** loop will already be over and the current value of **f** will be 3 (after the loop is over, the final value of **f** is 3 because there are three **<p>** elements inside the **<section>** element). This means that the value this code passes to the **changecolor()** function is always 3, no matter what element is clicked.


***Figure 6-10:*** *The value passed to the function is always 3*

This issue may be solved by combining two anonymous functions. One function is executed right away, and the other is returned by the first one. The main function receives the current value of **f**, stores it in another variable, and then returns a second anonymous function with these values. The anonymous function returned is the one that is going to be executed when the event occurs.

***Listing 6-161:*** *Passing values to a listener with anonymous functions*

```
<script>
  function addClicks() {
    var list = document.querySelectorAll("section > p");
    for (var f = 0; f < list.length; f++) {
      var element = list[f];
```

```
    element.addEventListener("click", function(x) {
       return function(event) {
         var ourvalue = x;
         changecolor(event, ourvalue);
       };
    }(f));
   }
  }
  function changecolor(event, ourvalue) {
   var element = event.target;
   element.innerHTML = "Value " + ourvalue;
  }
  window.addEventListener("load", addClicks);
</script>
```

The main anonymous function is executed when the **addEventListener()** method is processed. It receives the current value of the **f** variable (see the parentheses at the end) and puts that value in the **x** variable. Then, the function returns a second anonymous function that assigns the value of the variable **x** to **ourvalue** and calls the **changecolor()** function to respond to the event. Because the interpreter reads the value of **f** in every cycle of the loop to send it to the main anonymous function, the anonymous function returned always works with a different value. For the first **<p>** element, the value will be 0 (it is the first element on the list and **f** starts counting from 0), the second element gets 1, and the third element gets 2. Now the code produces a different content for each element.


***Figure 6-11:*** *The value is different for each element*

Some events generate unique values that are passed to the listener for processing. These events work with their own type of objects that inherit

from the **Event** object. For instance, mouse events send a **MouseEvent** object to the function. The following are some of its properties.

**button**—This property returns an integer that represents the button that was pressed (0 = left button).

**ctrlKey**—This property returns a Boolean value that determines if the Control key was pressed when the event occurred.

**altKey**—This property returns a Boolean value that determines if the Alt (Option) key was pressed when the event occurred.

**shiftKey**—This property returns a Boolean value that determines if the Shift key was pressed when the event occurred.

**metaKey**—This property returns a Boolean value that determines if the Meta key was pressed when the event occurred (the Meta key is the Windows key on Windows keyboards or the Command key on Macintosh keyboards).

**clientX**—This property returns the horizontal coordinate where the mouse was located when the event occurred. The coordinate is returned in pixels, and it is relative to the area occupied by the window.

**clientY**—This property returns the vertical coordinate where the mouse was located when the event occurred. The coordinate is returned in pixels, and it is relative to the area occupied by the window.

**offsetX**—This property returns the horizontal coordinate where the mouse was located when the event occurred. The coordinate is returned in pixels, and it is relative to the area occupied by the element that received the event.

**offsetY**—This property returns the vertical coordinate where the mouse was located when the event occurred. The coordinate is returned in pixels, and it is relative to the area occupied by the element that received the event.

**pageX**—This property returns the horizontal coordinate where the mouse was located when the event occurred. The coordinate is returned in pixels, and it is relative to the document. It takes scrolling into account to calculate the value.

**pageY**—This property returns the vertical coordinate where the mouse was located when the event occurred. The coordinate is returned in pixels, and it is relative to the document. It takes scrolling into account to calculate the value.

**screenX**—This property returns the horizontal coordinate where the mouse was located when the event occurred. The coordinate is returned in pixels, and it is relative to the screen.

**screenY**—This property returns the vertical coordinate where the mouse was located when the event occurred. The coordinate is returned in pixels, and it is relative to the screen.

**movementX**—This property returns the difference between the previous and the current positions of the mouse in the horizontal axis. The value is returned in pixels, and it is relative to the screen.

**movementY**—This property returns the difference between the previous and the current positions of the mouse in the vertical axis. The value is returned in pixels, and it is relative to the screen.

In Chapter 3, we explained that the screen is divided into rows and columns of pixels and that computers use a coordinate system to identify the position of each pixel (see Figure 3-50). What we did not mention is that the same coordinate system is applied to every area, including the screen, the browser window, and the HTML elements, so every one of them has its own origin (their coordinates always start at 0, 0 at the top-left corner). The **MouseEvent** object can give us the coordinates of the mouse when the event occurred, but because each area has its own coordinate system, different values are reported. For example, the **clientX** and **clientY** properties contain the coordinates of the mouse in the coordinate system of the window, but the **offsetX** and **offsetY** properties report the position in the coordinate system of the element that received the event. The following example detects a click and shows the position of the mouse inside the window using the **clientX** and **clientY** properties.

***Listing 6-162:*** *Reporting the position of the mouse*

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function showposition(event) {
      alert("Position: " + event.clientX + " / " + event.clientY);
    }
    window.addEventListener("click", showposition);
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <p>This is my website</p>
  </section>
</body>
</html>
```

The code in Listing 6-162 is listening to the **click** event on the **Window** object, so a click anywhere on the window will execute the **showposition()** function and show the mouse's position on the screen. This function reads the **clientX** and **clientY** properties to get the position of the mouse relative to the screen. If we want to get the position relative to an element, we have to listen to the event on the element and read the properties **offsetX** and **offsetY**. The following example uses these properties to create a progress bar which size is determined by the current position of the mouse when it is over the element.

***Listing 6-163:*** *Calculating the position of the mouse on an element*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <style>
    #container {
      width: 500px;
```

```
      height: 40px;
      padding: 10px;
      border: 1px solid #999999;
    }
    #progressbar {
      width: 0px;
      height: 40px;
      background-color: #000099;
    }
  </style>
  <script>
    function initiate() {
      var element = document.getElementById("container");
      element.addEventListener("mousemove", movebar);
    }
    function movebar(event) {
      var widthbar = event.offsetX - 10;
      if (widthbar < 0) {
        widthbar = 0;
      } else if (widthbar > 500) {
        widthbar = 500;
      }
      var element = document.getElementById("progressbar");
      element.style.width = widthbar + "px";
    }
    window.addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
    <h1>Level</h1>
    <div id="container">
      <div id="progressbar"></div>
    </div>
  </section>
</body>
</html>
```

The document of Listing 6-163 includes two ** ** elements, one inside another, to recreate a progress bar. The ** ** element identified with the name **container** works as a container to establish the boundaries of the bar, and the one identified with the name **progressbar** represents the bar itself. The purpose of the application is to let the user set the size of the bar with the mouse, so we have to listen to the **mousemove** event to follow every move of the mouse and read the **offsetX** property to calculate the size of the bar based on the current position.

Because the area occupied by the **progressbar** element will always be different (it is defined with a width of 0 pixels by default), we have to listen to the **mousemove** event from the **container** element. This requires the code to adjust the values returned by the **offsetX** property to the position of the **progressbar** element. The container element includes a padding of 10 pixels, so the bar will be 10 pixels from the left side of its container, and that's the number we must subtract from the current value of **offsetX** to determine the width of the bar (**event.offsetX - 10**). If the mouse is 20 pixels from the left side of the container, it means that it is 10 pixels from the left side of the bar, so the bar should have a width of 10 pixels. This works until the mouse is over the paddings of the **container** element. When the mouse is located over the left padding, lets say at the position 5, the operation returns the value -5, but we cannot set a negative size for the bar. Something similar happens when the mouse is located over the right padding. In this case, the bar will exceed the maximum size of the container. The issues are solved by the **if** instructions. If the new width is less than 0, we set it as 0, and if it is greater than 500, we set it as 500. With the limits established, we get a reference to the **progressbar** element and modify its **width** property to set the new width.


***Figure 6-12:*** *Progress bar*

**Do It Yourself:** Create a new HTML file with the document of Listing 6-163 and open the document in your browser. Move the mouse over the

**container** element. You should see the **progressbar** element expand or shrink to follow the mouse, as shown in Figure 6-12.

Other events that produce their own **Event** object are those related to the keyboard (**keypress**, **keydown**, and **keyup**). The object is of type **KeyboardEvent**, and it includes the following properties.

**key**—This property returns a string that identifies the key or keys that triggered the event.

**ctrlKey**—This property returns a Boolean value that determines if the Control key was pressed when the event occurred.

**altKey**—This property returns a Boolean value that determines if the Alt (Option) key was pressed when the event occurred.

**shiftKey**—This property returns a Boolean value that determines if the Shift key was pressed when the event occurred.

**metaKey**—This property returns a Boolean value that determines if the Meta key was pressed when the event occurred (the Meta key is the Windows key on Windows keyboards or the Command key on Macintosh keyboards).

**repeat**—This property returns a Boolean value that determines if the user is holding down the key.

The most important property of the **KeyboardEvent** object is **key**. This property returns a string that represents the key that triggered the event. Common keys like numbers and letters produce a string with the same characters in lowercase letters. For example, if we want to check if the key pressed was the letter A, we have to compare the value with the string "a". The following example compares the value returned by the **key** property against a series of numbers to check whether the key pressed was 0, 1, 2, 3, 4, or 5.

***Listing 6-164:*** *Detecting the key pressed*

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
<meta charset="utf-8">
<title>JavaScript</title>
<style>
  section {
    text-align: center;
  }
  #block {
    display: inline-block;
    width: 150px;
    height: 150px;
    margin-top: 100px;
    background-color: #990000;
  }
</style>
<script>
  function detectkey(event) {
    var element = document.getElementById("block");
    var code = event.key;
    switch (code) {
      case "0":
        element.style.backgroundColor = "#990000";
        break;
      case "1":
        element.style.backgroundColor = "#009900";
        break;
      case "2":
        element.style.backgroundColor = "#000099";
        break;
      case "3":
        element.style.backgroundColor = "#999900";
        break;
      case "4":
        element.style.backgroundColor = "#009999";
        break;
      case "5":
        element.style.backgroundColor = "#990099";
        break;
```

```
    }
  }
  window.addEventListener("keydown", detectkey);
 </script>
</head>
<body>
 <section>
  <div id="block"></div>
 </section>
</body>
</html>
```

The document of Listing 6-164 draws a red block at the center of the window. To respond to the keyboard, we add a listener for the **keydown** event to the window (the **keydown** event is fired for every key, while the **keypress** event is only fired for common keys, such as letters and numbers). Every time a key is pressed, we read the value of the **key** property and compare it to a series of numbers. If a match is found, we assign a different color to the element's background. Otherwise, the code does nothing.

Besides the common keys, the **key** property also reports special keys like Alt or Control. The strings for the most common keys are "Alt", "Control", "Shift", "Meta", "Enter", "Tab", "Backspace", "Delete", "Escape", " " (space bar), "ArrowUp", "ArrowDown", "ArrowLeft", "ArrowRight", "Home", "End", "PageUp", and "PageDown". The following code detects the arrow keys to change the size of the block created by the previous example.

***Listing 6-165:*** *Detecting special keys*

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="utf-8">
 <title>JavaScript</title>
 <style>
  section {
   text-align: center;
  }
  #block {
```

```
      display: inline-block;
      width: 150px;
      height: 150px;
      margin-top: 100px;
      background-color: #990000;
    }
  </style>
  <script>
    function detectkey(event) {
      var element = document.getElementById("block");
      var size = element.clientWidth;
      var code = event.key;
      switch (code) {
        case "ArrowUp":
          size += 10;
          break;
        case "ArrowDown":
          size -= 10;
          break;
      }
      if (size < 50) {
        size = 50;
      }
      element.style.width = size + "px";
      element.style.height = size + "px";
    }
    window.addEventListener("keydown", detectkey);
  </script>
</head>
<body>
  <section>
    <div id="block"></div>
  </section>
</body>
</html>
```

As we did in the example of Listing 6-146, we get the current width of the element from the **clientWidth** property and then assign the new value to the **width** property of the **Styles** object. The new value depends on the key that was pressed. If the key was the up arrow, we increase the size in 10 pixels, but if the key was the down arrow, the size is reduced by 10 pixels. In the end, we check this value to make sure the block is never smaller than 50 pixels.

**Do It Yourself:** Create a new HTML file with the document of Listing 6-165. Open the document in your browser and press the up or down arrow. The block should expand or shrink according to the key pressed.

# 6.6 Debugging

Debugging is the process of finding and correcting errors in our code. There are different types of errors, from programming errors to logical errors, or even custom errors we generate to indicate a problem found by the code itself. Some require the use of tools to find them and figure out a solution and others just a little bit of patience and perseverance. Most of the time, determining what is wrong with our code requires reading the instructions one by one and following the logic until something does not make sense. Fortunately, browsers provide a set of tools to deal with these issues, and JavaScript includes some techniques we can implement to make our lives easier.

# Console

The most useful tool to check for errors and debug our code is the Console. Consoles are available in almost every browser but in different forms. Usually, they are opened at the bottom of the browser window and are composed of several panels detailing information of every aspect of the document, including the HTML code, CSS styles and, of course, JavaScript. The panel called *Console* is the one that displays errors and custom messages.

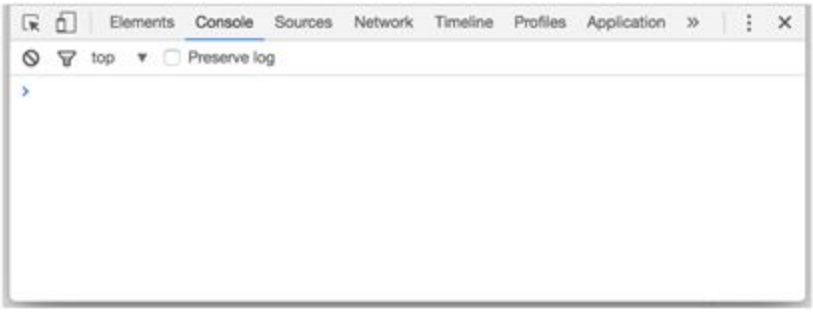


***Figure 6-13:*** *Google Chrome console*

**The Basics:** How to access this console varies from browser to browser, and even among different versions of the same browser, but the options are usually in the browser's main menu under the name Development Tools or More Tools.

The types of errors we will often see displayed on this console are programming errors. For instance, if we call a non-existent function or try to read a property that is not part of an object, it is considered a programming error and it is reported on the console.

***Listing 6-166:*** *Generating an error*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    wrongfunction();
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
  </section>
</body>
</html>
```

In Listing 6-166 we try to execute a function called **wrongfunction()** that was not previously defined. The browser finds the error and shows the message "wrongfunction is not defined" on the console to report it.

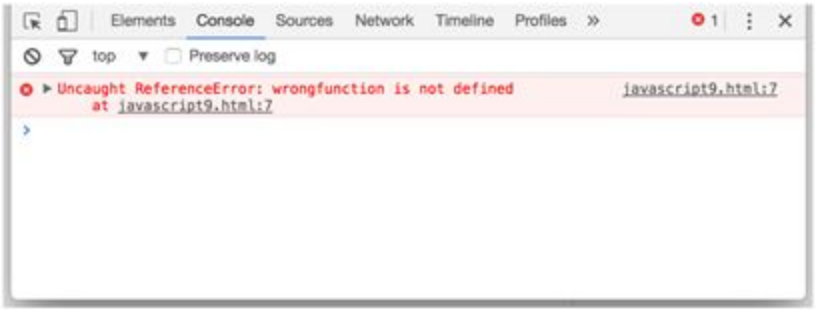


***Figure 6-14:*** *Error reported on the console*

**Do It Yourself:** Create a new HTML file with the document of Listing 6-166 and open the document in your browser. Access the browser's main menu and find the option to open the console. In Google Chrome, the menu is at the upper-right corner, and the option is called More Tools / Developer Tools. You should see the error produced by the function printed on the console, as illustrated in Figure 6-14.

# Console Object

Sometimes errors are not programming errors but logical errors. The JavaScript interpreter cannot find any errors in the code, but the application is not doing what we expected. This may be caused by a number of reasons, from an operation we forgot to make to a variable set with an incorrect value. These are the most difficult errors to detect, but there is a traditional programming technique that can help us find a solution called *breakpoints*. Breakpoints are stopping points in our code that we set to check the current state of the application. In a breakpoint, we show the current values of variables or a message letting us know that the interpreter reached that point.

Traditionally, JavaScript programmers inserted an **alert()** method in parts of the code to expose some values that could provide information to help them find the bug, but this method was not suitable for most common situations because it stops the execution of the code until the popup window is closed. Browsers simplify this process by creating an object of type **Console**. This object is assigned to the **console** property of the **Window** object and it becomes the connection between our code and the browser's console. The following are some of the methods provided by the **Console** object to manage the console.

**log(**value**)**—This method displays the value of the attribute on the console.

**assert(**condition, values**)**—This method shows on the console the values specified by the attributes if the condition specified by the first attribute is false.

**clear()**—This method clears the console. Browsers also offer a button at the top of the console with the same functionality.

The most important method in the **Console** object is **log()**. With this method, we can print a message on the console any time we want, without interrupting the execution of the code, which means that we can check the values of the variables and properties at any time and see if they match our expectations. For example, we can print on the console the values generated by a loop in every cycle to make sure that we are creating the right sequence of numbers.

***Listing 6-167:*** *Displaying messages on the console with the* log() *method*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    var list = [0, 5, 103, 24, 81];
    for(var f = 0; f < list.length; f++) {
      console.log("The current number is " + f);
    }
    console.log("The final value of f is: " + f);
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
  </section>
</body>
</html>
```

The code in Listing 6-167 calls the **log()** method to show a message in every cycle of the loop and also at the end to show the last value of the **f** variable. A total of 5 messages are printed on the console.

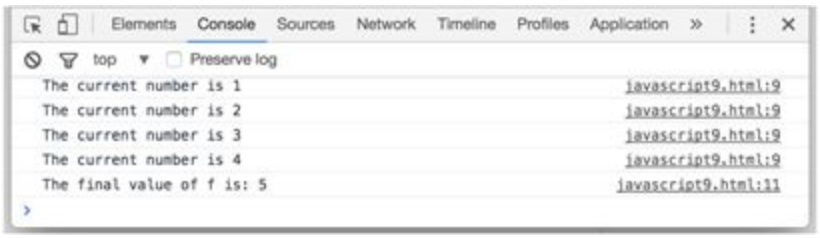


***Figure 6-15:*** *Messages on the console*

This little example illustrates the power of the **log()** method and how it can help us understand the way our code works. In this case, it demonstrates the mechanics of a **for** loop. The value of the **f** variable in the loop goes from 0 to one number less than the total number of values in the array (5), so the instruction inside the loop prints a total of 5 messages with the values 0, 1, 2, 3, and 4. This may be expected, but the **log()** method at the end of the code prints the final value of **f**, which is not 4 but 5. In the first cycle of the loop, the interpreter checks the condition with the initial value of **f**. If the condition is true, it executes the code. But the next cycle, the interpreter executes the operation assigned to the loop (**f**++) before checking the condition. If this time the condition is false, the loop is interrupted. This is the reason why the final value of **f** is 5. At the end of the loop, the value of **f** was increased once again before the condition was checked and the loop interrupted.

> **The Basics:** The **log()** method can also print objects on the console, allowing us to read the content of an unknown object and identify its properties and methods.

# Error Event

At some point, we will find that sometimes errors are not our fault. When our applications get more complex and incorporate sophisticated libraries and APIs, errors are caused by factors beyond our control, such as momentarily inaccessible resources or an unexpected change in the device running the application. In order to help the code to detect these errors and correct itself, JavaScript provides the **error** event. This event is available in several APIs, as we will see later, but also as a global event that we can listen from the **Window** object.

Like other events, the **error** event creates its own **Event** object, called **ErrorEvent**, to transmit information to the listener. This object includes the following properties.

**error**—This property returns an object with information about the error.

**message**—This property returns a string describing the error.

**lineno**—This property returns the line in the document where the error occurred.

**colno**—This property returns the column where the instruction that produced the error begins.

**filename**—This property returns the URL of the file where the error occurred.

Taking advantage of this event, we can program our code to respond to expected and unexpected errors.

