As the Web matures, it becomes clear that Web pages are not simply documents, but a programming framework. HTML5 reinforces that idea to a great extent. Many of the new features aren’t really part of HTML at all, but enhancements in programming features added to Web browsers. Virtually all Web browsers include the JavaScript programming language, and the language has some important new features as part of HTML5.

This part is not a complete introduction to computer programming in JavaScript. Programming is a complex business, and learning how to program in JavaScript deserves its own book; see my JavaScript and AJAX For Dummies (Wiley) or HTML, XHTML, and CSS All-In-One For Dummies, 2nd edition (Wiley) books for a more complete treatment.

This part gives you enough of a review (or preview) of JavaScript that you’ll be able to follow JavaScript as I use it throughout the book. If you are already comfortable in any other programming language, you should be able to pick up JavaScript with the overview presented here. If programming is new to you, this introduction should give you enough foundation that you’ll be able to read about the new JavaScript features in HTML5.

The JavaScript described in this part is suitable for current browsers. Part 7 discusses the newer JavaScript ideas associated with HTML5.

Be sure to check out my Web site for working examples of every code fragment in the book: www.aharrisbooks.net/h5qr.

In this part . . .

✓ Attaching JavaScript to Your Page
✓ Introducing Variables and Functions
✓ Working with Variables
✓ Intermingling with the Web Page
✓ Creating Choices with if
✓ Handling Repetition with for Loops
✓ Building while Loops
✓ Transmitting Data To and From Functions
✓ Utilizing Arrays to Simplify Data
Adding JavaScript to Your Page

JavaScript is a programming language embedded within the Web browser. It works very closely with HTML. In fact, HTML forms the user interface of most JavaScript programs. Where HTML provides the structure of a Web page and CSS provides the visual interface, JavaScript adds action. Here’s an extremely simple example:

```html
<!DOCTYPE HTML>
<html lang = "en">
<head>
    <title>alert.html</title>
    <meta charset = "UTF-8" />
</head>

<body>
    <h1>Add quick JavaScript to a page</h1>
    <p>
        <button onclick = "alert('Hi there!')">
            click me
        </button>
    </p>
</body>
</html>
```

This looks a lot like an HTML page because that’s mainly what it is. Here’s what’s interesting about the page:

- **It has a button.** This button isn’t in a form (although often they are). Note that HTML buttons don’t do anything when you click them. You’re about to change that.

- **The button has an onclick attribute.** onclick is an event attribute. It’s a special attribute of form elements that allows you to specify what should happen when an event occurs. The onclick attribute signifies what should happen when the user clicks a button.

- **The value of the onclick attribute is JavaScript code.** You can use the onclick attribute to attach a single line of JavaScript code to your button press. In the preceding example, that line is alert('Hi there!').

- **The alert() command pops up a message box.** The alert command is used to send a message to the user without changing the structure of the page. Alert is easy to use, but it’s annoying for the user. See “Interacting with the Web Page” for details on a more elegant way to talk to the user.

- **Any text inside the alert’s parentheses will be displayed to the user.** In this case, the message box will contain the greeting "Hi there!".
Adding JavaScript to Your Page — Becoming Familiar with Variables and Functions

✓ Note that I used single quotes. JavaScript and HTML both allow single and double quotes. Since this JavaScript code was embedded in an HTML statement, I used a combination of single and double quotes to keep the browser from being confused about which quote was which.

The result of this program is shown in Figure BP1-1.

Becoming Familiar with Variables and Functions

Most of the interesting stuff done by computer programs involves data, which is stored in computer memory as variables. A variable is simply a name for a section of memory that will hold data. Any interesting program is likely to get data from someplace, do something with that data, and send data back out. As an example, look at the program in Figures BP1-2 and BP1-3.
This program is exciting because, as simple as it is, it illustrates all kinds of important ideas:

✓ **A single button press leads to a whole flurry of activity.** The onclick event calls only a single line of JavaScript code, but a lot of stuff is happening. Somehow one line is doing more than one thing.

✓ **A new type of dialog is asking for input.** You can use a special dialog to get input from the user.

✓ **When the user types his name, it has to go somewhere.** You need someplace in memory to store the name.

✓ **It repeats the name.** The program produces a dialog box with the name in it.

✓ **Somehow the name is integrated into some other output.** It’s easy to see how you could print out an ordinary message, but this one has a name somehow stuck in there.