***Listing 6-168:*** Responding to errors

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
    function showerror(event){
      console.log('Error: ' + event.error);
      console.log('Message: ' + event.message);
      console.log('Line: ' + event.lineno);
      console.log('Column: ' + event.colno);
      console.log('URL: ' + event.filename);
    }
    window.addEventListener('error', showerror);
    wrongfunction();
  </script>
</head>
<body>
```

```
  <section>
    <h1>Website</h1>
  </section>
</body>
</html>
```

In the code in Listing 6-168, the error is produced by the execution of a nonexistent function called **wrongfunction()**. When the browser tries to execute this function, it finds the error and triggers the **error** event to report it. To identify the error, we print messages on the console with the values of the **ErrorEvent** object's properties.

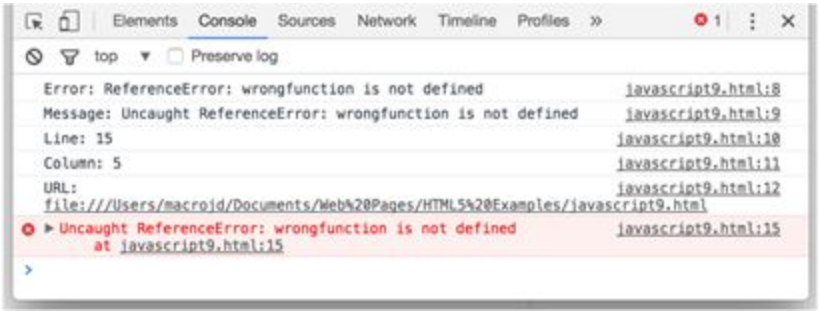


***Figure 6-16:*** *Information about an error*

# Exceptions

There are times when we know that our code could produce an error. For example, we may have a function that calculates a number from a value inserted by the user. If the value received is out of a particular range, the operation may be invalid. In computer programming, errors that can be handled by the code are usually called *exceptions*, and the process to generate an exception is called *throw*. In this terms, when we report our own errors we say that we are *throwing an exception*. JavaScript includes the following instructions to throw exceptions and catch errors.

**throw**—This instruction generates an exception.

**try**—This instruction indicates the set of instructions that should be tested for errors.

**catch**—This instruction indicates the set of instructions that should be executed if an exception occurs.

If we know that a function may produce an error, we can detect it, throw an exception with the **throw** instruction, and then respond to the exception with the combination of the **try** and **catch** instructions. The following example illustrates how this process works.

***Listing 6-169:*** Throwing exceptions

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>JavaScript</title>
  <script>
   var stock = 5;
   function sold(amount) {
     if (amount > stock) {
       var error = {
         name: "ErrorStock",
         message: "Out of stock"
       };
       throw error;
     } else {
       stock = stock - amount;
     }
   }
   try {
     sold(8);
   } catch(error) {
     console.log(error.message);
   }
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
```

```
  </section>
</body>
</html>
```

The **throw** instruction works in a similar way than the **return** instruction. It breaks the execution of the function and returns a value that is captured by the **catch** instruction. The value must be specified as an object with the properties **name** and **message**. The **name** property should have a value that identifies the exception and the **message** property should contain a string describing the error. Once we have a function that throws an exception, we have to call it from the **try catch** instructions. The syntax of these instructions is similar to the **if else** instructions. They define two blocks of code. If the instructions inside the **try** block throw an exception, the instructions in the **catch** block are executed.

In our example, we have created a function called **sold()** that keeps track of the items sold in a store. When a customer makes a purchase, we call this function with the number of items sold. The function receives this value and subtracts it from the **stock** variable. Here is when we check if this is a valid transaction. If there is not enough stock to fulfill the order, we throw an exception. In this case, the **stock** variable is initialized with the value 5, and the **sold()** function is called with the value 8, so the function throws and exception. Because the call is made inside a **try** block, the exception is captured, the instructions inside the **catch** block are executed, and the message "Out of stock" is printed on the console.

# 6.7 APIs

No matter the experience and the knowledge we have about computer programming and the programming language we use to create our applications, we cannot program the whole applications ourselves. Creating a database system from scratch or generating complex graphics on the screen could take a lifetime if we do not have the help of other programmers and developers. In computer programming, that help comes in the form of libraries and APIs. A library is a collection of variables, functions and objects that perform common tasks, such as calculating the pixels the system has to activate on the screen to show a tridimensional object or filtering the values returned by a database. They reduce the amount of code a developer has to write and provide a standard solution that works in every browser. Because of their complexity, libraries always offer an interface, a set of variables, functions, and objects, that we can use to communicate with the code and describe what we want the library to do for us. This visible part of the library is called *API* (Application Programming Interface), and it is what we have to learn to be able to include a library in our projects.

## Native Libraries

What turned HTML5 into the leading development platform that it is today was not the improvements on HTML, or the integration between this language with CSS and JavaScript, but the definition of a path to standardize the tools and features that vendors provide in their browsers by default. This includes a set of libraries that address common issues like the generation of 2D and 3D graphics, data storage, communication, and more. Because of HTML5, browsers now come packed with powerful libraries, and their APIs are integrated with common JavaScript objects and therefore readily available from our documents. Implementing these APIs in our code, we can perform complex tasks by calling a method or setting a property.

> **IMPORTANT:** Native APIs have become a fundamental part of the development of professional applications and video games, which is why

they will be the subject of study in the rest of this book.

# External Libraries

Before the appearance of HTML5, several JavaScript libraries were developed to overcome the limitations of the technologies available at the time. Some of those libraries were created with specific purposes, from processing and validating forms to the generation and manipulation of graphics. Through the years, they have become extremely popular, and some of them, like Google Maps, are almost impossible to imitate by independent developers.

These libraries are not part of HTML5 but are an important aspect of web development, and some of them have been implemented in today's most successful websites and applications. They enhance JavaScript and contribute to the development of new technologies for the Web. The following is a list of the most popular.

- **jQuery** (www.jquery.com) is a multipurpose library that simplifies JavaScript code and the interaction with the document. It makes HTML elements easier to select and animations easy to generate as well as handling events and helping implement Ajax in our applications.
- **React** (facebook.github.io/react) is a graphic library that helps with the creation of interactive user interfaces.
- **AngularJS** (www.angularjs.org) is a library that expands HTML elements to make them more dynamic and interactive.
- **Node.js** (www.nodejs.org) is a server-side library which purpose is the construction of network applications.
- **Modernizr** (www.modernizr.com) is a library that can detect the features available in the browser, including CSS properties, HTML elements, and JavaScript APIs.
- **Moment.js** (www.momentjs.com) is a library which sole purpose is to process dates.
- **Three.js** (www.threejs.org) is a 3D graphic library based on an API included in browsers called WebGL (Web Graphics Library). We will study this library in Chapter 12.

- **Google Maps** (developers.google.com/maps/) is a set of libraries designed to include maps in our websites and applications.

These libraries are usually small files that we can download from a website and include in our documents with the **<script>** element, as we do with our own JavaScript files. Once the library is included in our document, we can access its API from our code. For instance, the following document includes the Modernizr library to detect whether the CSS property **box-shadow** is available or not.

***Listing 6-170:*** *Detecting features with Modernizr*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Modernizr</title>
  <script src="modernizr-custom.js"></script>
  <script>
    function initiate(){
      var element = document.getElementById("subtitle");
      if (Modernizr.boxshadow) {
        element.innerHTML = 'Box Shadow is available';
      } else {
        element.innerHTML = 'Box Shadow is NOT available';
      }
    }
    window.addEventListener('load', initiate);
  </script>
</head>
<body>
  <section>
    <h1>Website</h1>
    <p id="subtitle"></p>
  </section>
</body>
</html>
```

Modernizr creates an object called **Modernizr** that offers properties for every HTML5 feature we want to detect. These properties return a Boolean value that will be **true** or **false** depending on whether or not the features are available. To include the library, we have to download the file from their website (www.modernizr.com) and then add it to our document with a **<script>** element, as we did in Listing 6-170.

The file generated by the website is called modernizr-custom.js and contains a detection system for all the features we have selected. In our example, we have selected the feature Box Shadow because that is the only thing we want to check. Once the library is loaded, we have to read the value of the property that represents the feature and respond accordingly. In this case, we insert a string in a **<p>** element. If the **box-shadow** property is available, the element will show the message "Box Shadow is available", if not, the message shown by the element will be "Box Shadow is NOT available".

**Do It Yourself:** Create a new HTML file with the document of Listing 6-170. Go to www.modernizr.com, select the feature Box Shadow (or the features you want to check), and click on Build to create your file. A file called modernizr-custom.js will be downloaded to your computer. Move the file to your document's folder and open the document in your browser. If your browser supports the CSS property **box-shadow**, you should see the message "Box Shadow is available" on the screen.

**IMPORTANT:** There are dozens of external JavaScript libraries available. This book does not cover the subject, but you can visit our website and follow the links for this chapter to find more information.

# Chapter 7 - Forms API

## 7.1 Processing Forms

The Forms API is a set of properties, methods, and events that we can use to process forms and create our own validation system. The API is embedded in forms and form elements, so we can listen to the events or call the methods from the elements themselves. The following are some of the methods available for **<form>** elements.

**submit()**—This method submits the form.

**reset()**—This method resets the form (it sets the elements to their default value).

**checkValidity()**—This method returns a Boolean value to indicate if the form is valid or not.

The API also offers the following event to report every time a character is inserted or a value is selected.

**input**—This event is fired on the form or the elements when the user inserts or erase a character in an input field, and also when a new value is selected.

**change**—This event is fired on the form or the elements when a new value is inserted or selected.

Using these methods and events, we can control the submission process from JavaScript. The following example submits the form when a button is pressed.

***Listing 7-1:*** *Submitting a form from JavaScript*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
```

```
  <title>Forms</title>
  <script>
    function initiate() {
      var button = document.getElementById("send");
      button.addEventListener("click", sendit);
    }
    function sendit() {
      var form = document.querySelector("form[name='information']");
      form.submit();
    }
    window.addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
    <form name="information" method="get" action="process.php">
      <p><label>Email: <input type="email" name="myemail" id="myemail"
required></label></p>
      <p><button type="button" id="send">Sign Up</button></p>
    </form>
  </section>
</body>
</html>
```

The code in Listing 7-1 adds a listener for the **click** event to the **<button>** element to execute the **sendit()** function every time the button is pressed. In this function, we get a reference to the **<form>** element and then submit the form with the **submit()** method.

In this example, we have decided to get a reference to the **<form>** element with the **querySelector()** method and a selector that matches its **name** attribute, but we could have also added an **id** attribute to the element to get the reference with the **getElementById()** method, as we did with the button. Another alternative is to get the reference from the **forms** property of the **Document** object. This property returns an array with references to all the **<form>** elements in the document, so we have to read the value at the index corresponding to the form we want to process (in our case, there is only one **<form>** element and therefore its reference is at index 0).

***Listing 7-2:*** *Getting a reference to the* <form> *element from the* forms *property*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Forms</title>
  <script>
    function initiate() {
      var button = document.getElementById("send");
      button.addEventListener("click", sendit);
    }
    function sendit() {
      var list = document.forms;
      var form = list[0];
      form.submit();
    }
    window.addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
    <form name="information" method="get" action="process.php">
      <p><label>Email: <input type="email" name="myemail" id="myemail"
required></label></p>
      <p><button type="button" id="send">Sign Up</button></p>
    </form>
  </section>
</body>
</html>
```

Submitting the form with the **submit()** method is the same as doing it with an **<input>** element of type **submit** (see Chapter 2), but the procedure skips the browser's validation process. If we want the form to be validated, we have to do it ourselves with the **checkValidity()** method. The method

checks the form and returns **true** or **false** to indicate whether it is valid or not.

***Listing 7-3:*** *Checking the validity of the form*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Forms</title>
  <script>
    function initiate() {
      var button = document.getElementById("send");
      button.addEventListener("click", sendit);
    }
    function sendit() {
      var form = document.querySelector("form[name='information']");
      var valid = form.checkValidity();
      if (valid) {
        form.submit();
      } else {
        alert("The form can't be submitted");
      }
    }
    window.addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
    <form name="information" method="get" action="process.php">
      <p><label>Email: <input type="email" name="myemail" id="myemail"
required></label></p>
      <p><button type="button" id="send">Sign Up</button></p>
    </form>
  </section>
</body>
</html>
```

The code in Listing 7-3 checks the values in the form to determine its validity. If the form is valid, it is submitted with the **submit()** method. Otherwise, a message is shown on the screen to warn the user.

**Do It Yourself:** Create a new HTML file with the document of Listing 7-3. Open the document in your browser and try to submit the form. You should see a popup window telling you that the form cannot be submitted. Insert a valid email address into the field. Now the form should be submitted.

# 7.2 Validation

As we have seen in Chapter 2, there are many ways to validate forms in HTML. We can use input types that require validation, such as **email**, turn a regular **text** type into a required field with the **required** attribute, or even use special types like **pattern** to customize the validation requirements. However, when it comes to more complex validation mechanisms, like comparing the values of two or more fields or checking the results of an operation, our only option is to customize the validation process using the Forms API.

# Custom Errors

Browsers show an error message when the user tries to submit a form that has an invalid field. These are predefined messages that describe known errors, but we can define custom messages to generate our own errors. To that end, **Element** objects representing form elements include the following method.

**setCustomValidity(**message**)**—This method sets a custom error and the message to be shown to the user if the form is submitted. If no message is provided, the error is cleared.

The following example presents a situation of complex validation. Two input fields are created to receive the user's first and last name. However, the form is only invalid when both fields are empty. Users may enter either their first or last name to validate the form. In a case like this, it is impossible to use the **required** attribute, because we do not know which input field the user will choose to fill. Only by using customized errors, we are able to create an effective validation mechanism for this scenario.

***Listing 7-4:*** *Setting custom error messages*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
```

```
  <title>Forms</title>
  <script>
    var name1, name2;
    function initiate() {
      name1 = document.getElementById("firstname");
      name2 = document.getElementById("lastname");
      name1.addEventListener("input", validation);
      name2.addEventListener("input", validation);
      validation();
    }
    function validation() {
      if (name1.value == "" && name2.value == "") {
        name1.setCustomValidity("Insert at least one name");
        name1.style.background = "#FFDDDD";
        name2.style.background = "#FFDDDD";
      } else {
        name1.setCustomValidity("");
        name1.style.background = "#FFFFFF";
        name2.style.background = "#FFFFFF";
      }
    }
    window.addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
    <form name="registration" method="get" action="process.php">
      <p><label>First Name: <input type="text" name="firstname"
id="firstname"></label></p>
      <p><label>Last Name: <input type="text" name="lastname"
id="lastname"></label></p>
      <p><input type="submit" value="Sign Up"></p>
    </form>
  </section>
</body>
</html>
```

The code in Listing 7-4 begins by creating references to the two **<input>** elements and adding listeners for the **input** event to both of them. This event is fired every time the user adds or erase a character, so we can track every value inserted into the field and validate or invalidate the form from the **validation()** function.

Because the **<input>** elements are empty when the document is loaded, we have to stop the user from submitting the form before at least one name is typed into the fields. For this reason, the **validation()** function is also called at the end of the **initiate()** function to check this condition.

The **validation()** function checks whether the form is valid or not and sets or removes a custom error with the **setCustomValidity()** method. If both names are empty strings, a custom error is set, and the background color for both elements is changed to red to indicate the error to the user. However, if that condition does not hold true later because at least one of the names was inserted, the error is cleared assigning an empty string to the method, and the backgrounds are set back to white.

It is important to mention that the only change produced during this process is the modification of the background color. The message declared for the error with the **setCustomValidity()** method will only be displayed when the user tries to submit the form.

**Do It Yourself:** Create a new HTML file with the document of Listing 7-4 and open the document in your browser. Try to submit the form. You should see an error with the message "Insert at least one name". Insert your name. The error should be removed.

**The Basics:** Several variables may be declared in the same line separated by commas. In Listing 7-4, we declared two global variables called **name1** and **name2**. The instruction could have been avoided because variables declared inside functions without the **var** operator have global scope, but declaring the variables at the beginning of the code makes it easier to maintain because we can identify with no much effort the variables our code requires to work.

# The invalid Event

Every time the user submits a form, an event is fired if an invalid entry is detected. The event is called **invalid** and it is fired on the element that produced the error. To customize the response, we can listen to this event from the **<form>** element, as in the following example.

***Listing 7-5:*** *Creating a custom validation system*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Forms</title>
  <script>
    var form;
    function initiate() {
      var button = document.getElementById("send");
      button.addEventListener("click", sendit);
      form = document.querySelector("form[name='information']");
      form.addEventListener("invalid", validation, true);
    }
    function validation(event) {
      var element = event.target;
      element.style.background = "#FFDDDD";
    }
    function sendit() {
      var valid = form.checkValidity();
      if (valid) {
        form.submit();
      }
    }
    window.addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
    <form name="information" method="get" action="process.php">
      <p><label>Nickname: <input pattern="[A-Za-z]{3,}"
name="nickname" id="nickname" maxlength="10" required></label></p>
```

```
    <p><label>Email: <input type="email" name="myemail" id="myemail"
required></label></p>
    <p><button type="button" id="send">Sign Up</button></p>
   </form>
  </section>
</body>
</html>
```

In Listing 7-5, we create a new form with two input fields to ask for a nickname and an email. The **email** input has its natural limitations due to its type and a **required** attribute, but the **nickname** input has three validation attributes: the **pattern** attribute that only admits a minimum of 3 characters from A to Z (uppercase and lowercase), the **maxlength** attribute that limits the input to a maximum of 10 characters, and the **required** attribute that invalidates the field if it is empty.

The code is very similar to previous examples. We listen to the **load** event to execute the **initiate()** function when the document finishes loading, add the listener for the **click** event to the **<button>** element, as always, but then a listener for the **invalid** event is added to the **<form>** element instead of the **<input>** elements. This is because we want to achieve a validation control for the entire form, not just for a single element. To do this, we have to include the value **true** as the third attribute of the **addEventListener()** method. This attribute tells the browser that it has to propagate the event to the rest of the elements in the hierarchy. As a result, even when the listener is added to the **<form>** element, it responds to events fired by the elements within the form. To determine which is the invalid element that calls the **validation()** function, we read the value of the **target** property. As we have seen in the previous chapter, this property returns a reference to the element that fired the event. Using that reference, the last instruction in this function changes the background color of the element to red.

# The ValidityState Object

The document of Listing 7-5 does not perform any real-time validation. The fields are validated only when the form is submitted. Considering the need for a more dynamic validation system, the Forms API includes the

**ValidityState** object. This object offers a series of properties to indicate the validity status of a form element.

**valid**—This property returns **true** if the value of the element is valid.

The **valid** property returns the validity state of an element considering every other possible validity status. If every condition is valid, then the **valid** property returns **true**. If we want to check a particular condition, we can read the rest of the properties offered by the **ValidityState** object.

**valueMissing**—This property returns **true** when the **required** attribute was declared, and the input field is empty.

**typeMismatch**—This property returns **true** when the syntax of the value inserted by the user does not comply with the type of the field. For example, if the text inserted for an input of type **email** is not an email address.

**patternMismatch**—This property returns **true** when the value inserted by the user does not match the provided pattern.

**tooLong**—This property returns **true** when the **maxlength** attribute was declared, and the value inserted by the user is longer than the value specified by the attribute.

**rangeUnderflow**—This property returns **true** when the **min** attribute was declared, and the value inserted by the user is less than the value specified by the attribute.

**rangeOverflow**—This property returns **true** when the **max** attribute was declared, and the value inserted by the user is greater than the value specified by the attribute.

**stepMismatch**—This property returns **true** when the **step** attribute was declared, and its value does not correspond to the value of the attributes **min**, **max** and **value**.

**customError**—This property returns **true** when we set a custom error. For example, when we use the **setCustomValidity()** method studied earlier.

The **ValidityState** object is assigned to a property called **validity**, available in every form element. The following example reads the value of this property to dynamically determine the validity of the elements in the form.

***Listing 7-6:*** *Validation in real time*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Forms</title>
  <script>
    var form;
    function initiate() {
      var button = document.getElementById("send");
      button.addEventListener("click", sendit);
      form = document.querySelector("form[name='information']");
      form.addEventListener("invalid", validation, true);
      form.addEventListener("input", checkval);
    }
    function validation(event) {
      var element = event.target;
      element.style.background = "#FFDDDD";
    }
    function sendit() {
      var valid = form.checkValidity();
      if (valid) {
        form.submit();
      }
    }
    function checkval(event) {
      var element = event.target;
      if (element.validity.valid) {
        element.style.background = "#FFFFFF";
      }else{
        element.style.background = "#FFDDDD";
      }
```

```
  }
  window.addEventListener("load", initiate);
 </script>
</head>
<body>
 <section>
  <form name="information" method="get" action="process.php">
   <p><label>Nickname: <input pattern="[A-Za-z]{3,}"
name="nickname" id="nickname" maxlength="10" required></label></p>
   <p><label>Email: <input type="email" name="myemail" id="myemail"
required></label></p>
   <p><button type="button" id="send">Sign Up</button></p>
  </form>
 </section>
</body>
</html>
```

In the code in Listing 7-6, a listener for the **input** event is added to the form. Every time the user modifies a field, inserting or erasing a character, the **checkval()** function is executed to respond to the event.

The **checkval()** function also takes advantage of the **target** property to get a reference to the element that fired the event and controls its validity by checking the value of the **valid** property inside the **validity** property of the **Element** object (**element.validity.valid**). Using this information, we change the color of the element's background that fired the **input** event in real time. The color will be red until the user types a valid entry.

**Do It Yourself:** Create a new HTML file with the document of Listing 7-6. Open the document in your browser and insert values in the input fields. You should see the background of the fields changing colors according to their validity (white valid, red invalid).

We can use the rest of the properties offered by the **ValidityState** object to know exactly what triggered the error, as in the following example.

***Listing 7-7:*** *Reading validity states to display a customized error message*

```
function sendit() {
```

```
  var element = document.getElementById("nickname");
  var valid = form.checkValidity();
  if (valid) {
    form.submit();
  } else if (element.validity.patternMismatch ||
element.validity.valueMissing) {
    alert('The nickname must have a minimum of 3 characters');
  }
}
```

In Listing 7-7, the **sendit()** function is modified to detect specific errors. The form is validated by the **checkValidity()** method, and if valid, it is submitted with **submit()**. Otherwise, the values of the **patternMismatch** and **valueMissing** properties of the **nickname** input field are checked, and an error message is shown when one or both of them return **true**.

**Do It Yourself:** Replace the **sendit()** function in the document of Listing 7-6 with the new function in Listing 7-7 and open the document in your browser. Type a single character in the **nickname** field and submit the form. You should see a popup window asking you to insert a minimum of 3 characters.

# 7.3 Pseudo-Classes

In addition to all the properties and methods provided by the Forms API, CSS includes a few pseudo-classes to modify the styles of an element depending on its validity status.

# Valid and Invalid

These pseudo-classes affect **<input>** elements with a valid or an invalid value.

***Listing 7-8:*** *Using the* :valid *and* :invalid *pseudo-classes*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Forms</title>
  <style>
    input:valid{
      background: #EEEEFF;
    }
    input:invalid{
      background: #FFEEEE;
    }
  </style>
</head>
<body>
  <section>
    <form name="myform" method="get" action="process.php">
      <input type="email" name="myemail" required>
      <input type="submit" value="Send">
    </form>
  </section>
</body>
</html>
```

The form in Listing 7-8 includes an **<input>** element for emails. When the content of the element is valid, the **:valid** pseudo-class assigns a blue background to the field. As soon as the content becomes invalid, the **:invalid** pseudo-class changes the background color to red.

# Optional and Required

These pseudo-classes affect form elements declared as required or optional.

***Listing 7-9:*** *Using the* :required *and* :optional *pseudo-classes*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Forms</title>
  <style>
    input:optional{
      border: 2px solid #009999;
    }
    input:required{
      border: 2px solid #000099;
    }
  </style>
</head>
<body>
  <section>
    <form name="myform" method="get" action="process.php">
      <p><input type="text" name="myname"></p>
      <p><input type="text" name="mylastname" required></p>
      <p><input type="submit" value="Send"></p>
    </form>
  </section>
</body>
</html>
```

The example of Listing 7-9 includes two input fields: **myname** and **mylastname**. The first one is optional, but **mylastname** is required. The pseudo-classes assign a different color to the border of these fields according to their condition (required is shown in blue and optional in green).

# In-range and Out-of-range

These pseudo-classes affect elements with a value inside or outside the specified range.

***Listing 7-10:*** *Using the* :in-range *and* :out-of-range *pseudo-classes*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Forms</title>
  <style>
    input:in-range{
      background: #EEEEFF;
    }
    input:out-of-range{
      background: #FFEEEE;
    }
  </style>
</head>
<body>
  <section>
    <form name="myform" method="get" action="process.php">
      <input type="number" name="mynumber" min="0" max="10">
      <input type="submit" value="Send">
    </form>
  </section>
</body>
</html>
```

An input field of type **number** is included in the example above to test these pseudo-classes. When the value entered in the element is less than **0** or greater than **10**, the background turns red, but as soon as we enter a value inside the specified range, the background turns blue.

# Chapter 8 - Media

## 8.1 Video

A video is a powerful form of communication. Nobody can deny the importance of video in websites and web applications today, let alone those developing the technologies for the Web. This is the reason why HTML5 includes an element to load and play videos.

**<video>**—This element embeds a video in the document.

The **<video>** element includes the following attributes to set the area occupied by the video and configure the player.

**src**—This attribute specifies the URL of the video to be played.

**width**—This attribute determines the width of the display area.

**height**—This attribute determines the height of the display area.

**controls**—This is a Boolean attribute. If present, the browser displays an interface to let the user control the video.

**autoplay**—This is a Boolean attribute. If present, the browser automatically plays the video as soon as it can.

**loop**—This is a Boolean attribute. If present, the browser plays the video in a loop.

**muted**—This is a Boolean attribute. If present, the audio is silenced.

**poster**—This attribute specifies the URL of the image that will be shown while waiting for the video to be played.

**preload**—This attribute determines if the browser should preload the video. It can take three values: **none**, **metadata** or **auto**. The first value indicates that the video shouldn't be cached, usually with the purpose of minimizing unnecessary traffic. The second value, **metadata**, recommends the browser to fetch some information about the resource, such as the dimensions, duration, first frame, etc. The third value, **auto**,

prompts the browser to download the file as soon as possible (this is the value by default).

The **<video>** element requires opening and closing tags and only a few parameters to accomplish its function. The syntax is simple, and only the **src** attribute is necessary to load a video.

***Listing 8-1:*** *Playing video with the* <video> *element*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Video Player</title>
</head>
<body>
  <section>
    <video src="trailer.mp4">
    </video>
  </section>
</body>
</html>
```

The **<video>** element loads the video specified by the **src** attribute and reserves an area in the document of the size of the video, but the video is not played. We have to tell the browser when we want to play the video or provide the tools for the user to decide. There are two attributes we can add to the element for this purpose: **controls** and **autoplay**. The **controls** attribute indicates to the browser that it should provide its own elements (buttons and sliders) to let the user control the video, and the **autoplay** attribute asks the browser to start playing the video as soon as it can.

***Listing 8-2:*** *Activating the browser's controls*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Video Player</title>
```

```
</head>
<body>
  <section>
    <video src="trailer.mp4" controls autoplay>
    </video>
  </section>
</body>
</html>
```

**Do It Yourself:** Create a new HTML file with the document of Listing 8-2. Download the trailer.mp4 video from our website. Open the document in your browser. The browser should begin playing the video right away and provide buttons to control it.

The browsers determine the size of the display area from the size of the video, but we can define a custom size with the **width** and **height** attributes. These attributes declare the dimensions of the element in pixels. When they are present, the size of the video is automatically adjusted to fit into those dimensions, but they are not intended to stretch the video. We may use them to limit the area occupied by the media and preserve consistency in our design.

***Listing 8-3:*** *Defining the display area*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Video Player</title>
</head>
<body>
  <section>
    <video src="trailer.mp4" width="720" height="400" controls>
    </video>
  </section>
</body>
</html>
```

**The Basics:** If you have to make the video responsive, you can ignore the **width** and **height** attributes and declare the size of the video from CSS and Media Queries. The **<video>** element may be adapted to the size of its container as we did with images in Chapter 5.

The **<video>** element includes additional attributes that may be useful sometimes. For instance, the **preload** attribute asks the browser to start downloading the video as soon as it can, so when the user decides to play it, the video is ready. We also have the **loop** attribute to play the video on a loop, and the **poster** attribute to specify an image that will be shown instead of the video while it is not being played.

***Listing 8-4:*** *Including a placeholder for the video*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Video Player</title>
</head>
<body>
  <section>
    <video src="trailer.mp4" width="720" height="400" preload
controls loop poster="poster.jpg">
    </video>
  </section>
</body>
</html>
```

The document of Listing 8-4 preloads the video, provides default controls, plays the video repeatedly, and shows an image instead of the video until the video starts playing. Figure 8-1 shows what we see before we press the play button.

***Figure 8-1:*** *Video poster*
*© Copyright 2008, Blender Foundation / [www.bigbuckbunny.org](http://www.bigbuckbunny.org)*

**Do It Yourself:** Update your document with the code in Listing 8-4. Download the poster.jpg file from our website. Open the document in your browser. You should see something similar to Figure 8-1.

# Video Formats

In theory, the **<video>** element alone should be more than enough to load and play a video, but things get a little more complicated in real life. This is because even when the **<video>** element and its attributes are standard, there is no standard video format for the Web. The problem is that some browsers support a group of codecs that others do not, and the codec used in the MP4 format (the only one supported by major browsers like Safari and Internet Explorer) has a commercial license.

The most common options are OGG, MP4, and WebM. These formats are containers for video and audio. OGG contains Theora video and Vorbis audio codecs, MP4 contains H.264 for video and AAC for audio, and WebM uses VP8 video codec and Vorbis audio codec. Presently, OGG and WebM are supported by Mozilla Firefox, Google Chrome, and Opera, while MP4 works on Safari, Internet Explorer and Google Chrome.

HTML considers this scenario and includes an additional element that works along with the **<video>** element to define possible sources for a video.

**<source>**—This element defines a source for a video. It must include the **src** attribute to define the file's URL.

When we need to specify multiple sources for the same video, we have to replace the **src** attribute in the **<video>** element with **<source>** elements between the tags, as in the following example.

***Listing 8-5:*** *Creating a cross-browser video player with default controls*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Video Player</title>
</head>
<body>
  <section>
    <video width="720" height="400" controls>
      <source src="trailer.mp4">
      <source src="trailer.ogg">
    </video>
  </section>
</body>
</html>
```

In Listing 8-5, the **<video>** element is expanded. Now, within the element's tags, there are two **<source>** elements. These elements provide different video sources for the browser to choose from. The browser reads the **<source>** elements and decides which file should be played according to the supported formats (in this case, MP4 or OGG).

**Do It Yourself:** Create a new HTML file with the document of Listing 8-5. Download the files trailer.mp4 and trailer.ogg from our website. Open the document in your browser. The video should play as usual, but now the browser selects which source to use.

**IMPORTANT:** Browsers require videos to be sent by the server with the corresponding MIME type. Every file has a MIME type associated to indicate the format of its content. For instance, the MIME type for an HTML file is **text/html**. Servers are already configured for the most

common video formats but usually are not for new formats such as OGG or WEBM. How to include this new MIME type depends on the characteristics of your server. A simple way to do it is to append a new line to the **.htaccess** file. Most servers provide this configuration file in the root folder of every website. The corresponding syntax is **Addtype MIME/type extension** (e.g., **AddType video/ogg ogg**).

# 8.2 Audio

Audio is not as popular as video. We can shoot a video with a personal camera that will generate millions of views but accomplishing the same result with an audio file would be almost impossible. However, audio is still present on the Web through radio shows and podcasts. For this reason, HTML5 also provides an element to play audio files.

**<audio>**—This element embeds audio in the document.

This element works in the same way and shares several attributes with the **<video>** element.

**src**—This attribute specifies the URL of the file to be played.

**controls**—This is a Boolean attribute. If present, it activates the interface provided by browsers.

**autoplay**—This is a Boolean attribute. If present, the browser automatically plays the audio as soon as it can.

**loop**—This is a Boolean attribute. If present, the browser plays the audio in a loop.

**preload**—This attribute determines if the browser should preload the audio file. It can take three values: **none**, **metadata** or **auto**. The first value indicates that the audio shouldn't be cached, usually with the purpose of minimizing unnecessary traffic. The second value, **metadata**, recommends the browser to fetch some information about the resource, such as the duration. The third value, **auto**, prompts the browser to download the file as soon as possible (this is the value by default).

The implementation of the **<audio>** element in our document is very similar to the **<video>** element. We have to specify the source and provide a way for the user to play the audio.

***Listing 8-6:*** *Playing audio with the* <audio> *element*

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="utf-8">
  <title>Audio Player</title>
</head>
<body>
  <section>
    <audio src="beach.mp3" controls>
    </audio>
  </section>
</body>
</html>
```

Again, we have to talk about codecs, and again, the HTML code in Listing 8-6 should be more than enough, but it is not. MP3 is under commercial license; therefore, it is not supported by browsers like Mozilla Firefox or Opera. Vorbis (the audio codec in the OGG container) is supported by these browsers but not by Safari and Internet Explorer. So, once again, we have to use the **<source>** element to provide at least two formats for the browser to choose from.

***Listing 8-7:*** *Creating a cross-browser audio player with default controls*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Audio Player</title>
</head>
<body>
  <section>
    <audio id="media" controls>
      <source src="beach.mp3">
      <source src="beach.ogg">
    </audio>
  </section>
</body>
</html>
```

The document of Listing 8-7 plays music in every browser with controls by default. Those that cannot play MP3 will play OGG and vice versa. What we have to remember is that MP3, as well as the MP4 format for video, are restricted by commercial licenses, so we can only use them in some circumstances according to what is determined those licenses.

# 8.3 Media API

Adding the **controls** attribute to the **<video>** or **<audio>** elements, we activate the interface provided by each browser. This interface may work for simple websites or small applications, but in a professional application, where every detail counts, it is necessary to have absolute control over the whole process and provide a consistent design across all devices and applications. Browsers include the Media API to provide this alternative. This is a group of properties, methods, and events designed to manipulate and integrate video and audio. Combining this API with HTML and CSS, we can create our own video or audio player with the features we want.

The API is embedded in the **Element** objects that represent the **<video>** and **<audio>** elements. The following are some of the properties included in these objects to provide information about the media.

**paused**—This property returns **true** if the media is currently paused or hasn't started playing.

**ended**—This property returns **true** if the media has finished playing.

**duration**—This property returns the length of the media in seconds.

**currentTime**—This property sets or returns a value that determines the position in which the media is being played or should start playing.

**volume**—This property sets or returns the volume of the media. The possible values range from 0.0 to 1.0.

**muted**—This property sets or returns the condition of the audio. The values are **true** (muted) or **false** (not muted).

**error**—This property returns the error value if an error has occurred.

**buffered**—This property offers information about how much of the file has been loaded into the buffer. Because users may force the browser to download the media from different positions in the timeline, the information returned by **buffered** is an array containing every part of the media that was downloaded, not just the one that starts from the beginning. The elements of the array are accessible by the attributes **end()** and **start()**. For example, the code **buffered.start(0)** returns the

time when the first portion of the media begins, and **buffered.end(0)** returns the time when it ends.

The elements also include the following methods to manipulate the media.

**play()**—This method plays the media.

**pause()**—This method pauses the media.

**load()**—This method loads the media file. It is useful for loading the media in advance from dynamic applications.

**canPlayType(**type**)**—This method returns a value that determines whether a file format is supported by the browser or not. The **type** attribute is a MIME type that represents the media format, such as video/mp4 or video/ogg. The method can return three values depending on how confident it is that it can play the media: an empty string (not supported), the string "maybe" and the string "probably".

Events are also available for this API to inform about the current situation of the media, like the progress downloading the file, whether the video has reached the end, or whether the video is paused or playing, among others. The following are the most frequently used.

**progress**—This event is fired periodically to provide updates on the progress downloading the media. The information is accessible through the **buffered** attribute.

**canplaythrough**—This event is fired when the entire media can be played without interruption. The status is established considering the current download rate and assuming that it will be the same for the rest of the process. There is another event for this purpose called **canplay**, but it does not consider the whole situation, and it is fired when there are only a couple of frames available.

**ended**—This event is fired when the media reaches the end.

**pause**—This event is fired when the media is paused.

**play**—This event is fired when the media starts playing.

**error**—This event is fired when an error occurs. It is delivered to the **<source>** element corresponding to the media source that produces the error.

# Video Player

Every video player needs a control panel with some basic features. The API does not provide a way to create these buttons or sliders; we have to define them using HTML and CSS. The following example creates an interface with two buttons to play and mute the video, a ** ** element to represent a progress bar, and a slider to control the volume.

***Listing 8-8:*** *Creating a video player with HTML*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Video Player</title>
  <link rel="stylesheet" href="player.css">
  <script src="player.js"></script>
</head>
<body>
  <section id="player">
    <video id="media" width="720" height="400">
      <source src="trailer.mp4">
      <source src="trailer.ogg">
    </video>
    <nav>
      <div id="buttons">
        <input type="button" id="play" value="Play">
        <input type="button" id="mute" value="Mute">
      </div>
      <div id="bar">
        <div id="progress"></div>
      </div>
      <div id="control">
```

```
      <input type="range" id="volume" min="0" max="1" step="0.1"
value="0.6">
     </div>
     <div class="clear"></div>
   </nav>
  </section>
</body>
</html>
```

This document also includes resources from two external files. One of those files is player.css with the styles required by the player.

***Listing 8-9:*** *Designing the video player*

```
#player {
  width: 720px;
  margin: 20px auto;
  padding: 10px 5px 5px 5px;
  background: #999999;
  border: 1px solid #666666;
  border-radius: 10px;
}
#play, #mute {
  padding: 5px 10px;
  width: 65px;
  border: 1px solid #000000;
  background: #DDDDDD;
  font-weight: bold;
  border-radius: 10px;
}
nav {
  margin: 5px 0px;
}
#buttons {
  float: left;
  width: 135px;
  height: 20px;
  padding-left: 5px;
```

```
}
#bar {
  float: left;
  width: 400px;
  height: 16px;
  padding: 2px;
  margin: 2px 5px;
  border: 1px solid #CCCCCC;
  background: #EEEEEE;
}
#progress {
  width: 0px;
  height: 16px;
  background: rgba(0,0,150,.2);
}
.clear {
  clear: both;
}
```

The CSS code in Listing 8-9 uses techniques from the Traditional Box Model to draw a box at the center of the screen that contains every component of the video player. There is nothing unusual in this code except for the size assigned to the **progress** element. As we did in the example of Listing 6-163 (Chapter 6), we define the initial width of this element as 0 pixels to use it as a progress bar. The result is illustrated in Figure 8-2.


***Figure 8-2:*** *Custom video player*

**Do It Yourself:** Create a new HTML file with the document of Listing 8-8. Create two more files for the CSS styles and JavaScript codes called player.css and player.js, respectively. Copy the code in Listing 8-9 into the CSS file and then copy every JavaScript code listed below into the JavaScript file.

As always, we should begin our JavaScript code by declaring the variables required by our application and the function that is going to be executed as soon as the document is loaded.