There is a lot of magic going on here, but it’s really not that hard to figure out. First, look over the entire code listing, and then I explain what’s going on in the following sections:

```html
<!DOCTYPE HTML>
<html lang = "en">
<head>
    <title>functionIO</title>
    <meta charset = "UTF-8" />

    <script type = "text/javascript">
        //<![CDATA[
        function greet(){
            var userName = prompt("What is your name?");
            alert("Hi there, " + userName);
        } // end greet

    //]]></script>

</head>

<body>

</body>
```
<h1>Functions, Variables, and Dialog-based I/O</h1>

```html
<form>
  <fieldset>
    <button type = "button"
      onclick = "greet()">click me
    </button>
  </fieldset>
</form>
</body>
</html>

Building the HTML framework

Normally, the best way to get your head around JavaScript code is to start with the HTML, which provides the structure and framework, and often provides cues about what the JavaScript will do. I usually create the HTML part of a page first and then add JavaScript to breathe life into it. Here’s what’s interesting about the HTML:

- **It’s basically a form.** The `form` construct was designed for server-side processing in languages like PHP. You don’t technically need the `form` tag itself for the client-side programming done in this book, but most JavaScript code uses form elements. Normally, I go ahead and include the `form` tags to provide structure for CSS formatting.

- **It includes a button.** Nothing will happen in this program until the user clicks a button, so we need to add a button. You can either use the `input type = "button"> element or the newer `button type = "button">` tag. I prefer the `button` tag because frequently I need to give buttons a different style than standard input elements. Of course, Part 6 also explains how to pick input elements that are of type “button” if you prefer that approach.

- **The onclick event is set to `greet()`**. The basic rule is you can add one line of JavaScript code to each event. In this case, that one line calls a special thing called a function, which is any number of code lines combined with a single name. I call the function `greet()` and build it in the next section.

Constructing a function with variables

As you’ve seen, programmers like to separate various parts of their work. In the HTML code, it’s important to specify that the user will be greeted when he clicks the button. But it makes sense to move the actual definition of what it means to greet the user to another part of the page. This is done with the `<script>` tag, normally placed in the page’s header area. Here’s how it works:
1. Create a script tag in the header. Normally, the <script> tag also includes a type specifier: <script type = "text/javascript">.

2. Indicate that your script is not HTML code. If you are running your page through a validator (and you should be doing this), the validation program may get confused when it sees JavaScript code. JavaScript is a different language from CSS and HTML, and it has different rules. Use //<![CDATA[ at the beginning of your script element and //]]> at the end to inform validators that they can ignore the content of the script tag. While this step is not absolutely necessary, I use it in most examples in this book. If you leave the CDATA indicator out, the program will still work fine, but the validator may indicate false errors because it is mistakenly interpreting JavaScript as HTML.

3. Create a function called greet(). The page has already indicated that a batch of code called greet() should run as soon as the user clicks the button. In the script area, define what it means to greet. Use the function keyword to indicate you are building a function. Function names are always followed by parentheses. For now, leave the parentheses empty; see “Sending Data To and From Functions” for an example of how to use them.

4. Create a variable called username. A variable is a name for information in memory. When you create a variable, you are setting aside a piece of memory that will contain data. In JavaScript, you can use the var statement to indicate that you’re building a variable. See “Dealing with different data types” for more information on variable types, especially if you’re familiar with a more traditional programming language like C++ or Java.

5. Use the prompt statement to ask for user input. The prompt() command is much like the alert() statement. It takes text input and creates a pop-up dialog box. However, the prompt box is a bit different because it allows the user to type information into the box. When the user dismisses the dialog box, the prompt statement returns the text typed by the user.

6. Assign the results of the prompt to the variable. The single equals sign should be read as “gets,” so the statement var userName = prompt ("What is your name?") should be read as “Variable userName gets prompt of ‘What is your name?’”. By the end of this statement, whatever the user typed in the prompt dialog box is stored in the variable userName.

7. Greet the user with an alert() dialog. The alert() dialog will be used as output. The alert statement begins with standard quoted text.

8. Integrate the variable with text. Note that the prompt contains the value "Hi there " + userName + "!". You can probably guess what this does. The "Hi there " and "!" elements are normal text, and they are placed in quotes. You don’t really want to see the text “userName” on the screen, but the value of the userName variable. Use the plus (+) sign to
combine text and variables. The fancy name for combining text and variables is *concatenation*. Gotta love those computer scientists and their complicated words for easy ideas.

**Managing the pesky details**

There are a number of other details going on in this example that are important. First, note the special punctuation. Just like sentences end in punctuation marks like periods and question marks, most programming languages also have special punctuation. If you’re familiar with CSS, you’ve already got a head start on JavaScript formatting because both borrow style from the C programming language. Here’s the skinny:

- **Use squiggly braces** ({} to indicate a section of code. A function contains several lines of code that are meant to perform one task. All code inside the function goes between a left brace ( { ) and a right brace ( } ).

- **Indent all code inside braces.** Soon enough, your programs will get complicated, and you’ll have code with many levels of nested sections. Indent all the code between braces, just as you do with HTML code.

- **End most lines with a semicolon.** In general, any line of code that doesn’t end with a brace ends with a semicolon (;). Again, this is an idea you’ve probably gotten familiar with in CSS.

- **Capitalization and spelling matters.** HTML can be somewhat forgiving, but programming languages (in general) are not. If you get the capitalization or spelling wrong, the whole thing is likely to break.

- **Include comments to indicate what’s going on.** The // character at the beginning of a line indicates a comment. It’s useful to add comments to explain what’s happening. I use a comment after each right brace ( } ) to indicate what type of structure I’m ending.

---

**Employing Variables**

Variables are a really important part of computer programming. While JavaScript makes variables pretty easy to use, there are still some very important things to keep in mind when you use a variable.

- **The var statement is preferred but not required.** It’s generally considered best to use the var statement to define a variable.

- **Just mentioning a variable creates it.** If you leave out the var statement, JavaScript will try to create a variable for you. For example, `userName = "Andy";` is similar to `var userName = "Andy";`. 

The variable type is automatically determined by JavaScript. Different kinds of information are stored differently in the computer’s memory. Integers (counting numbers), real numbers (with a decimal point), and text are all stored differently in memory. JavaScript tries to automatically create the right type of variable based on the context.

Dealing with different data types

The point about variable types is really important. Different kinds of data are stored in different ways in the computer. Some languages (like C++ and Java) are extremely picky about variable types and require you to think carefully about what type of data goes into what variable. JavaScript is much easier-going about data types, but it still must figure out how to store the data. When you assign a value to a JavaScript variable, JavaScript will turn it into one of these main types:

- **Integers**: Integers are the counting numbers, zero, and negative numbers. Integers do not have a decimal point. They are pretty easy for the computer to work with and rarely cause problems, so they are a favorite data type.

- **Floating point**: Numbers with a decimal point are often called floats. (Some languages have more than one floating type, but JavaScript just has floats.) Floating point data requires a lot more memory than integers, and can introduce some crazy errors, so JavaScript stores values with the float mechanism only if needed (that is, a numeric value has a decimal part).

- **Strings**: Text data is a special case. Text is really stored as a bunch of integers in contiguous memory cells. This reminded early programmers of beads on a string, so programmers never say “text” data, but call text “strings” instead. Almost all user input and output is done through strings.

- **Boolean**: Boolean value is another special case. A Boolean data element contains only the values true or false. Simple as they are, Booleans are extremely useful.

- **Objects**: JavaScript supports an advanced programming idea called object-oriented programming. A JavaScript object can be a very complex element containing variables and functions. Any element on the Web page can be converted into a JavaScript object, and JavaScript supports a rich framework of other types of objects as well.

Exploiting a computer program to do bad math

While JavaScript tries to shield you from worrying about data types, you still have to think about this issue because sometimes it will cause you problems. For example, consider the page shown in Figure BP1-4.
This program asks the user for two numbers and then tries to add them together. However, it doesn’t do it correctly. If the user inputs 3 and 5, the result is 35. The Add Wrong button calls the (cleverly named) addWrong() function. Here’s the code for addWrong():