***Listing 8-10:*** *Initializing the application*

```
var maxim, mmedia, play, bar, progress, mute, volume, loop;
function initiate() {
  maxim = 400;
  mmedia = document.getElementById("media");
  play = document.getElementById("play");
  bar = document.getElementById("bar");
  progress = document.getElementById("progress");
  mute = document.getElementById("mute");
  volume = document.getElementById("volume");

  play.addEventListener("click", push);
  mute.addEventListener("click", sound);
  bar.addEventListener("click", move);
  volume.addEventListener("change", level);
}
```

In the **initiate()** function in Listing 8-10, we create a reference to each element and also set the variable **maxim** to always know the maximum size of the progress bar (400 pixels). In addition, we set up listeners for multiple events to respond to the user. There are several actions we have to pay attention to: when the user clicks on the Play and Mute buttons, when changes the volume from the **range** input, and when clicks on the progress bar to move the video forward or backward. For this purpose, listeners for the **click** event are added to the elements **play**, **mute**, and **bar**, and one for

the **change** event is added to the **range** input to control the volume. Every time the user clicks on one of these elements or moves the slider, the correspondent functions are executed: **push()** for the Play button, **sound()** for the Mute button, **move()** for the progress bar, and **level()** for the slider.

The first function we have to implement is **push()**. This function is responsible for playing or pausing the video when the Play or Pause buttons are pressed.

***Listing 8-11:*** *Playing and pausing the video*

```
function push() {
  if (!mmedia.paused && !mmedia.ended) {
    mmedia.pause();
    play.value = "Play";
    clearInterval(loop);
  } else {
    mmedia.play();
    play.value = "Pause";
    loop = setInterval(status, 1000);
  }
}
```

The **push()** function is executed when the user clicks the Play button. This button has two purposes: it shows the text "Play" to play the video or "Pause" to pause it, according to the current state. So when the video is paused, or it has not started, pressing the button will play the video, but the opposite occurs if the video is already being played. To determine this condition, the code detects the state of the media checking the properties **paused** and **ended**. This is done by the **if** instruction in the first line of the function. If the values of the **paused** and **ended** properties are **false**, it means that the video is playing, and therefore the **pause()** method is executed to pause the video, and the text of the button is changed to "Play". Notice that we apply the **!** operator to each property (logical NOT) to achieve this purpose. If the properties return **false**, the operator changes this value to **true**. The **if** instruction should be read "if the media is *not* paused and the media has *not* ended, then do this".

If the video is paused or has finished playing, the condition is **false**, and the **play()** method is executed to start or resume the video. In this case, we

are also performing an important action which is to initiate the execution of the **status()** function each second with the **setInterval()** method of the **Window** object (see Chapter 6) to update the progress bar. The following is the implementation of this function.

***Listing 8-12:*** *Updating the progress bar*

```
function status() {
  if (!mmedia.ended) {
    var size = parseInt(mmedia.currentTime * maxim /
mmedia.duration);
    progress.style.width = size + "px";
  } else {
    progress.style.width = "0px";
    play.value = "Play";
    clearInterval(loop);
  }
}
```

The **status()** function is executed each second while the video is being played. In this function, we also have an **if** instruction to test the status of the video. If the **ended** property returns **false**, we calculate how long the progress bar should be in pixels and set a new size for the ** ** element that is representing it. But if the value of the **ended** property is **true** (which means the video has finished), we set the size of the progress bar back to **0** pixels, change the text of the button to "Play", and cancel the loop with the **clearInterval()** method. After this, the **status()** function will no longer be executed.

This code updates the progress bar, but let's see how we calculate the current value. We need the value of the **currentTime** property to know what part of the video is being played, the value of the **duration** property to know how long is the video, and the value of the **maxim** variable to know the maximum size allowed for the progress bar. The formula is a simple rule of three. We have to multiply the current time by the maximum size and divide the result by the total duration (**current-time** × **maximum** / **total-duration**). The result is the new size in pixels of the ** ** element that represents the progress bar.

The function to handle the **click** event for the **play** element (the Play button) has already been created; now it is time to do the same for the progress bar. We called this method **move()**.

***Listing 8-13:*** *Playing from the position selected by the user*

```
function move(event) {
  if (!mmedia.paused && !mmedia.ended) {
    var mouseX = event.offsetX - 2;
    if (mouseX < 0) {
      mouseX = 0;
    } else if (mouseX > maxim) {
      mouseX = maxim;
    }
    var newtime = mouseX * mmedia.duration / maxim;
    mmedia.currentTime = newtime;
    progress.style.width = mouseX + "px";
  }
}
```

In the **initiate()** function, a listener for the **click** event was added to the **bar** element to check every time the user wants to start playing the video from a new position. When the event is fired, the **move()** function is executed to handle it. This function begins with an **if** instruction, like the previous functions, but this time the goal is to perform the action only when the video is being played. If the properties **paused** and **ended** are **false** it means that the video is playing and the code has to be executed.

To calculate the time when the video should start playing, we get the position of the mouse from the **offsetX** property (see Listing 6-163, Chapter 6), calculate the size of the progress bar considering the paddings of the **bar** element, and then convert this size into seconds to start playing the video from the new location. The value in seconds that the position of the mouse represents in the timeline is calculated by the formula **mouseX** × **video.duration** / **maxim**. Once we get this value, we have to assign it to the **currentTime** property to start playing the video from that time. In the end, the position of the mouse is assigned to the **width** property of the **progress** element to reflect the new state of the video on the screen.

There are two small functions left we need to introduce to control the volume of the media. The first one is the **sound()** function, assigned as the listener for the **click** event of the Mute button.

***Listing 8-14:*** *Switching to sound or silence with the* muted *property*

```
function sound() {
  if (mute.value == "Mute") {
    mmedia.muted = true;
    mute.value = "Sound";
  } else {
    mmedia.muted = false;
    mute.value = "Mute";
  }
}
```

The function in Listing 8-14 activates or deactivates the audio of the media depending on the value of the **value** attribute of the Mute button. The button displays different texts according to the situation. If the current value is "Mute", the sound is muted, and the value of the button is changed to "Sound". Otherwise, the audio is activated, and the value of the button is changed to "Mute".

When the sound is active, the volume can be controlled from the slider created by the **range** input, located at the end of the progress bar. The element fires the **change** event when its value changes. The event is handled by the **level()** function.

***Listing 8-15:*** *Controlling the volume*

```
function level() {
  mmedia.volume = volume.value;
}
```

The function in Listing 8-15 assigns the value of the **value** attribute of the **<input>** element that represents the slider to the **volume** property of the media. The only thing we have to keep in mind is that this property takes values from **0.0** to **1.0**. Numbers out of that range will return an error.

With this small function, the code for the video player is almost ready; all it is left to do is to listen to the **load** event to start the application when

the document is loaded.

***Listing 8-16:*** *Listening to the* load *event to start the application*

```
window.addEventListener("load", initiate);
```

> **Do It Yourself:** Copy all the JavaScript codes since Listing 8-10 into the player.js file. Open the document of Listing 8-8 in your browser and click the Play button. Try the application in different browsers.

# 8.4 Closed Captioning

Closed captioning is a feature added to video players to show text on the screen while the video is being played. It has been used in television and different forms of video distribution for decades, but it has always been difficult to replicate on the Web. HTML introduces the following element to simplify its implementation.

**<track>**—This element adds closed captioning to a video.

The element has to be included as a child of the **<video>** or **<audio>** elements. To specify the source, type, and how the closed captioning is going to be shown on the screen, the **<track>** element provides the following attributes.

**src**—This attribute declares the URL of the file containing the text for the closed captioning. The format of this file may be any of those supported by browsers, but the specification declares the WebVTT format as the official for this element.

**srclang**—This attribute declares the human language of the text. It works with the same values as the **lang** attribute of the **<html>** element studied in Chapter 2.

**default**—This attribute declares the track we want to include by default. If only one **<track>** element is provided, this attribute may be used to activate closed captioning.

**label**—This attribute provides a title for the track. If multiple **<track>** elements are included, this attribute helps users find the right one.

**kind**—This attribute declares the type of content assigned to the track. The possible values are **subtitles** (for subtitles), **captions** (usually for captions representing sound), **descriptions** (for audio synthesis), **chapters** (for navigation between chapters) and **metadata** (for additional information, not shown on the screen). The default value is **subtitles**.

The **<track>** element works with both the **<video>** and the **<audio>** elements to add subtitles or show additional information for our videos and

audio tracks.

***Listing 8-17:*** *Adding subtitles with the* <track> *element*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Video Subtitles</title>
</head>
<body>
  <section>
    <video width="720" height="400" controls>
      <source src="trailer.mp4">
      <source src="trailer.ogg">
      <track src="subtitles.vtt" srclang="en" default>
    </video>
  </section>
</body>
</html>
```

> **IMPORTANT:** The **<track>** element does not allow the construction of cross-origin applications, and therefore you have to upload all the files to your own server and run everything under the same domain (including the HTML document, the videos, and the subtitles). Another alternative to test this example is to use a local server, like the one installed by the MAMP application (see Chapter 1 for more information). This situation can be avoided using the CORS technology, and the **crossorigin** attribute explained in Chapter 11.

In Listing 8-17, we declared only one **<track>** element. The language for the source was set as English, and the **default** attribute was included to set up this track as default and activate closed captioning. The source of this element was declared as a file in the WebVTT format (subtitles.vtt). WebVTT stands for Web Video Text Tracks, the standard format for closed captioning. The files in this format are text files with a specific structure, as illustrated by the following example.

***Listing 8-18:*** *Defining a WebVTT file*

```
WEBVTT

00:02.000 --> 00:07.000
Welcome
to the &lt;track&gt; element!

00:10.000 --> 00:15.000
This is a simple example.

00:17.000 --> 00:22.000
Several tracks can be used simultaneously

00:22.000 --> 00:25.000
to provide text in different languages.

00:27.000 --> 00:30.000
Goodbye!
```

Listing 8-18 shows the basic structure of a WebVTT file. The first line with the text "WEBVTT" is mandatory as it is the empty line in between declarations. The declarations are called *cues*, and they require the syntax minutes:seconds.milliseconds to indicate the start and the end times. This is a rigid format; we always have to respect the structure shown in the example of Listing 8-18 and declare each parameter with the same number of digits (two for minutes, two for seconds and three for milliseconds).

**Do It Yourself:** Create a new HTML file with the document of Listing 8-17. Create a text file with the declarations in Listing 8-18 and name it subtitles.vtt. Upload these files and the video files to your server or copy the files to your local server (see MAMP in Chapter 1), and open the HTML document in your browser. You should see the video playing with subtitles and a button on the side to activate or deactivate the subtitles.

The cues (the lines of text in a closed captioning file) may include the special tags **<b>**, **<i>**, **<u>**, **<v>** and **<c>**. The first three are for emphasis,

as in HTML, while the **<v>** tag declares whose voice the text is representing and the **<c>** tag allows us to provide custom styles using CSS.

***Listing 8-19:*** *Including tags in a WebVTT file*

```
WEBVTT

00:02.000 --> 00:07.000
<i>Welcome</i>
to the &lt;track&gt; element!

00:10.000 --> 00:15.000
<v Robert>This is a simple <c.captions>example</c>.

00:17.000 --> 00:22.000
<v Martin>Several tracks can be used simultaneously

00:22.000 --> 00:25.000
<v Martin>to provide text in different languages.

00:27.000 --> 00:30.000
<b>Good bye!</b>
```

WebVTT uses a pseudo-element for referencing the cues. This pseudo-element, called **::cue**, can take a selector for a class between parentheses. In Listing 8-19, the class is called *captions* (<c.captions>), so the CSS selector should be written as **::cue(.captions)**, as in the following example.

***Listing 8-20:*** *Declaring styles for WebVTT*

```
::cue(.captions){
  color: #990000;
}
```

**IMPORTANT:** Only a reduced group of CSS properties, such as **color**, **background**, and **font**, are available for this pseudo-element; the rest are ignored.

WebVTT also provides the possibility of aligning and positioning each cue using the following parameters and values.

**align**—This parameter aligns the cue in relation to the center of the space covered by the media. The possible values are **start**, **middle**, and **end**.

**vertical**—This parameter changes the orientation to vertical and orders the cue according to two values: **rl** (right to left) or **lr** (left to right).

**position**—This parameter sets the position of the cue in columns. The value can be expressed as a percentage or a number from **0** to **9**. The position declared is established according to the orientation.

**line**—This parameter sets the position of the cue in rows. The value can be expressed as a percentage or a number from **0** to **9**. In a horizontal orientation, the position declared is vertical, and vice versa. Positive numbers set the position from one side and negative numbers from the other, depending on the orientation.

**size**—This parameter sets the size of the cue. The value may be declared as a percentage, and it is determined from the width of the media.

These parameters and their corresponding values are declared at the end of the cue separated by a colon. Multiple declarations can be made for the same cue, as shown in the following example.

***Listing 8-21:*** *Configuring cues*

```
WEBVTT

00:02.000 --> 00:07.000 align:start position:5%
<i>Welcome</i>
to the &lt;track&gt; element!
```

**Do It Yourself:** Using the WebVTT file created in Listing 8-18, try to combine different parameters in the same cues to see the effects available for this format.

# 8.5 TextTrack API

The TextTrack API was defined to offer access from JavaScript to the content of the tracks used in closed captioning. The API includes an object called **TextTrack** to return this information. There are two ways to get this object: from the media element or from the **<track>** element.

**textTracks**—This property contains an array with the **TextTrack** objects corresponding to each track of the media. The **TextTrack** objects are stored in the array in sequential order.

**track**—This property returns the **TextTrack** object of the specified track.

If the media (video or audio) has several **<track>** elements, it may be easier to find the track we want accessing the **textTracks** array.

***Listing 8-22:*** *Getting the* TextTrack *object*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Working with Tracks</title>
  <script>
    function initiate() {
      var video = document.getElementById("media");
      var track1 = video.textTracks[0];
      var mytrack = document.getElementById("mytrack");
      var track2 = mytrack.track;
      console.log(track1);
      console.log(track2);
    }
    window.addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
```

```
    <video id="media" width="720" height="400" controls>
      <source src="trailer.mp4">
      <source src="trailer.ogg">
      <track id="mytrack" label="English Subtitles" src="subtitles.vtt"
srclang="en" default>
    </video>
  </section>
</body>
</html>
```

The document of Listing 8-22 illustrates how to access the **TextTrack** object using both properties. The **initiate()** function creates a reference to the **<video>** element to get the **TextTrack** object accessing the **textTracks** array with the index **0** and then obtains the same object from the **<track>** element using the **track** property. Finally, the content of both variables is printed on the console.

**Do It Yourself:** Create a new HTML file with the document of Listing 8-22. Like previous examples, this application does not work on a local computer; you have to upload all the files to your server or local server for testing.

# Reading Tracks

Once we get the **TextTrack** object of the track we want to work with, we can access its properties.

**kind**—This property returns the type of the track, as specified by the **kind** attribute of the **<track>** element (**subtitles**, **captions**, **descriptions**, **chapters** and **metadata**).

**label**—This property returns the label of the track, as specified by the **label** attribute of the **<track>** element.

**language**—This property returns the language of the track, as specified by the **srclang** attribute of the **<track>** element.

**mode**—This property returns or sets the mode of the track. The three possible values are **disabled**, **hidden**, and **showing**. It may be used to switch tracks.

**cues**—This property is an array containing all the cues from the track file.

**activeCues**—This property returns the cues currently shown on the screen (the previous, current, and next cue).

From the properties of the **TextTrack** object, we have access to all the information stored in the track.

***Listing 8-23:*** *Displaying the track's information on the screen*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Working with Tracks</title>
  <style>
    #player, #info{
      float: left;
    }
  </style>
  <script>
    function initiate() {
      var info = document.getElementById("info");
      var mytrack = document.getElementById("mytrack");
      var obj = mytrack.track;

      var list = "";
      list += "<br>Kind: " + obj.kind;
      list += "<br>Label: " + obj.label;
      list += "<br>Language: " + obj.language;
      info.innerHTML = list;
    }
    window.addEventListener("load", initiate);
  </script>
```

```
</head>
<body>
  <section id="player">
    <video id="media" width="720" height="400" controls>
      <source src="trailer.mp4">
      <source src="trailer.ogg">
      <track id="mytrack" label="English Subtitles" src="subtitles.vtt"
srclang="en" default>
    </video>
  </section>
  <aside id="info"></aside>
</body>
</html>
```

The document of Listing 8-23 includes styles for the two columns created by the **<section>** and **<aside>** elements and JavaScript code to get and show the data from the **<track>** element. This time, the **TextTrack** object is retrieved from the **track** property and stored in the **obj** variable. From this variable, we get the values of the object's properties and generate the text to be shown on the screen.

# Reading Cues

In addition to the properties applied in the last example, there is an important property called **cues**. This property contains an array with **TextTrackCue** objects representing each cue of the track.

***Listing 8-24:*** *Displaying cues on the screen*

```
<script>
  function initiate() {
    var info = document.getElementById("info");
    var mytrack = document.getElementById("mytrack");
    var obj = mytrack.track;
    var mycues = obj.cues;

    var list = "";
```

```
    for (var f = 0; f < mycues.length; f++) {
      list += mycues[f].text + "<br>";
    }
    info.innerHTML = list;
  }
  window.addEventListener("load", initiate);
</script>
```

The new **initiate()** function in Listing 8-24 accesses every cue using a **for** loop. Inside the loop, the values of the array are added to the **list** variable along with a **<br>** element to show the cues one per line, and then the whole text is inserted into the **<aside>** element and displayed on the screen.

The **TextTrackCue** objects provide properties to access the information of the cue. In our example, we showed the content of the **text** property. The following is a list of all the properties available.

**text**—This property returns the text of the cue.

**startTime**—This property returns the start time of the cue in seconds.

**endTime**—This property returns the end time of the cue in seconds.

**vertical**—This property returns the value of the **vertical** parameter. If the parameter was not defined, the value returned is an empty string.

**line**—This property returns the value of the **line** parameter. If the parameter was not defined, the default value is returned.

**position**—This property returns the value of the **position** parameter. If the parameter was not defined, the default value is returned.

**size**—This property returns the value of the **size** parameter. If the parameter was not defined, the default value is returned.

**align**—This property returns the value of the **align** parameter. If the parameter was not defined, the default value is returned.

The following code updates the previous example to include the starting time of each cue.

***Listing 8-25:*** *Displaying information about cues*

```
<script>
  function initiate() {
    var info = document.getElementById("info");
    var mytrack = document.getElementById("mytrack");
    var obj = mytrack.track;
    var mycues = obj.cues;

    var list = "";
    for (var f = 0; f < mycues.length; f++) {
      var cue = mycues[f];
      list += cue.startTime + " - ";
      list += cue.text + "<br>";
    }
    info.innerHTML = list;
  }
  window.addEventListener("load", initiate);
</script>
```

**Do It Yourself:** Create a new HTML file with the document of Listing 8-23. Upload the files to your server or copy the files to your local server and open the document in your browser. To work with cues, replace the **initiate()** function in the document with the one you want to try. The cues and their values should be printed on the right side of the video.

# Adding Tracks

The **TextTrack** object that represents a track has not only properties but also three important methods to create new tracks and cues from JavaScript.

**addTextTrack(**kind, label, language**)**—This method creates a new track for the media and returns the corresponding **TextTrack** object. The attributes are the values of the attributes for the new track (only **kind** is mandatory).

**addCue(**object**)**—This method adds a new cue to the specified track. The **object** attribute is a **TextTrackCue** object returned by the

**VTTCue()** constructor.

**removeCue(**object**)**—This method removes a cue from the specified track. The **object** attribute is a **TextTrackCue** object returned by the **TextTrack** object.

To add cues to the track, we have to provide a **TextTrackCue** object. The API includes a constructor to create this object.

**VTTCue(**startTime, endTime, text**)**—This constructor returns a **TextTrackCue** object to use with the **addCue()** method. The attributes represent the data for the cue: start time, end time, and the text for the caption.

The following example demonstrates how to add a track to a media.

***Listing 8-26:*** *Adding tracks and cues from JavaScript*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Working with Tracks</title>
  <script>
    function initiate(){
      var cue;
      var cues = [
        { start: 2.000, end: 7.000, text: "Welcome"},
        { start: 10.000, end: 15.000, text: "This is an example"},
        { start: 15.001, end: 20.000, text: "of how to add"},
        { start: 20.001, end: 25.000, text: "a new track."},
        { start: 27.000, end: 30.000, text: "Good bye!"},
      ];
      var video = document.getElementById("media");
      var newtrack = video.addTextTrack("subtitles");
      newtrack.mode = "showing";
      for (var f = 0; f < cues.length; f++) {
        cue = new VTTCue(cues[f].start, cues[f].end, cues[f].text);
```

```
      newtrack.addCue(cue);
    }
    video.play();
  }
  window.addEventListener("load", initiate);
 </script>
</head>
<body>
 <section>
  <video id="media" width="720" height="400" controls>
   <source src="trailer.mp4">
   <source src="trailer.ogg">
  </video>
 </section>
</body>
</html>
```

in Listing 8-26, we start the **initiate()** function by defining the **cues** array with five cues for our new track. The cues are declared as elements of the array. Each cue is an object with the properties **start**, **end** and **text**. The values of the start and end times do not use the syntax of a WebVTT file; they have to be declared in seconds as decimal numbers.

The cues may be added to an existing track or a new one. In our example, we create a new track of type **subtitles** using the **addTextTrack()** method. We also have to declare the **mode** of this track as **showing**, to ask the browser to show the track on the screen. When the track is ready, all the cues from the **cues** array are converted into **TextTrackCue** objects and added to the track using the **addCue()** method.

**Do It Yourself:** Update your document with the code in Listing 8-26 and open the document in your browser. You should see the new track on top of the video. Remember to upload the files to your server or copy the files to your local server.

# Chapter 9 - Stream API

## 9.1 Capturing Media

The Stream API allows us to access media streams produced by the device. The most common stream sources are the webcam and the microphone, but the API was defined to access any other source that produces a video or audio stream.

The API defines the **MediaStream** object to reference media streams and the **LocalMediaStream** object for media generated by local devices. These objects have an input represented by the device and an output represented by the **<video>** element, the **<audio>** element, or other APIs. To get the **LocalMediaStream** object representing the stream, the API includes an object called **MediaDevices** that, among others, it provides the following method.

**getUserMedia(**constraints**)**—This method asks the user permission to access the video or audio stream and in response generates a **LocalMediaStream** object to represent the stream. The **constraints** attribute is an object with two properties **video** and **audio** to indicate the type of media to be captured.

The **getUserMedia()** method performs an asynchronous operation, which means that the method will try to access the stream while the rest of the code is being processed. For this purpose, it returns an object of type **Promise** to report the result. This is an object specifically designed to manage asynchronous operations. It provides two methods to process the response.

**then(**function**)**—This method is executed by an asynchronous operation in case of success. If the operation is successful, this method is called, and the function specified by the attribute is executed.

**catch(**function**)**—This method is executed by an asynchronous operation in case of failure. If the operation fails, this method is called, and the function specified by the attribute is executed.

When a stream is accessed, the **then()** method sends the reference of the **LocalMediaStream** object that is representing the stream to the function. To show it to the user, we have to assign this object to a **<video>** or **<audio>** element. For this purpose, the **Element** objects representing these elements provide the following property.

**srcObject**—This property sets or returns a **MediaStream** object representing a stream.

The most common device used to stream video is the webcam. The following example illustrates how to access the webcam and show the stream on the screen.

***Listing 9-1:*** *Accessing the webcam*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Stream API</title>
  <script>
    function initiate() {
      var promise = navigator.mediaDevices.getUserMedia({video: true});
      promise.then(success);
      promise.catch(showerror);

      function success(stream) {
        var video = document.getElementById("media");
        video.srcObject = stream;
        video.play();
      }
      function showerror(error) {
        console.log("Error: " + error.name);
      }
    }
    window.addEventListener("load", initiate);
  </script>
```

```
</head>
<body>
  <section>
    <video id="media"></video>
  </section>
</body>
</html>
```

The **initiate()** function in Listing 9-1 gets the video from the webcam using the **getUserMedia()** method. This prompts a request to the user. If the user allows our application to access the webcam, the **then()** method of the **Promise** object is called and the **success()** function is executed. This function receives the **LocalMediaStream** object produced by the **getUserMedia()** method and assigns it to the **<video>** element in the document through the **srcObject** property.

In case of failure, the **catch()** method of the **Promise** object is called, and the **showerror()** function is executed. The function receives an object with information about the error. The most common error is PermissionDeniedError, produced when the user denies access to the media or the media is not available for other reasons.

Notice that we did not have to declare the **src** attribute in the **<video>** tag because the source will be the media stream captured by the JavaScript code, but we could have set the **width** and **height** attributes if we wanted to specify the size of the video.

**IMPORTANT:** The **getUserMedia()** method can only be executed from a server, and the server must be secure, which means that the application will not work unless you upload the files to a server and the server uses the https protocol. If you do not have a secure server, you can test the examples in this chapter using a local server set up by an application like MAMP. See Chapter 1 for more information.

**The Basics:** The **MediaDevices** object that provides the **getUserMedia()** method belongs to the **Navigator** object (see Chapter 6). When the browser creates the **Window** object, it stores a reference to the **Navigator** object in a property called **navigator** and the **MediaDevices** object in a property called **mediaDevices**. This is why, in

the example of Listing 9-1, we use these properties to call the **getUserMedia()** method (**navigator.mediaDevices.getUserMedia()**).

# The MediaStreamTrack object

**MediaStream** objects (and therefore **LocalMediaStream** objects) contain **MediaStreamTrack** objects that represent each media track (usually one for the video and another for the audio). **MediaStream** objects include the following methods to retrieve these **MediaStreamTrack** objects.

**getVideoTracks()**—This method returns an array with **MediaStreamTrack** objects representing the video tracks included in the stream.

**getAudioTracks()**—This method returns an array with **MediaStreamTrack** objects representing the audio tracks included in the stream.

The **MediaStreamTrack** objects returned by these methods include properties and methods to get information and control the video or audio track. The following are the most frequently used.

**enabled**—This property returns **true** or **false** according to the track's status. If the track is still associated with the source, the value is **true**.

**kind**—This property returns the type of source the track represents. It has two possible values: **audio** or **video**.

**label**—This property returns the name of the source for the track.

**stop()**—This method stops the track.

If we want to collect information about the stream, we have to create the stream with the **getUserMedia()** method as we did in the previous example, get references to the tracks in the stream with the **getVideoTracks()** method, and then read the values of the tracks returned.

***Listing 9-2:*** *Displaying the information of the stream*

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Stream API</title>
  <style>
    section {
      float: left;
    }
  </style>
  <script>
    function initiate() {
      var promise = navigator.mediaDevices.getUserMedia({video: true});
      promise.then(success);
      promise.catch(showerror);

      function success(stream) {
        var video = document.getElementById("video");
        video.srcObject = stream;
        video.play();

        var databox = document.getElementById("databox");
        var videoTracks = stream.getVideoTracks();
        var track = videoTracks[0];
        var data = "";
        data += "<br> Enabled: " + track.enabled;
        data += "<br> Type: " + track.kind;
        data += "<br> Device: " + track.label;
        databox.innerHTML = data;
      }
      function showerror(error) {
        console.log("Error: " + error.name);
      }
    }
    window.addEventListener("load", initiate);
  </script>
</head>
<body>
```

```
  <section>
    <video id="video"></video>
  </section>
  <section id="databox"></section>
</body>
</html>
```

The **getVideoTracks()** method of the **MediaStream** object returns an array. Because the webcam contains only one video track, to get a reference to this track we have to read the element at index **0** (the first track on the list).

In the document of Listing 9-2, we use the same code from the previous example to access the stream, but this time the track for the webcam is stored in the **track** variable, and then every one of its properties is shown on the screen.

> **IMPORTANT:** The tracks mentioned here are video and audio tracks. These types of tracks are not related to the closed captioning tracks studied in Chapter 8.

Once we get a reference to the track, we can stop it with the **stop()** method. In the following example, we add a button to the interface to stop the stream when it is pressed by the user. Notice that the listener for the **click** event of the button is only added if the code is able to access the camera. Otherwise, the button does nothing.

***Listing 9-3:*** *Stopping the stream*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Stream API</title>
  <script>
    function initiate() {
      var promise = navigator.mediaDevices.getUserMedia({video: true});
      promise.then(success);
      promise.catch(showerror);
```

```
    }
    function success(stream) {
      var button = document.getElementById("button");
      button.addEventListener("click", function() { stopstream(stream)
});
      var video = document.getElementById("video");
      video.srcObject = stream;
      video.play();
    }
    function showerror(event) {
      console.log("Error: " + error.name);
    }
    function stopstream(stream) {
      var videoTracks = stream.getVideoTracks();
      var track = videoTracks[0];
      track.stop();
      alert("Stream Canceled");
    }
    window.addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
    <video id="video"></video>
  </section>
  <nav>
    <button type="button" id="button">Turn Off</button>
  </nav>
</body>
</html>
```

In addition to the properties and methods mentioned above, **MediaStreamTrack** objects also offer events to report the state of the track.

**muted**—This event is fired when the track is unable to provide data.

**unmuted**—This event is fired as soon as the track starts providing data again.

**ended**—This event is fired when the track can no longer provide data. There may be several reasons why this occurs, from the user denying access to the stream to the use of the **stop()** method.

**IMPORTANT:** These events are meant to be used to control remote streaming, a process studied in Chapter 24.

# Chapter 10 - Fullscreen API

## 10.1 Modern Applications

The Web is a multipurpose and multimedia platform where everything is possible. With so many interesting new applications, the word *navigation* has almost lost significance. The browser's interface is not only unnecessary but sometimes may get in the way. For this reason, browsers include the Fullscreen API.

## Full Screen

The Fullscreen API lets us expand any element in the document to occupy the entire screen. As a result, the browser’s interface is hidden in the background, allowing the user to focus his or her attention on our video, picture, application or video game.

The API provides a few properties, methods and events to take an element to full screen mode, exit the full screen mode, and retrieve some information from the element and the document that are participating in the process.

**fullscreenElement**—This property returns a reference to the element that is in full screen mode. If no element is full screen, the property returns **null**.

**fullscreenEnabled**—This Boolean property returns **true** when the document is able to go full screen or **false** otherwise.

**requestFullscreen()**—This method is available for every element in the document. It activates full screen mode for the element.

**exitFullscreen()**—This method is available for the document. If an element is in full screen mode, the method cancels the mode and returns the focus back to the browser window.

The API also offers the following events.

**fullscreenchange**—This event is fired by the document when an element enters or exits the full screen mode.

**fullscreenerror**—This event is fired by the element in case of failure (the full screen mode is not available for that element or for the document).

The **requestFullscreen()** method and the **fullscreenerror** event are associated with the elements in the document, but the rest of the properties, methods, and events are part of the **Document** object, and therefore they are accessible from the **document** property.

***Listing 10-1:*** *Taking a* <video> *element full screen*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Full Screen</title>
  <script>
    var video;
    function initiate() {
      video = document.getElementById("media");
      video.addEventListener("click", gofullscreen);
    }
    function gofullscreen() {
      if (!document.webkitFullscreenElement) {
        video.webkitRequestFullscreen();
        video.play();
      }
    }
    window.addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
    <video id="media" width="720" height="400" poster="poster.jpg">
      <source src="trailer.mp4">
```

```
    <source src="trailer.ogg">
   </video>
  </section>
</body>
</html>
```

The **initiate()** function in Listing 10-1 adds a listener for the **click** event to the **<video>** element. The **gofullscreen()** function, set as the listener of this event, is executed every time the user clicks on the video. In this function, we use the **fullscreenElement** property to detect whether an element is already in full screen mode or not, and if not, the video is made full screen by the **requestFullscreen()** method. At the same time, the video is played with the **play()** method.

> **IMPORTANT:** This is an experimental API. The properties, methods, and events have been prefixed by browsers until the final specification is implemented. In the case of Google Chrome, instead of the original names we have to use **webkitRequestFullscreen()**, **webkitExitFullscreen()**, **webkitfullscreenchange**, **webkitfullscreenerror** and **webkitFullscreenElement**. For Mozilla Firefox, the names are **mozRequestFullScreen()**, **mozCancelFullScreen()**, **mozfullscreenchange**, **mozfullscreenerror** and **mozFullScreenElement**.

# Full Screen Styles

Browsers adjust the dimensions of the **<video>** element to the size of the screen automatically, but for any other element, the original dimensions are preserved, and the free space on the screen is filled with a black background. For this reason, CSS includes a pseudo-class called **:full-screen** to modify the styles of the element when it expands to full screen.

> **IMPORTANT:** The last specification declares the name of this pseudo-class without the hyphen (**:fullscreen**), but, at the time of writing, browsers like Mozilla Firefox and Google Chrome have only implemented the previous specification and work with the name

mentioned in this chapter (**:full-screen**). It also has to be prefixed to work properly (**:-webkit-full-screen**, **:-moz-full-screen**, etc.).

***Listing 10-2:*** *Taking any element full screen*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Full Screen</title>
  <style>
    #player:-webkit-full-screen, #player:-webkit-full-screen #media{
      width: 100%;
      height: 100%;
    }
  </style>
  <script>
    var video, player;
    function initiate() {
      video = document.getElementById("media");
      player = document.getElementById("player");
      player.addEventListener("click", gofullscreen);
    }
    function gofullscreen() {
      if (!document.webkitFullscreenElement) {
        player.webkitRequestFullscreen();
        video.play();
      } else {
        document.webkitExitFullscreen();
        video.pause();
      }
    }
    window.addEventListener("load", initiate);
  </script>
</head>
<body>
  <section id="player">
    <video id="media" width="720" height="400" poster="poster.jpg">
```

```
      <source src="trailer.mp4">
      <source src="trailer.ogg">
    </video>
  </section>
</body>
</html>
```

In Listing 10-2, we take the **<section>** element and its content full screen. The **gofullscreen()** function is modified to activate and deactivate the full screen mode for this element. As in the previous example, the video is played when it is in full screen mode, but now it will be paused when the mode is canceled.

These improvements are insufficient to make our player full screen. In full screen mode, the new container for the element is the screen, but the original dimensions and styles for the **<section>** and **<video>** elements are preserved intact. Using the **:full-screen** pseudo-class, we change the values of the **width** and **height** properties of these elements to 100%, matching the dimensions of the container. Now the elements occupy the whole screen and truly reflect the full screen mode.

**Do It Yourself:** Create a new HTML file to test the examples in this chapter. Once the document is opened in the browser, click on the video to activate the full screen mode and click again to deactivate it.

# Chapter 11 - Canvas API

## 11.1 Graphics

In the introduction of this book, we discussed how HTML5 is replacing previous plugins, like Flash. There were at least two important things to consider in making the Web independent of third-party technologies: video processing and graphic applications. The **<video>** element and the APIs studied in previous chapters cover the first aspect very well, but they do not deal with graphics. For this purpose, browsers introduce the Canvas API. This API lets us draw, render graphics, animate, and process images and text, and it works with the rest of the APIs to create full applications and even 2D and 3D video games for the Web.

## The Canvas

The Canvas API can only draw in an area in the document that was designated for that purpose. To define that area, HTML includes the following element.

**<canvas>**—This element creates an area for drawing. It must include the **width** and **height** attributes to determine the dimensions of the area.

The **<canvas>** element generates an empty rectangular space on the page where the results of the methods provided by the API are shown. It produces a white space, like an empty ** ** element, but for an entirely different purpose. The following example demonstrates how to include this element in our document.

***Listing 11-1:*** *Including the* <canvas> *element*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Canvas API</title>
```

```
  <script src="canvas.js"></script>
</head>
<body>
  <section id="canvasbox">
    <canvas id="canvas" width="500" height="300"></canvas>
  </section>
</body>
</html>
```

# The Context

The purpose of the **<canvas>** element is to create an empty box on the screen. To draw something on the canvas, we have to use a context. The API offers the following method for this purpose.

**getContext(**type**)**—This method obtains a drawing context to draw on the canvas. It accepts two values: **2d** (two-dimensional space) and **webgl** (three-dimensional space).

The **getContext()** method is the first method we have to call for the **<canvas>** element to be ready to work. Through the object returned by this method, we are able to apply the rest of the API.

***Listing 11-2:*** *Getting the drawing context for the canvas*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");
}
window.addEventListener("load", initiate);
```

In Listing 11-2, a reference to the **<canvas>** element is stored into the **element** variable, and a reference to the 2D context is obtained with the **getContext()** method. The 2D canvas drawing context returned by this method is a grid of pixels organized in rows and columns from top to bottom and left to right, with its origin (the pixel 0, 0) located at the top-left corner of the rectangle.

**Do It Yourself:** Create a new HTML file with the document of Listing 11-1. Create a file named canvas.js and copy every JavaScript code presented from Listing 11-2 into this file. Each example in this chapter is independent and replaces the previous one. All the images used in this chapter are available on our website.

**The Basics:** Currently, the **2d** context is available in every HTML5 compatible browser, while **webgl** is only applicable in browsers that have implemented and enabled the WebGL library for the generation of 3D graphics. We will study WebGL and 3D graphics in the next chapter.

# 11.2 Drawing

After the **<canvas>** element and its context are ready, we can finally create and manipulate graphics. The list of tools provided by this API for this purpose is extensive, allowing the generation of multiple objects and effects, from simple shapes to text, shadows or complex transformations. In this section of the chapter, we will study these methods one by one.

# Rectangles

Usually, developers must prepare the figure to be drawn before sending it to the context (as we will see soon), but there are a few methods that let us draw directly on the canvas.

**fillRect(**x, y, width, height**)**—This method draws a solid rectangle. The top-left corner is located at the position specified by the **x** and **y** attributes. The **width** and **height** attributes declare the size of the rectangle.

**strokeRect(**x, y, width, height**)**—Similar to the previous method, but this one draws an empty rectangle (just the outline).

**clearRect(**x, y, width, height**)**—This method is used to subtract pixels from the area specified by its attributes. It is like a rectangular eraser.

Drawing rectangles is simple; we have to call the method on the context, and the shapes are displayed on the canvas right away.

***Listing 11-3:*** *Drawing rectangles*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.strokeRect(100, 100, 120, 120);
  canvas.fillRect(110, 110, 100, 100);
  canvas.clearRect(120, 120, 80, 80);
}
```

window.addEventListener("load", initiate);

In Listing 11-3, the context is assigned to the variable **canvas**, and from this reference, we call the drawing methods. The first method, **strokeRect(100, 100, 120, 120)**, draws an empty rectangle with the top-left corner at the position **100,100** and a size of 120 pixels. The second method, **fillRect(110, 110, 100, 100)**, draws a solid rectangle, this time starting from the position **110,110** of the canvas. And finally, with the last method, **clearRect(120, 120, 80, 80)**, a square of 80 pixels is removed from the middle of the previous rectangle.

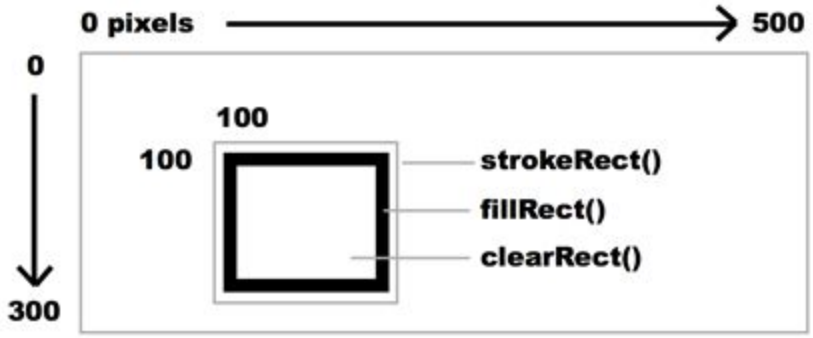


***Figure 11-1:*** *Rectangles on the canvas*

Figure 11-1 is a representation of what we see after the execution of the code in Listing 11-3. The **<canvas>** element is rendered as a grid with its origin at the top-left corner and the size specified by its attributes. The rectangles are drawn on the canvas at the position declared by the attributes **x** and **y** and one on top of another according to the order in the code. The first one to appear in the code is drawn first, the second is drawn on top of it, and so on (There is a method for customizing how the shapes are drawn, but we will see that later).

# Colors

Up to this point, we have been using the default color: black. We can specify the color we want to use with CSS syntax and the following properties.