```javascript
function addWrong(){
    // input two numbers
    var x = prompt("X");
    var y = prompt("Y");
    var sum = x + y;
    alert(x + " + " + y + " = " + sum);
} // end addWrong
```

The code for addWrong() looks perfectly reasonable. Here’s what it does:

1. Ask the user for x and y. Use the prompt statement to ask the user for two numbers. Store these two numbers in the variables x and y.
2. Add x and y and put the result in sum. Make a new variable called sum that will contain the sum of x and y.
3. Output the value of sum. Use the standard alert() statement to output the result.

This code seems completely straightforward, and it ought to work. However, it doesn’t do what you want. It reports that 3 + 5 is 35 — and computers are supposed to be good at math.

### Managing data types correctly

The key to fixing the addWrong problem is to understand how the computer is misinterpreting the data. Here’s the underlying problem: The prompt() command is asking for text from the user. That text is stored in a string variable because JavaScript assumes any input from the user is a string. So the value 3 isn’t stored as the number 3, but as a text variable with the value ‘3’. (The quotes are important because text values are always encased in quotes and numeric values are not.) The plus sign combines two string values, so if x is ‘3’ and y is ‘5’, x + y means “concatenate ‘3’ and ‘5’, giving ‘35’.” That is not what we want at all.
The way to fix this is to tell the computer that $x$ and $y$ should be interpreted as integers, like this:

```javascript
function addRight(){
    //input two numbers
    var x = prompt("X");
    var y = prompt("Y");

    //force values to integer format
    x = parseInt(x);
    y = parseInt(y);

    var sum = x + y;
    alert(x + " + " + y + " = " + sum);
}
```

This code is very similar to the `addWrong()` function, but it adds a new section. The `parseInt()` function accepts a string and converts it to an integer. If it cannot convert the value, it returns the special value `NaN` (Not a Number).

There are similar functions for converting data to other types. Use `parseFloat()` to convert a string value to a floating point (decimal) value.

This version of the code still pulls in the values as strings, but it converts them to integer values before doing any calculation. The exact same code is used to add the values ($sum = x + y$), but this time, the values are integers — so the computer knows that the plus sign really means “add.”

Figure BP1-5 illustrates the `addRight` function working as expected.

Experienced programmers (especially Java programmers) might be horrified at the cavalier way JavaScript lets you just create and change variable types on the fly. However, it really works pretty well most of the time, so you might just have to relax and appreciate that different languages have different goals. JavaScript tries to do as much automatically as it can, which is nice for beginners. Java (which is an entirely different language) is more focused on protecting you from various kinds of mistakes often brought on by sloppy coding. Java has much stricter rules, but when you follow those rules, you tend to make fewer mistakes.
Interacting with the Web Page

Of course, the main purpose of JavaScript is to manipulate the Web page. The `alert()` and `prompt()` commands are easy, but they’re pretty awkward to use. Most of the time, you’ll use JavaScript to get information from a Web form and use it to manipulate the page directly. As an example, look at Figure BP1-6.

![Figure BP1-6](image)

This program is very similar to the `functionIO` program described earlier in this part (see “Becoming Familiar with Variables and Functions”), but it uses the Web site itself for input and output. As usual, take a look at the overall code listing first to see what’s going on, and then I explain all in the sections that follow.

```html
<!DOCTYPE HTML>
<html lang = "en">
<head>
    <title>hiUser.html</title>
    <meta charset = "UTF-8" />
    <script type = "text/javascript">
        //<![CDATA[
        function sayHi(){
            //get form elements
            lblOutput = document.getElementById("lblOutput");
            txtInput = document.getElementById("txtInput");
```