**strokeStyle**—This property declares the color of the lines of the shape.

**fillStyle**—This property declares the color of the interior of the shape.

**globalAlpha**—This property sets the transparency of the shape.

The colors are defined with the same values we use in CSS between quotes.

***Listing 11-4:*** *Changing colors*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.fillStyle = "#000099";
  canvas.strokeStyle = "#990000";

  canvas.strokeRect(100, 100, 120, 120);
  canvas.fillRect(110, 110, 100, 100);
  canvas.clearRect(120, 120, 80, 80);
}
window.addEventListener("load", initiate);
```

The colors in Listing 11-4 are declared using hexadecimal numbers, but we can also use functions such as **rgb()** or even specify transparency with the **rgba()** function. These functions also have to be declared between quotes, as in **strokeStyle = "rgba(255, 165, 0, 1)"**. When a new color is specified, it becomes the default color for later drawings.

**The Basics:** Besides the **rgba()** function, the context provides a property to set the level of transparency called **globalAlpha**. The possible values for this property are between 0.0 (fully opaque) and 1.0 (fully transparent).

# Gradients

Gradients are part of the toolkit of every drawing application, and the Canvas API is no exception. As in CSS, the gradients on the canvas may be linear or radial, and we can provide stop points to combine colors.

**createLinearGradient(**x1, y1, x2, y2**)**—This method creates a linear gradient object to apply to the canvas.

**createRadialGradient(**x1, y1, r1, x2, y2, r2**)**—This method creates a radial gradient object to apply to the canvas using two circles. The values represent the position of the center of each circle and their radius.

**addColorStop(**position, color**)**—This method specifies the colors to create the gradient. The **position** attribute is a value between 0.0 and 1.0 to determine where the degradation starts for that particular color.

The following example illustrates how to apply a linear gradient to our canvas.

***Listing 11-5:*** *Applying a linear gradient to the canvas*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  var gradient = canvas.createLinearGradient(0, 0, 10, 100);
  gradient.addColorStop(0.5, "#00AAFF");
  gradient.addColorStop(1, "#000000");
  canvas.fillStyle = gradient;

  canvas.fillRect(10, 10, 100, 100);
  canvas.fillRect(150, 10, 200, 100);
}
window.addEventListener("load", initiate);
```

In Listing 11-5, we create the gradient object from the position **0,0** to **10,100**, producing a slight inclination to the left. The colors are set by the **addColorStop()** methods, and the final gradient is applied with the **fillStyle** property, like a regular color.

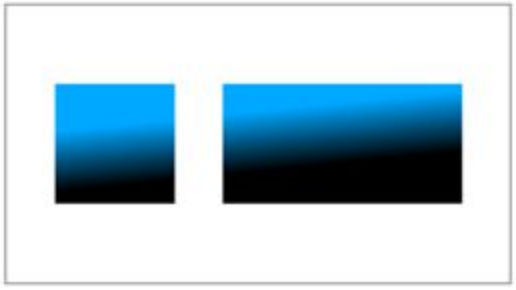

***Figure 11-2:*** *Linear gradient for the canvas*

Notice that the positions of the gradient are relative to the canvas, not to the shapes we want to affect. The result is that, if we move the rectangles to a new position on the screen, the gradient for those rectangles changes.

**Do It Yourself:** The radial gradient is similar to CSS. Try to replace the linear gradient in the code in Listing 11-5 with a radial gradient using an instruction such as **createRadialGradient(0, 0, 30, 0, 0, 300)**. You can also experiment with the location of the rectangles to see how the gradient is applied.

# Paths

The methods studied above draw directly on the canvas, but that is not always the case. When we have to draw complex figures, we have to process shapes and images in the background, and then send the result to the context to be drawn. For this purpose, the Canvas API introduces several methods to generate paths.

A path is like a blueprint for a pen to follow. Once we set a path, it has to be sent to the context to be drawn permanently on the canvas. The path may include different kinds of strokes to create complex shapes, such as straight lines, arcs, rectangles and others. The following are the methods to begin and close a path.

**beginPath()**—This method begins a new path.

**closePath()**—This method closes the path, generating a straight line from the last point to the point of origin. It can be omitted when we want an open path or when we use the **fill()** method to draw the path.

We also have three methods to draw the path on the canvas.

**stroke()**—This method draws the path as an outlined shape.

**fill()**—This method draws the path as a solid shape. When we use this method, we do not need to close the path with **closePath()**; it is automatically closed with a straight line from the last point to the first one.

**clip()**—This method sets a new clipping area for the context. When the context is initialized, the clipping area is the entire area occupied by the canvas. The **clip()** method changes the clipping area to a new shape, creating a mask. Everything that falls outside that mask is not drawn.

Every time we want to create a path, we have to call the **beginPath()** method to initialize it, as in the following example.

***Listing 11-6:*** *Starting and ending a path*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.beginPath();
  // here goes the path
  canvas.stroke();
}
window.addEventListener("load", initiate);
```

The code in Listing 11-6 does not create anything, it simply begins the path and draws it with the **stroke()** method. Nothing is drawn because we have not yet define the path. The following are the methods available to set the path and create the shape.

**moveTo(**x, y**)**—This method moves the pen to a new position. It lets us start or continue the path from different points on the grid, avoiding continuous lines.

**lineTo(**x, y**)**—This method generates a straight line from the pen's current position to the one declared by the **x** and **y** attributes.

**rect(**x, y, width, height**)**—This method generates a rectangle. Unlike the methods studied before, this method generates a rectangle that is part of the path (not directly drawn on the canvas). The attributes have the same function as previous methods.

**arc(**x, y, radius, startAngle, endAngle, direction**)**—This method generates an arc or a circle with the center at the coordinates indicated by **x** and **y** and with the radius and angles declared by the rest of the attributes. The last value has to be declared as **true** or **false** for the direction (counterclockwise or clockwise, respectively).

**quadraticCurveTo(**cpx, cpy, x, y**)**—This method generates a quadratic Bézier curve starting from the current position of the pen and ending at the position declared by the **x** and **y** attributes. The **cpx** and **cpy** attributes define the control point that shapes the curve.

**bezierCurveTo(**cp1x, cp1y, cp2x, cp2y, x, y**)**—This method is similar to the previous method but adds two more attributes to generate a cubic Bézier curve. It generates two control points in the grid declared by the values **cp1x**, **cp1y**, **cp2x** and **cp2y** to shape the curve.

The following code generates a simple path to illustrate how these methods work.

***Listing 11-7:*** *Creating a path*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);
  canvas.stroke();
}
window.addEventListener("load", initiate);
```

It is always recommended to set the initial position of the pen immediately after starting the path. In the code in Listing 11-7, we first move the pen to the position **100,100** and then generate a line from this point to **200,200**. The position of the pen is now **200,200**, and the next line is drawn from there to the point **100,200**. Finally, the path is drawn as an outline shape by the **stroke()** method.

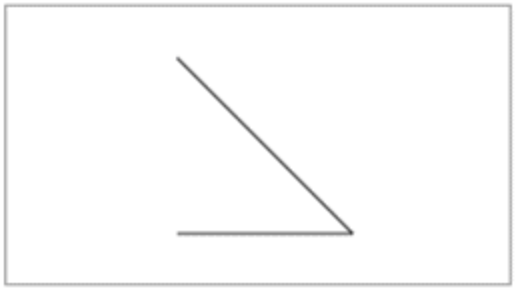

***Figure 11-3:*** *Open path*

Figure 11-3 shows a representation of the open triangle produced by the code in Listing 11-7. This triangle may be closed or even filled using different methods, as shown in the following examples.

***Listing 11-8:*** *Completing the triangle*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);

  canvas.closePath();
  canvas.stroke();
}
window.addEventListener("load", initiate);
```

The **closePath()** method adds a straight line to the path, from the end point to the start point, closing the shape.

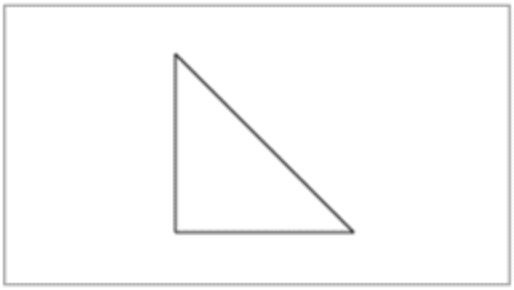

***Figure 11-4:*** *Close path*

Using the **stroke()** method at the end of our path, we drew an empty triangle on the canvas. If we want a solid triangle, we have to use the **fill()** method.

***Listing 11-9:*** *Drawing a solid triangle*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);
  canvas.fill();
}
window.addEventListener("load", initiate);
```

Now the figure on the screen is a solid triangle. The **fill()** method closes the path automatically, so we no longer have to use the **closePath()** method.

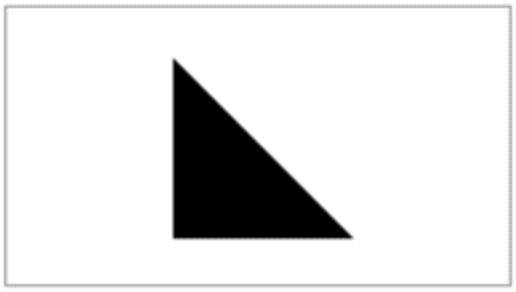

***Figure 11-5:*** *Solid triangle*

One of the methods previously introduced to draw a path on the canvas was **clip()**. This method does not draw anything, but it creates a mask with the shape of the path to select what will be drawn and what will not be drawn. Everything that falls outside the mask is not drawn.

***Listing 11-10:*** *Using a triangle as a mask*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);
  canvas.clip();

  canvas.beginPath();
  for (var f = 0; f < 300; f = f + 10) {
    canvas.moveTo(0, f);
    canvas.lineTo(500, f);
  }
  canvas.stroke();
}
window.addEventListener("load", initiate);
```

To show how the **clip()** method works, in Listing 11-10 we create a **for** loop to generate horizontal lines every 10 pixels. The lines go from the left

to the right side of the canvas, but only the parts of the lines that fall inside the triangle mask are drawn.

***Figure 11-6:*** *Clipping area*

Now that we know how to draw paths, it is time to look at other ways to create shapes. Thus far, we have generated straight lines and square shapes. For circular shapes, the API provides three methods: **arc()**, **quadraticCurveTo()** and **bezierCurveTo()**. The first one is relatively simple to use and can generate partial or full circles, as shown in the following example.

***Listing 11-11:*** *Drawing circles with the* arc() *method*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.beginPath();
  canvas.arc(100, 100, 50, 0, Math.PI * 2, false);
  canvas.stroke();
}
window.addEventListener("load", initiate);
```

In the **arc()** method in Listing 11-11 we use the value PI from the **Math** object to specify the angle. This is necessary because this method uses radians instead of degrees. In radians, the value of PI represents 180 degrees, so the formula **PI × 2** multiplies PI by 2 getting an angle of 360 degrees.

This example generates an arc with a center at the point **100,100** and a radius of 50 pixels, starting at **0** degrees and ending at **Math.PI * 2** degrees,

which represents a full circle.

If we need to calculate the value in radians from degrees, we have to use the formula: **Math.PI / 180 × degrees**, as in the following example.

***Listing 11-12:*** *Drawing an arc of 45 degrees*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  var radians = Math.PI / 180 * 45;

  canvas.beginPath();
  canvas.arc(100, 100, 50, 0, radians, false);
  canvas.stroke();
}
window.addEventListener("load", initiate);
```

With the code in Listing 11-12 we get an arc that covers 45 degrees of a circle, but if we change the value of the direction to **true** in the **arc()** method, the arc is generated from 0 degrees to 315, creating an open circle, as illustrated in Figure 11-7.

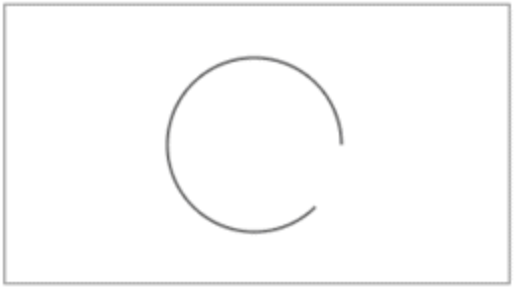

***Figure 11-7:*** *Semicircle with the* arc() *method*

**The Basics:** If you continue working with this path, the current starting point will be the end of the arc. If you do not want to start at the end of the arc, you will have to use the **moveTo()** method to change the position of the pen. However, if the next shape is another arc, you have to remember that the **moveTo()** method moves the virtual pen to the point at which the circle begins to be drawn, not to the center of the circle.

Besides **arc()**, we have two more methods for drawing more complex curves. The **quadraticCurveTo()** method generates a quadratic Bézier curve, and the **bezierCurveTo()** method generates a cubic Bézier curve. The difference between the methods is that the first one has only one point of control and the second method has two, thus creating different types of curves.

***Listing 11-13:*** *Creating complex curves*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.beginPath();
  canvas.moveTo(50, 50);
  canvas.quadraticCurveTo(100, 125, 50, 200);

  canvas.moveTo(250, 50);
  canvas.bezierCurveTo(200, 125, 300, 125, 250, 200);
  canvas.stroke();
}
window.addEventListener("load", initiate);
```

To create the quadratic curve in this example, we move the virtual pen to the position **50,50** and finish the curve at the position **50,200**. The control point for this curve is at the position **100,125**. The cubic curve generated by the **bezierCurveTo()** method is more complicated. There are two control points for this curve, the first one at the position **200,125** and the second one at the position **300,125**. The result is illustrated in Figure 11-8.

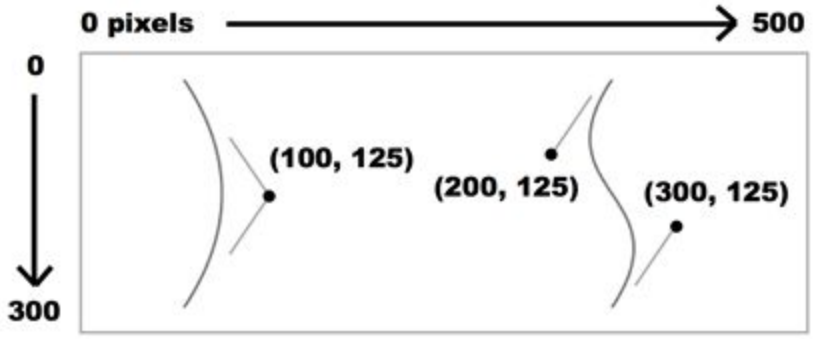


***Figure 11-8:*** *Representation of Bézier curves and their control points on the canvas*

The values in Figure 11-8 indicate the position of the control points for the curves. Moving those points changes the shape of the curve.

**Do It Yourself:** You can add as many curves as you need to build your shape. Try to change the values of the control points in Listing 11-13 to see how they affect the curves. Build more complex shapes by combining curves and lines to understand how a path is made.

# Lines

Up to this point, we have been using the same style for every line drawn on the canvas. The width, ending, and other aspects of the line can be manipulated to get exactly the type of line we need for our drawings. For this purpose, the API includes four properties.

**lineWidth**—This property determines the line thickness. By default, the value is 1.0.

**lineCap**—This property determines the shape of the end of the line. There are three possible values: **butt**, **round**, and **square**.

**lineJoin**—This property determines the shape of the connection between two lines. The possible values are **round**, **bevel** and **miter**.

**miterLimit**—This property determines how far the connections between lines will be extended when the **lineJoin** property is set to **miter**.

These properties affect the entire path. Every time we have to change the characteristics of the lines, we have to create a new path with new property values.

***Listing 11-14:*** *Testing properties for lines*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.beginPath();
  canvas.arc(200, 150, 50, 0, Math.PI * 2, false);
  canvas.stroke();
  canvas.lineWidth = 10;
  canvas.lineCap = "round";
  canvas.beginPath();
  canvas.moveTo(230, 150);
  canvas.arc(200, 150, 30, 0, Math.PI, false);
  canvas.stroke();

  canvas.lineWidth = 5;
  canvas.lineJoin = "miter";
  canvas.beginPath();
  canvas.moveTo(195, 135);
  canvas.lineTo(215, 155);
  canvas.lineTo(195, 155);
  canvas.stroke();
}
window.addEventListener("load", initiate);
```

We start the drawing in the example of Listing 11-14 by defining a path for a full circle with the properties by default. Then, using **lineWith**, we change the width of the line to **10** and set the **lineCap** to **round**. This makes the path thick with rounded endings, which will create a smiling mouth. To create the path, we move the pen to the position **230,150** and then generate a semicircle. Finally, we add another path created by two lines to form a shape like a nose. Notice that the lines for this path have a width of **5** and

are joined with the property **lineJoin** and the value **miter**. This property makes the nose pointy, expanding the outside edges of the corner until they reach a single point.

***Figure 11-9:*** *Different types of lines*

**Do It Yourself:** Experiment with changing the lines for the nose modifying the property **miterLimit** (e.g., **miterLimit = 2**). Change the value of the **lineJoin** property to **round** or **bevel**. You can also modify the shape of the mouth by trying different values for the property **lineCap**.

# Text

Writing text on the canvas is as simple as defining a few properties and calling the appropriate method. There are three properties to configure text.

**font**—This property sets the type of font to be used for the text. It takes the same values as the **font** property from CSS.

**textAlign**—This property aligns the text horizontally. The alignment can be made to the **start**, **end**, **left**, **right** or **center**.

**textBaseline**—This property is for vertical alignment. It sets different positions for the text (including Unicode text). The possible values are **top**, **hanging**, **middle**, **alphabetic**, **ideographic**, and **bottom**.

The following are the methods available to draw the text.

**strokeText(**text, x, y**)**—This method draws the specified text as an outline shape at the position **x** and **y**. It can also include a fourth value to

declare the maximum size. If the text is longer than this value, it will be shrunk to fit into that space.

**fillText(**text, x, y**)**—This method is similar to the previous method except it draws a solid text.

The following example illustrates how to draw a single text with a custom font and in a specific position on the canvas.

***Listing 11-15:*** *Drawing text*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.font = "bold 24px verdana, sans-serif";
  canvas.textAlign = "start";
  canvas.fillText("My message", 100, 100);
}
window.addEventListener("load", initiate);
```

The **font** property can take several values at once using exactly the same syntax as CSS. On the other hand, the **textAlign** property indicates that the text has to begin at the position **100,100** (if the value of this property was **end**, for example, the text would have ended at the position **100,100**). Finally, the **fillText()** method draws a solid text on the canvas.

Besides the methods already mentioned, the API provides another important method to work with text.

**measureText(**text**)**—This method returns information about the size of the text between parentheses.

The **measureText()** method can be useful to calculate positions and even collisions in animations, as well as combining text with other shapes on the canvas, as in the following example.

***Listing 11-16:*** *Measuring text*

```
function initiate() {
```

```
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.font = "bold 24px verdana, sans-serif";
  canvas.textAlign = "start";
  canvas.textBaseline = "bottom";
  canvas.fillText("My message", 100, 124);

  var size = canvas.measureText("My message");
  canvas.strokeRect(100, 100, size.width, 24);
}
window.addEventListener("load", initiate);
```

In this example, we begin with the same code used in the example of Listing 11-15, but this time the **textBaseline** is set to **bottom**, which means that the lower part of the text will be at the position **124**. This way, we know the exact vertical position of the text on the canvas. Next, we read the **width** property in the object returned by the **measureText()** method to get the text's width. With all the measures taken, we can now draw a rectangle surrounding the text.


***Figure 11-10:*** *Working with text*

**Do It Yourself:** Using the code in Listing 11-16, test different values for the **textAlign** and **textBaseline** properties. Use the rectangle as a reference to see how they work. Write different texts to see how the rectangle automatically adapts to each size.

**IMPORTANT:** The **measureText()** method returns an object with several properties. The **width** property used in our example is only one of

them. For a complete list, read the specification of the Canvas API. The link is available on our website.

# Shadows

Shadows are, of course, also an important part of the Canvas API. We can generate shadows for every path, including texts. The API provides four properties to define a shadow.

**shadowColor**—This property declares the color of the shadow using CSS syntax.

**shadowOffsetX**—This property determines how far the shadow will be from the object in the horizontal direction.

**shadowOffsetY**—This property determines how far the shadow will be from the object in the vertical direction.

**shadowBlur**—This property produces a blurring effect for the shadow.

The properties must be defined before the path is drawn on the canvas. The following example generates a shadow for a text.

***Listing 11-17:*** *Applying shadows to text*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.shadowColor = "rgba(0, 0, 0, 0.5)";
  canvas.shadowOffsetX = 4;
  canvas.shadowOffsetY = 4;
  canvas.shadowBlur = 5;

  canvas.font = "bold 50px verdana, sans-serif";
  canvas.fillText("My message", 100, 100);
}
window.addEventListener("load", initiate);
```

The shadow created in Listing 11-17 is using the **rgba()** function to get a semi-transparent black color. It is displaced 4 pixels from the text and has a blur value of 5.


***Figure 11-11:*** *Text with a shadow*

# Transformations

The canvas may be transformed, which affects the figures that are drawn later. The following are the methods available to perform these operations.

**translate(**x, y**)**—This method is used to move the origin of the canvas.

**rotate(**angle**)**—This method rotates the canvas around the origin as many radians as specified by the attribute.

**scale(**x, y**)**—This method increases or decreases the units of the canvas to reduce or enlarge everything drawn on it. The scale can be changed independently for horizontal or vertical values using the **x** and **y** attributes. The values may be negative, producing a mirror effect. By default, the values are set to 1.0.

**transform(**m1, m2, m3, m4, dx, dy**)**—This method applies a new matrix over the current one to modify the properties of the canvas.

**setTransform(**m1, m2, m3, m4, dx, dy**)**—This method resets the current transformation matrix and sets a new one from the values provided by the attributes.

Every canvas has the point 0, 0 (the origin) located at the top-left corner, with the values increasing in any direction inside the canvas (negative values are outside the canvas). The **translate()** method lets us move the origin to a new position to use it as a reference for our drawings. In the

following example, we move the origin two times to change the position of a text.

***Listing 11-18:*** *Translating, rotating and scaling*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("TEST", 50, 20);

  canvas.translate(50, 70);
  canvas.rotate(Math.PI / 180 * 45);
  canvas.fillText("TEST", 0, 0);

  canvas.rotate(-Math.PI / 180 * 45);
  canvas.translate(0, 100);
  canvas.scale(2, 2);
  canvas.fillText("TEST", 0, 0);
}
window.addEventListener("load", initiate);
```

In Listing 11-18, we apply the **translate()**, **rotate()** and **scale()** methods to the same text. First, we draw a text on the canvas with the state of the canvas by default. The text appears at the position **50,20** with a size of 20 pixels. By using the **translate()** method, the canvas origin is moved to the position **50,70**, and the entire canvas is rotated 45 degrees with the **rotate()** method. In consequence, the next text is drawn at the new origin with an inclination of 45 degrees. The transformations applied to the canvas become the default values, so to test the **scale()** method, we rotate the canvas 45 degrees back to its default position and again translate the origin down 100 pixels. Finally, the scale is duplicated and another text is drawn, this time twice the size of the original.


***Figure 11-12:*** *Applying transformations*

**IMPORTANT:** Every transformation is cumulative. If we perform two transformations using **scale()**, for example, the second method performs the scaling using the current status. A **scale(2,2)** after another **scale(2,2)** will quadruple the scale of the canvas. This applies to every transformation, including the methods that transform the matrix, as we will see next.

The canvas has a matrix of values that define its properties. Modifying this matrix, we can change the characteristics of the canvas. The API offers two methods for this purpose: **transform()** and **setTransform()**.

***Listing 11-19:*** *Cumulative transformation of the matrix*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.transform(3, 0, 0, 1, 0, 0);
  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("TEST", 20, 20);

  canvas.transform(1, 0, 0, 10, 0, 0);
  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("TEST", 20, 20);
}
window.addEventListener("load", initiate);
```

As in the previous example, in Listing 11-19 we apply transformation methods to the same text to compare effects. The default values of the

canvas matrix are **1, 0, 0, 1, 0, 0**. By changing the first value to **3** in the first transformation, we stretch the canvas horizontally. The text drawn after this transformation is wider than the text drawn in normal conditions.

With the next transformation in the code, the canvas is stretched vertically by changing the fourth value to **10** and preserving the rest as is. The result is illustrated below.


***Figure 11-13:*** *Modifying the transformation matrix*

One important thing to remember is that transformations are applied to the matrix set by a previous transformation, so the second text shown by the code in Listing 11-19 is stretched horizontally and vertically, as shown in Figure 11-13. To reset the matrix and set new transformation values, we can use the **setTransform()** method.

# Status

The accumulation of transformations makes it dificult to return to previous states. In the code in Listing 11-18, for example, we had to remember the value of the rotation to perform a new rotation and undo the previous transformations. Considering this situation, the Canvas API provides two methods to save and retrieve the canvas state.

**save()**—This method saves the canvas state including transformations already applied, values of styling properties, and the current clipping path (the area created by the **clip()** method, if any).

**restore()**—This restores the last state saved.

Every time we want to go back to a previous state, we have to store the state we want to preserve with the **save()** method and then get it back with

the **restore()** method. Any change performed on the canvas in the middle will not affect the drawings after the state is restored.

***Listing 11-20:*** *Saving and restoring the canvas state*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.save();
  canvas.translate(50, 70);
  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("TEST1", 0, 30);
  canvas.restore();
  canvas.fillText("TEST2", 0, 30);
}
window.addEventListener("load", initiate);
```

After executing the JavaScript code in Listing 11-20, the text "TEST1" is drawn in big letters at the center of the canvas and "TEST2" in a smaller size, close to the origin. What we do in this example is to save the canvas state and then set a new position for the origin and styles for the text. Before drawing the second text on the canvas, the original state is restored. As a result, the second text is shown with the styles by default, not with those declared for the first text.

It does not matter how many transformations are performed on the canvas; the state will return exactly to the previous condition when the **restore()** method is called.

# The GlobalCompositeOperation Property

When we were talking about paths, we mentioned that there is a property to determine how a shape is positioned and combined with previous shapes on the canvas. That property is **globalCompositeOperation** and its default value is **source-over**, which means that the new shape will be drawn over those already on the canvas. There are 11 more values available.

**source-in**—Only the part of the new shape that overlaps the existing shape is drawn. The rest of the shape and the rest of the existing shape are made transparent.

**source-out**—Only the part of the new shape that does not overlap the existing shape is drawn. The rest of the shape and the rest of the existing shape are made transparent.

**source-atop**—Only the part of the new shape that overlaps the existing shape is drawn. The existing shape is preserved, but the rest of the new shape is made transparent.

**lighter**—Both shapes are drawn, but the color of the parts that overlap is determined by adding the values of the colors.

**xor**—Both shapes are drawn, but the parts that overlap are made transparent.

**destination-over**—This is the opposite of the default value (**source-over**). The new shape is drawn behind the existing shapes on the canvas.

**destination-in**—The parts of the existing shape on the canvas that overlap with the new shape are preserved. The rest, including the new shape, are made transparent.

**destination-out**—The parts of the existing shape on the canvas that do not overlap with the new shape are preserved. The rest, including the new shape, are made transparent.

**destination-atop**—The existing shape and the new shape are only preserved where they overlap.

**darker**—Both shapes are drawn, but the color of the parts that overlap is determined by subtracting the values of the colors.

**copy**—Only the new shape is drawn, and the existing shape is made transparent.

As with most properties of this API, we first have to define the property and then draw the path on the canvas.

***Listing 11-21:*** *Testing the* globalCompositeOperation *property*

```
function initiate() {
```

```
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  canvas.fillStyle = "#666666";
  canvas.fillRect(100, 100, 200, 80);

  canvas.globalCompositeOperation = "source-atop";

  canvas.fillStyle = "#DDDDDD";
  canvas.font = "bold 60px verdana, sans-serif";
  canvas.textAlign = "center";
  canvas.textBaseline = "middle";
  canvas.fillText("TEST", 200, 100);
}
window.addEventListener("load", initiate);
```

Only visual representations of each possible value for the **globalCompositeOperation** property can help us understand how they work, so we prepared the example of Listing 11-21 with this purpose in mind. When this code is executed, a red rectangle is drawn in the middle of the canvas, but as a result of the **source-atop** value, only the part of the text that overlaps the rectangle is drawn on the screen.

***Figure 11-14:*** *Applying* globalCompositeOperation

**Do It Yourself:** Replace the value **source-atop** with any of the other values available for this property and check the result in your browser. Test your code in different browsers.

# 11.3 Images

The Canvas API would be useless without the capacity to process images, but despite the importance of images, only one method is required to draw them on the canvas. However, there are three versions of this method available that produce different results. The following are the possible combinations.

**drawImage(**image, x, y**)**—This syntax is used to draw an image on the canvas at the position declared by the **x** and **y** attributes. The first attribute is a reference to the image, which can be a reference to an **<img>** element, a **<video>** element, or another **<canvas>** element.

**drawImage(**image, x, y, width, height**)**—This syntax lets us scale the image before drawing it on the canvas, changing its size to the values of the **width** and **height** attributes.

**drawImage(**image, x1, y1, width1, height1, x2, y2, width2, height2**)**—This is the most complex syntax. There are two values for every parameter. The purpose is to slice part of an image and then draw it on the canvas with a customized size and position. The attributes **x1** and **y1** set the top-left corner of the portion of the image to be sliced. The attributes **width1** and **height1** indicate the size of the slice. The rest of the attributes (**x2**, **y2**, **width2** and **height2**) declare where the slice will be drawn on the canvas and its size (which can be different from the original).

In every case, the first attribute is always a reference to an image or a media element, including another canvas. It is not possible to use a URL or load files from external sources directly with this method; we first have to create an **<img>** element to load the image and then call the method with a reference to this element, as in the following example.

***Listing 11-22:*** *Drawing an image*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");
```

```
  var image = document.createElement("img");
  image.src = "snow.jpg";
  image.addEventListener("load", function() {
    canvas.drawImage(image, 20, 20);
  });
}
window.addEventListener("load", initiate);
```

The code in Listing 11-22 loads an image and draws it on the canvas. Because the canvas can only receive images that are already loaded, we need to monitor this situation checking the **load** event. After the image is created by the **createElement()** method and loaded, it is drawn at the position **20**, **20** with the **drawImage()** method.

***Figure 11-15:*** *Image on the canvas*

The following example illustrates how to resize an image adding more attributes to the **drawImage()** method.

***Listing 11-23:*** *Adjusting the image to the size of the canvas*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  var image = document.createElement("img");
  image.src = "snow.jpg";
  image.addEventListener("load", function() {
```

```
    canvas.drawImage(image, 0, 0, element.width, element.height);
  });
}
window.addEventListener("load", initiate);
```

In Listing 11-23, the size of the image is determined from the **width** and **height** properties of the **<canvas>** element, so the image is stretched by the method to cover the entire canvas.

***Figure 11-16:*** *Image stretched to cover the canvas*

The next example implements the most complex syntax available for the **drawImage()** method to extract a portion of the original image and draw it on the canvas.

***Listing 11-24:*** *Extracting, resizing and drawing*

```
function initiate() {
  var element = document.getElementById("canvas");
  var canvas = element.getContext("2d");

  var image = document.createElement("img");
  image.src = "snow.jpg";
  image.addEventListener("load", function() {
    canvas.drawImage(image, 135, 30, 50, 50, 0, 0, 300, 300);
  });
}
window.addEventListener("load", initiate);
```

In this example, we take a square of the original image starting from the position **135,50**, and with a size of **50,50** pixels. The block is resized to **300,300** pixels and, finally, drawn on the canvas at the position **0,0**.

***Figure 11-17:*** *A portion of the image* on the canvas

# Patterns

Patterns let us add texture to the shapes using an image. The procedure is similar to the one we used with gradients; the pattern is created and then applied to the path as a color. The following is the method included in the API to create a pattern.

**createPattern(**image, type**)**—This method creates and returns an object representing a pattern. The **image** attribute is a reference to the image we want to use for the pattern, and the **type** attribute determines its type. The possible values are **repeat**, **repeat-x**, **repeat-y**, and **no-repeat**.

As well as with gradients, the object representing the pattern must be created and then assigned to the **fillStyle** property of the canvas.

***Listing 11-25:*** *Adding a pattern to our paths*

```
var canvas, image;
function initiate() {
  var element = document.getElementById("canvas");
  canvas = element.getContext("2d");
  image = document.createElement("img");
  image.src = "bricks.jpg";
```

```
  image.addEventListener("load", modimage);
}
function modimage() {
  var pattern = canvas.createPattern(image, "repeat");
  canvas.fillStyle = pattern;
  canvas.fillRect(0, 0, 500, 300);
}
window.addEventListener("load", initiate);
```

The code in Listing 11-25 creates a rectangle of the size of the canvas and fills it with a pattern using the image bricks.jpg.

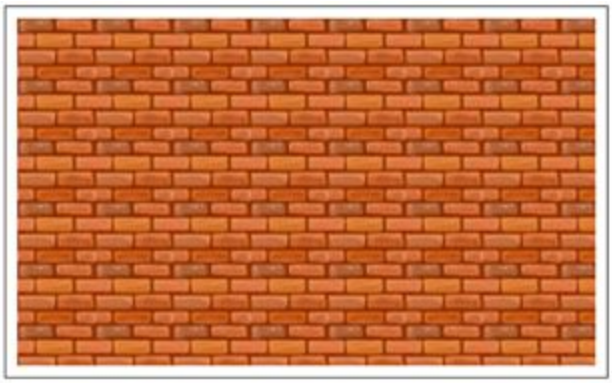

***Figure 11-18:*** *Pattern*

**Do It Yourself:** Update your canvas.js file with the code in Listing 11-25. Download the file bricks.jpg from our website and open the document of Listing 11-1 in your browser. You should see something similar to Figure 11-18. Experiment with the different values available for the **createPattern()** method and assign patterns to other shapes.

# Image Data

Earlier, we stated that **drawImage()** was the only method available to draw images on the canvas, but that is not entirely correct. There are a few methods to process images that can also draw them on the canvas; however, these methods work with data not images.

**getImageData(**x, y, width, height**)**—This method takes a rectangle of the canvas of the size declared by its attributes and converts it to data. It returns an object with the properties **width**, **height** and **data**.

**putImageData(**imagedata, x, y**)**—This method turns the data declared by the **imagedata** attribute into an image and draws the image on the canvas at the position specified by the **x** and **y** attributes. It is the opposite of **getImageData()**.

Every image can be described as a sequence of integers representing RGBA values. These are four values that define the levels of color red, green, blue and alpha (transparency) for each pixel. This information results in a one-dimensional array that can be used to generate an image. The position of every value in the array is calculated by the formula **(width × 4 × y) + (x × 4) + n**, where **n** is an index number for the values of the pixel, starting at 0. The formula for red is **(width × 4 × y) + (x × 4) + 0**; for green is **(width × 4 × y) + (x × 4) + 1**; for blue is **(width × 4 × y) + (x × 4) + 2**; and for the alpha value is **(width × 4 × y) + (x × 4) + 3**. The following example implements these formulas to create the negative of an image.

***Listing 11-26:*** *Generating the negative of an image*

```
var canvas, image;
function initiate() {
  var element = document.getElementById("canvas");
  canvas = element.getContext("2d");
  image = document.createElement("img");
  image.src = "snow.jpg";
  image.addEventListener("load", modimage);
}
function modimage() {
  canvas.drawImage(image, 0, 0);
  var info = canvas.getImageData(0, 0, 175, 262);
  var pos;
  for (var x = 0; x < 175; x++) {
    for (var y = 0; y < 262; y++) {
      pos = (info.width * 4 * y) + (x * 4);
      info.data[pos] = 255 - info.data[pos];
```

```
      info.data[pos+1] = 255 - info.data[pos+1];
      info.data[pos+2] = 255 - info.data[pos+2];
    }
  }
  canvas.putImageData(info, 0, 0);
}
window.addEventListener("load", initiate);
```

**IMPORTANT:** These methods present limitations for cross-origin access. The files for this example have to be uploaded to a server or a local server to work properly (including the snow.jpg image that you can download from our website). We will study cross-origin access next.

In Listing 11-26, we create a new function to process the image called **modimage()**. This function draws the image on the canvas at the position **0,0** using the **drawImage()** method and then processes the image data pixel by pixel.

The image from our example is 350 pixels wide by 262 pixels tall. Using the **getImageData()** method with the values **0,0** for the top-left corner and **175,262** for the horizontal and vertical dimensions, we extract the left half of the original image. The data obtained is stored in the **info** variable and then processed to get the desired effect.

Because every color is declared by a value between 0 and 255 (one byte), the negative value is obtained by subtracting the original value from 255 using the formula **color = 255 - color**. To do this for every pixel in our image, we create two **for** loops, one for columns and another for rows. Notice that the **for** loop for the **x** values ranges from 0 to 174 (the width of the part of the image we cut from the canvas), and the **for** loop for the **y** values ranges from 0 to 261 (the total number of vertical pixels in the image we are processing).

After every pixel is processed, the data contained by the **info** variable is transferred to the canvas using the **putImageData()** method. This new image is located at the same position than the original, replacing the left half of our original image with the negative we have just created.

***Figure 11-19:*** *Negative image*

# Cross-Origin

A cross-origin application is an application that is located in one domain and is accessing resources from another. Because of security concerns, some APIs restrict cross-origin access. In the case of the Canvas API, no information can be retrieved from the **<canvas>** element after an image from another domain has been drawn.

These restrictions may be avoided using a technology called *CORS* (Cross-Origin Resource Sharing). CORS defines a protocol for servers to share their resources and allow requests from other origins. The access from one origin to another must be authorized by the server. The authorization is made by declaring in the server the origins (domains) allowed to access the resources. This is done in the header sent by the server that allocates the file processing the request. For example, if our application is located at www.domain1.com and is accessing resources from www.domain2.com, this second server must be configured to declare the origin www.domain1.com as a valid origin for the request.

CORS provides several headers to be included as part of the HTTP header sent by the server, but the only one required is **Access-Control-Allow-Origin**. This header indicates what origins (domains) are able to access the server's resources. The * character may be declared to allow requests from any origin.

**IMPORTANT:** Your server must be configured to send CORS HTTP headers with every request to allow applications from other domains to access its files. An easy way to do this is to append a new line to the

**.htaccess** file. Most servers provide this configuration file in the root folder of every website. The syntax is **Header set CORS-Header value** (e.g., **Header set Access-Control-Allow-Origin ***). For more information on how to add HTTP headers to your server, visit our website and follow the links for this chapter.

Adding the HTTP headers to the server's configuration is only one of the steps we have to take to turn the codes into cross-origin applications. The codes also have to declare the files as a cross-origin resource using the **crossOrigin** attribute.

***Listing 11-27:*** *Enabling cross-origin access*

```
var canvas, image;
function initiate() {
  var element = document.getElementById("canvas");
  canvas = element.getContext("2d");

  image = document.createElement("img");
  image.crossOrigin = "anonymous";
  image.src = "http://www.formasterminds.com/content/snow.jpg";
  image.addEventListener("load", modimage);
}
function modimage() {
  canvas.drawImage(image, 0, 0);
  var info = canvas.getImageData(0, 0, 175, 262);
  var pos;
  for (var x = 0; x < 175; x++) {
    for (var y = 0; y < 262; y++) {
      pos = (info.width * 4 * y) + (x * 4);
      info.data[pos] = 255 - info.data[pos];
      info.data[pos+1] = 255 - info.data[pos+1];
      info.data[pos+2] = 255 - info.data[pos+2];
    }
  }
  canvas.putImageData(info, 0, 0);
}
window.addEventListener("load", initiate);
```

The **crossOrigin** attribute can take two values: **anonymous** or **use-credentials**. The first value ignores credentials while the second value requires credentials to be sent in the request. Credentials are shared automatically by the client and the server using cookies. To use credentials, we have to include a second header in the server called **Access-Control-Allow-Credentials** with the value **true**.

In Listing 11-27, only one modification was made from the previous example: we added the **crossOrigin** attribute to the **img** element to declare the image as a cross-origin resource. Now, we can run this code on our computer and use the image from a server without violating the same-origin policy (CORS headers were already added to the www.formasterminds.com server).

> **IMPORTANT:** The **crossOrigin** attribute has to be declared before the **src** attribute to set the source as cross-origin. Of course, the attribute may also be declared in the opening tag of the **<img>**, **<video>** and **<audio>** elements, as any other HTML attribute.

# Extracting Data

The **getImageData()** method previously studied returns an object that can be processed through its properties (**width**, **height**, and **data**) or used as is by the **putImageData()** method. The purpose of these methods is to provide access to the canvas' content and return that data to the same or another canvas after being processed. But sometimes this information may be required for other purposes, like assigning it as the source of an **<img>** element, sending it to a server, or storing it in a database. The Canvas API includes the following methods to get the canvas' content in a data format we can use to accomplish these tasks.

> **toDataURL(**type**)**—This method returns data in data:url format containing a representation of the canvas' content in the format specified by the **type** attribute and a resolution of 96dpi. If no type was declared, the data is returned in the PNG format. The possible values for the attribute are **image/jpeg** and **image/png**.

**toBlob(**function, type**)**—This method returns an object with a blob containing a representation of the canvas' content in the format specified by the **type** attribute and a resolution of 96dpi. The first attribute is the function in charge of processing the object. If no type was declared, the blob is returned in the PNG format. The possible values for the attribute are **image/jpeg** and **image/png**.

The following document adds a box next to the **<canvas>** element to show the image produced from its content.

***Listing 11-28:*** *Creating a document to show an image with the canvas' content*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Canvas API</title>
  <style>
    #canvasbox, #databox {
      float: left;
      width: 400px;
      height: 300px;
      padding: 10px;
      margin: 10px;
      border: 1px solid;
    }
    .clearfloat {
      clear: both;
    }
  </style>
  <script src="canvas.js"></script>
</head>
<body>
  <section id="canvasbox">
    <canvas id="canvas" width="400" height="300"></canvas>
  </section>
  <section id="databox"></section>
```

```
  <div class="clearfloat"></div>
</body>
</html>
```

The code for this example draws the image on the canvas, extracts the content, and generates an **<img>** element to show it on the screen.

***Listing 11-29:*** *Creating an image with the canvas' content*

```
function initiate() {
  var element = document.getElementById("canvas");
  var width = element.width;
  var height = element.height;
  element.addEventListener("click", copyImage);

  var canvas = element.getContext("2d");
  canvas.beginPath();
  canvas.arc(width / 2, height / 2, 150, 0, Math.PI * 2, false);
  canvas.clip();

  var image = document.createElement("img");
  image.src = "snow.jpg";
  image.addEventListener("load", function() {
    canvas.drawImage(image, 0, 0, width, height);
  });
}
function copyImage() {
  var element = document.getElementById("canvas");
  var data = element.toDataURL();

  var databox = document.getElementById("databox");
  databox.innerHTML = '<img src="' + data + '">';
}
window.addEventListener("load", initiate);
```

The code in Listing 11-29 creates a circular clipping area and then draws an image on the canvas. The effect makes it seem like the image was cropped in a circular shape. When the user clicks on the canvas, we extract

this image from the canvas with the **toDataURL()** method and use the data returned by the method as the source of a new **<img>** element. The element processes the data and shows the image on the screen.

***Figure 11-20:*** *Image created from the canvas' content*

**Do It Yourself:** Create a new HTML file with the document of Listing 11-28 and a JavaScript file called canvas.js with the code in Listing 11-29. Upload the files, including the snow.jpg file, to your server or local server and open the document in your browser. Click on the canvas to extract the content and show the image inside the box on the right.

**The Basics:** Data:url and blobs are two different formats for data distribution. The data:url format is officially called *Data URI Scheme*. This is a string of text representing the data necessary to recreate the resource and therefore it may be used as the source of HTML elements, as demonstrated by the example of Listing 11-29. Blobs, on the other hand, are blocks of raw data. We will study blobs in Chapter 16.

# 11.4 Animations

There is no method to help us animate things on the canvas, and there is no predetermined procedure to do it. We have to erase the area of the canvas we want to animate, draw the shapes on it, and repeat the process again and again. Once the shapes are drawn, they cannot be moved, and we are only able to construct an animation by erasing the area and drawing the shapes again in a different position. For this reason, in games or applications that require a large number of objects to be animated, it is better to use images instead of shapes built by complex paths (for example, games usually use PNG images).

# Simple Animations

There are many techniques to create animations; some are simple and others more complex, depending on the application. In the following example, we are going to implement a basic animation technique using the **clearRect()** method introduced before to clear the canvas and draw the shapes again (the code assumes that we are using the document of Listing 11-1 with a single **<canvas>** element of 500 pixels by 300 pixels).

***Listing 11-30:*** *Creating an animation*

```
var canvas;
function initiate() {
  var element = document.getElementById("canvas");
  canvas = element.getContext("2d");
  window.addEventListener("mousemove", animation);
}
function animation(event) {
  canvas.clearRect(0, 0, 500, 300);

  var xmouse = event.clientX;
  var ymouse = event.clientY;
  var xcenter = 220;
  var ycenter = 150;
  var ang = Math.atan2(xmouse - xcenter, ymouse - ycenter);
```

```
  var x = xcenter + Math.round(Math.sin(ang) * 10);
  var y = ycenter + Math.round(Math.cos(ang) * 10);

  canvas.beginPath();
  canvas.arc(xcenter, ycenter, 20, 0, Math.PI * 2, false);
  canvas.moveTo(xcenter + 70, 150);
  canvas.arc(xcenter + 50, 150, 20, 0, Math.PI * 2, false);
  canvas.stroke();

  canvas.beginPath();
  canvas.moveTo(x + 10, y);
  canvas.arc(x, y, 10, 0, Math.PI * 2, false);
  canvas.moveTo(x + 60, y);
  canvas.arc(x + 50, y, 10, 0, Math.PI * 2, false);
  canvas.fill();
}
window.addEventListener("load", initiate);
```

This animation is about a pair of eyes that look at the pointer of the mouse all the time. To move the eyes, we have to update their position every time the mouse is moved. This is accomplished by detecting the position of the mouse on the window with the **mousemove** event. Every time the event is fired, the **animation()** function is called. This function clears the canvas with the instruction **clearRect(0, 0, 500, 300)** and then initializes the variables. The position of the mouse is captured by the properties **clientX** and **clientY** and the coordinates for the first eye are stored in the variables **xcenter** and **ycenter**.

After the initial values are ready, it is time to do the math. Using the values of the position of the mouse and the center of the left eye, we calculate the angle of the invisible line from one point to another using the **atan2()** method of the **Math** object (see Chapter 6). This angle is used to calculate the point at the center of the iris with the formula **xcenter + Math.round(Math.sin(ang) × 10)**. The number **10** in the formula represents the distance from the center of the eye to the center of the iris (because the iris is not at the center of the eye, it is close to the edge).

With these values, we can start drawing the eyes on the canvas. The first path is two circles representing the eyes. The **arc()** method for the left eye is

positioned at the values of **xcenter** and **ycenter**, and the circle for the right eye is generated 50 pixels to the right using the instruction **arc(xcenter + 50, 150, 20, 0, Math.PI * 2, false)**.

The animation is created by the second path. This path uses the **x** and **y** variables with the position previously calculated from the angle. The irises are drawn as solid black circles using the **fill()** method.

The process is repeated and the values recalculated every time the **mousemove** event is fired (every time the user moves the mouse).

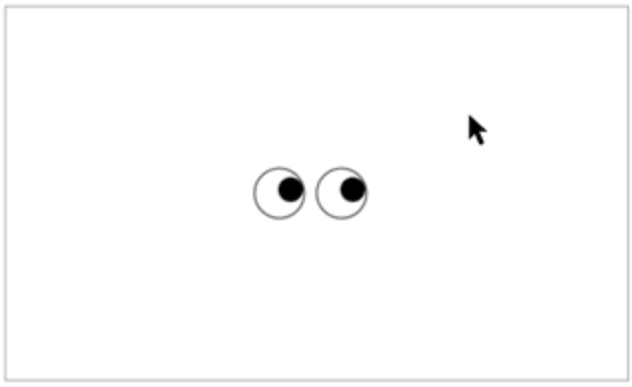

***Figure 11-21:*** *Simple Animation*

**Do It Yourself:** Copy the code in Listing 11-30 into the JavaScript file called canvas.js and open the document of Listing 11-1 in your browser. Move the mouse around the circles. You should see the eyes following the pointer.

**IMPORTANT:** In this example, the distance was calculated without taking into account the position of the canvas on the screen. This is because the **<canvas>** element in the document of Listing 11-1 was created at the top-left corner of the page and therefore the canvas' origin is the same as the document's origin. For more information on the **clientX** and **clientY** properties, see Chapter 6.

# Professional Animations

The loop for the animation in the previous example was generated by the **mousemove** event. In a professional animation, loops are controlled by the code and running all the time, independently of the user's activity. In

Chapter 6, we introduced two methods provided by JavaScript to run a loop: **setInterval()** and **setTimeout()**. These methods execute a function after a certain period of time. Working with these methods, we can produce decent animations, but they are not synchronized with the browser, causing glitches that are not tolerated in a professional application. In order to solve this problem, browsers include a little API with only two methods, one to generate a new cycle of the loop and another to cancel it.

**requestAnimationFrame(**function**)**—This method tells the browser that the function between parentheses should be executed. The browser calls the function when it is ready to refresh the window, synchronizing the animation with the browser window and the computer's display. We can assign this method to a variable and then cancel the process using the **cancelAnimationFrame()** method with the name of the variable between parentheses.

The **requestAnimationFrame()** method works like the **setTimeout()** method, we have to call it again in every cycle of the loop. The implementation is simple, but the code for a professional animation requires a certain organization that can only be achieved with advanced programming patterns. For our example, we are going to use a global object and distribute the tasks among several methods. The following is the document with the **<canvas>** element that we need to present the animation on the screen.

***Listing 11-31:*** *Creating the document to show a professional animation*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Animations</title>
  <style>
    #canvasbox {
      width: 600px;
      margin: 100px auto;
    }
    #canvas {
```

```
    border: 1px solid #999999;
   }
  </style>
  <script src="canvas.js"></script>
</head>
<body>
  <div id="canvasbox">
   <canvas id="canvas" width="600" height="400"></canvas>
  </div>
</body>
</html>
```

The CSS styles in the document of Listing 11-31 center the canvas on the screen and provide a border to identify its area. The document also loads a JavaScript file for the following code.

***Listing 11-32:*** *Creating a 2D video game*

```
var mygame = {
  canvas: {
    ctx: "",
    offsetx: 0,
    offsety: 0
  },
  ship: {
    x: 300,
    y: 200,
    movex: 0,
    movey: 0,
    speed: 1
  },
  initiate: function() {
    var element = document.getElementById("canvas");
    mygame.canvas.ctx = element.getContext("2d");
    mygame.canvas.offsetx = element.offsetLeft;
    mygame.canvas.offsety = element.offsetTop;
    document.addEventListener("click", function(event) {
      mygame.control(event);
```

```
    });
    mygame.loop();
  },
  loop: function() {
    if (mygame.ship.speed) {
      mygame.process();
      mygame.detect();
      mygame.draw();
      requestAnimationFrame(function() {
        mygame.loop();
      });
    } else {
      mygame.canvas.ctx.font = "bold 36px verdana, sans-serif";
      mygame.canvas.ctx.fillText("GAME OVER", 182, 210);
    }
  },
  control: function(event) {
    var distancex = event.clientX - (mygame.canvas.offsetx +
mygame.ship.x);
    var distancey = event.clientY - (mygame.canvas.offsety +
mygame.ship.y);
    var ang = Math.atan2(distancex, distancey);
    mygame.ship.movex = Math.sin(ang);
    mygame.ship.movey = Math.cos(ang);
    mygame.ship.speed += 1;
  },
  draw: function() {
    mygame.canvas.ctx.clearRect(0, 0, 600, 400);
    mygame.canvas.ctx.beginPath();
    mygame.canvas.ctx.arc(mygame.ship.x, mygame.ship.y, 20, 0, Math.PI /
180 * 360, false);
    mygame.canvas.ctx.fill();
  },
  process: function() {
    mygame.ship.x += mygame.ship.movex * mygame.ship.speed;
    mygame.ship.y += mygame.ship.movey * mygame.ship.speed;
  },
```

```
  detect: function() {
    if (mygame.ship.x < 0 || mygame.ship.x > 600 || mygame.ship.y < 0 ||
mygame.ship.y > 400) {
      mygame.ship.speed = 0;
    }
  }
};
window.addEventListener("load", function() {
  mygame.initiate();
});
```

A professional application should avoid global variables and functions and concentrate the code inside a unique global object. In Listing 11-32, we call this object **mygame**. All the properties and methods necessary to create a small video game are declared inside this object.

Our game is about a black spaceship that moves in the direction indicated by the click of the mouse. The goal is to change the direction of the ship to avoid the walls. Every time the direction is changed, the speed of the ship increases, making it more and more difficult to keep the ship inside the canvas.

The organization required to create this kind of application always includes some standard elements. We have to declare initial values, control the animation loop, and distribute the rest of the tasks in multiple methods. In the example of Listing 11-32, this organization is achieved with the inclusion of a total of six methods: **initiate()**, **loop()**, **control()**, **draw()**, **process()**, and **detect()**.

We start by declaring the **canvas** and **ship** properties. These properties contain objects with basic information for the game. The **canvas** object has three properties: **ctx** to store the context of the canvas, and **offsetx** and **offsety** to store the position of the **<canvas>** element related to the page. The **ship** object, on the other hand, has five properties: **x** and **y** to store the coordinates of the ship, **movex** and **movey** to determine the direction, and **speed** to know the ship's current speed. These are important properties required in almost every section of the code, but some of their values have to be initialized. As we did in previous examples, this work is performed by the **initiate()** method. This method is called by the **load** event when the

document has finished loading, and it is responsible for assigning to the properties all the values required to initiate the application.

The first task of the **initiate()** method is to get a reference to the context of the canvas and the position of the element in the window using the **offsetLeft** and **offsetTop** properties. Then, a listener is added to the **click** event to respond when the user clicks anywhere on the document.

The **loop()** method is the second most important method in the underlying organization of a professional application. This method calls itself again and again while the application is running, creating a loop that goes through every part of the process. In our example, this process is divided into three methods: **process()**, **detect()** and **draw()**. The **process()** method calculates the new position of the ship according to the current direction and speed, the **detect()** method compares the coordinates of the ship with the boundaries of the canvas to determine if the ship has crashed into the walls, and the **draw()** method draws the ship on the canvas. The loop executes these methods one by one and then calls itself with the **requestAnimationFrame()** method, starting a new cycle.

The last step to finish the basic structure of our application is to check the user's feedback. In our game, this is done by the **control()** method. When the user clicks on any part of the document, this method is called to calculate the direction of the ship. The formula is similar to previous examples; we calculate the distance from the ship to the place where the event occurred, then we get the angle of the invisible line between those two points, and finally, the direction in coordinates obtained by the **sin()** and **cos()** methods is stored in the **movex** and **movey** properties. The last instruction in the **control()** method increases the speed of the ship to make our game more interesting.

The game starts as soon as the document is loaded and finishes when the ship crashes into a wall. To make this small application look more like a video game, we added an **if** instruction in the loop to check the condition of the ship. This condition is determined by the value of the speed. When the **detect()** method determines that the coordinates of the ship are out of the boundaries of the canvas, the speed is reduced to **0**. This value turns the condition of the loop false and the message "GAME OVER" is shown on the screen.


***Figure 11-22:*** *Simple Video Game*

# 11.5 Video

As with animations, there is no particular method to show video on a **<canvas>** element. The only way to do it is by taking every frame from the **<video>** element and drawing it as an image on the canvas using the **drawImage()** method. The following document includes a short JavaScript code to turn the canvas into a mirror.

***Listing 11-33:*** *Displaying video on canvas*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Video on Canvas</title>
  <style>
    section {
      float: left;
    }
  </style>
  <script>
    var canvas, video;
    function initiate() {
      var element = document.getElementById("canvas");
      canvas = element.getContext("2d");
      video = document.getElementById("media");

      canvas.translate(483, 0);
      canvas.scale(-1, 1);
      setInterval(processFrames, 33);
    }
    function processFrames() {
      canvas.drawImage(video, 0, 0);
    }
    window.addEventListener("load", initiate);
  </script>
</head>
```

```
<body>
  <section>
    <video id="media" width="483" height="272" autoplay>
      <source src="trailer.mp4">
      <source src="trailer.ogg">
    </video>
  </section>
  <section>
    <canvas id="canvas" width="483" height="272"></canvas>
  </section>
</body>
</html>
```

As explained before, the **drawImage()** method can take three types of sources: an image, a video, or another canvas. Because of this, to show video on the canvas, we have to provide a reference to the **<video>** element as the first attribute of the method. Although this is enough to draw the video on the canvas, we have to consider that videos are composed of multiple frames, and the **drawImage()** method is only able to draw one frame at a time. For this reason, to show the entire video, we have to repeat the process for every frame. In the code in Listing 11-33, a **setInterval()** method is used to call the **processFrames()** function every 33 milliseconds. This function executes the **drawImage()** method with the video as its source (The time set by **setInterval()** is approximately the time it takes every frame to be shown on the screen).

When we draw an image on the canvas, it is bound to the canvas' properties and can be processed like any other content. In our example, the image is inverted to create a mirror effect. The effect is produced by the application of the **scale()** method to the context (everything drawn on the canvas is inverted).

***Figure 11-23:*** *Mirror image of a video*
*© Copyright 2008, Blender Foundation / www.bigbuckbunny.org*

# Real-Life Application

There are a million things we can do with the canvas, but it is always fascinating to see how far we can go combining simple methods from different APIs. The following example describes how to take a snapshot with the webcam and show it on the screen using the **<canvas>** element.

***Listing 11-34:*** *Programming a snapshot application*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Snapshot Application</title>
  <style>
    section {
      float: left;
      width: 320px;
      height: 240px;
      border: 1px solid #000000;
    }
  </style>
  <script>
    var video, canvas;
    function initiate() {
      var promise = navigator.mediaDevices.getUserMedia({video: true});
      promise.then(success);
      promise.catch(showerror);
    }
    function success(stream) {
      var element = document.getElementById("canvas");
      canvas = element.getContext("2d");
      var player = document.getElementById("player");
      player.addEventListener("click", snapshot);
```

```
    video = document.getElementById("media");
    video.srcObject = stream;
    video.play();
  }
  function showerror(event) {
    console.log("Error: " + event.name);
  }
  function snapshot() {
    canvas.drawImage(video, 0, 0, 320, 240);
  }
  window.addEventListener("load", initiate);
 </script>
</head>
<body>
 <section id="player">
  <video id="media" width="320" height="240"></video>
 </section>
 <section>
  <canvas id="canvas" width="320" height="240"></canvas>
 </section>
</body>
</html>
```

This document provides some CSS styles, the JavaScript code, and a few HTML elements, including the **<video>** and **<canvas>** elements. The **<canvas>** is declared as usual, but we did not specify the source for the **<video>** element because we are going to use it to show the video from the webcam.

The code is simple. Our standard **initiate()** function calls the **getUserMedia()** method as soon as the document is loaded to access to the webcam. In case of success, the **success()** function is called. Most of the work is done by this function. It gets the context of the canvas, adds a listener for the **click** event to the box corresponding to the video, and assigns the video from the webcam to the **<video>** element. In the end, the video is played using the **play()** method.

At this point, we have the video from the webcam displayed in the box on the left. To take a snapshot of that video and put it in the right box, we created the **snapshot()** function. This function responds to the **click** event. When the user clicks on the video, the function executes the **drawImage()** method with a reference to the video and the values corresponding to the size of the **<canvas>** element. The method takes the current frame and draws it on the canvas to show the snapshot on the screen.

**Do It Yourself:** Create a new HTML file with the document of Listing 11-34. Open the document in your browser and authorize the browser to access the camera. Click on the video. The image should be drawn on the canvas. Remember to upload the file to your server before testing.