```javascript
```
//get the username from the text box
userName = txtInput.value;

//prepare greeting
greeting = "Hi there, " + userName + ";

//send greeting to label
lblOutput.innerHTML = greeting;
} // end function
//]]>
</script>
<style type = "text/css">
label, button, input {
    display: block;
    text-align: center;
    width: 100%;
}

fieldset {
    width: 60%;
    margin: auto;
}
</style>
</head>

<body>
<h1>Hi User</h1>
<p>Demonstrates basic JavaScript I/O</p>
<form>
<fieldset>
    <label id = "lblOutput">
    Type your name and press the button
    </label>
    <input type = "text"
    id = "txtInput" />
    <button type = "button"
        onclick = "sayHi()">
        Click me
    </button>
</fieldset>
</form>
</body>
</html>
Setting up the HTML form

This page is HTML that is modified with JavaScript. Of course, the HTML needs to be set up in a way that lends itself to modification. To set up the page, follow these steps:

1. Build a form. Although output can be anywhere on a page, you'll generally get input from Web forms.
2. Create a text field for input. The input will come from a text field. If you're going to refer to a page element with JavaScript, you need to give it an id. You probably already do this if you're using CSS. My text field is called txtInput.
3. Create a label for output. You can use any HTML element you want for output, but I'm using a label inside the form. Be sure to give the element an id attribute so you can refer to it in code.
4. Build a button to trigger the process. You'll need a button to get everything started. The button doesn't need an id attribute, but you will need to add an onclick attribute to attach a function to the button.
5. Add CSS to beautify. The HTML itself might be a little ugly, so you'll probably want to add CSS to make things look better.

Writing the sayHi() function

The main feature of this page is the sayHi() function. This JavaScript code is called by the button. It does a pretty straightforward series of tasks.

1. Create a variable for the input element. If you want to work with elements from the Web page, you need to create JavaScript variables for them. The document.getElementById() command takes the id attribute of a page element and creates a JavaScript object referring to that element on the page. For this project, I need a variable for txtInput because I will get the user's name from this element.
2. Build a variable for any elements you'll be modifying. I will change the text of the element called lblOutput, so I use the document.getElementById() command to make a variable for that element as well. In general, you'll make a variable for any element you want to extract information from or change through code.
3. Get the value attribute from txtInput. The txtInput is a JavaScript variable associated with the txtInput text box on the Web page. The value of the text field is stored in a property called txtInput.value. You can read this value to discover what is in the text element, or change the value to modify what the text element displays. In this example, I retrieve the value from txtInput.value and put it in a new variable called userName.
4. Build a greeting string. The greeting is a combination of straight text and a variable, so I create it in a separate step. Although it's possible to combine many steps into one line, it's often better to break complex operations into many smaller lines. This way, if the code crashes on a particular line, you'll have a good idea what went wrong.

5. Copy the greeting to the label. The best way to present output to HTML pages is with the `innerHTML` technique. Every HTML element with a beginning and end tag has an `innerHTML` property. You can read this property to find out what text is in the element, and you can set the property to change the text. In this case, I copy the greeting to `lblOutput`'s `innerHTML` property. When this code is complete, `lblOutput` will display the greeting.

Making Choices with if

Sometimes you'll need your code to make decisions. For example, if somebody famous typed his name in your Web site, you might want to create a custom greeting for him. (I know this is a goofy example, but stay with me.) Take a look at the `ifElse.html` site in Figures BP1-7 and BP1-8.
As you can see, the program looks at the input in the text box and changes behavior based on the value of the text field. The code is quite similar to the code in the hiUser page. The only difference is the way the function is written. Here's the checkName() function called in ifElse.html:

```javascript
function checkName()
    // from ifElse.html
    lblOutput = document.getElementById("lblOutput");
    txtInput = document.getElementById("txtInput");

    userName = txtInput.value;
    if (userName == "Tim Berners-Lee"){
        lblOutput.innerHTML = "Thanks for inventing HTML!";
    } else {
        lblOutput.innerHTML = "Do I know you?";
    } // end if
} // end function
```

**Altering the greeting with if**

This code uses an important idea called a condition inside a construct called an if statement. Here's what's happening:
1. Set up the Web page as usual. The HTML code has elements called `lblOutput` and `txtInput`. It also has a button that calls `checkName()` when it is clicked.

2. Create variables for important page elements. You're getting data from `txtInput` and changing the HTML code in `lblOutput`, so create variables for these two elements.

3. Get `userName` from `txtInput`. Use the `txtInput.value` trick to get the value of the input element called `txtInput` and place it in the variable `userName`.

4. Set up a condition. The key to this program is a special element called a condition (an expression that can be evaluated as true or false). Conditions are often (as in this case) comparisons. Note that the double equals sign (**) is used to represent equality. In this example, I'm asking whether the `userName` variable equals the value "Tim Berners-Lee".

5. Place the condition in an `if` structure. The `if` statement is one of a number of programming constructs that use conditions. It contains the keyword `if` followed by a condition (in parentheses). If the condition is true, all of the code in the following set of braces is executed.

6. Write code to execute if the condition is true. Create a set of squiggly braces after the condition. Any code inside these braces will execute if the condition is true. Be sure to indent your code, and use the right squiggle brace (}) to end the block of code. In this example, I give a special greeting to Tim Berners-Lee (because he is just that awesome).

7. Build an `else` clause. You can build an `if` statement with a single code block, but often you want the code to do something else if the condition is false. Use the `else` construct to indicate that you will have a second code block that will execute only if the condition is false.

8. Write the code to happen when the condition is false. The code block following the `else` clause will execute only if the condition is false. In this particular example, I have a greeting for everyone except Berners-Lee.

Familiarizing yourself with the different flavors of `if`

If statements are extremely powerful, and there are a number of variations. You can actually have one, two, or any number of branches. You can write code like this:

```javascript
if (userName == "Tim Berners-Lee"){
    lblOutput.innerHTML = "Thanks for inventing HTML"
}
```
With this structure, the greeting will occur if `userName` is "Tim Berners-Lee", and nothing will happen if `userName` is anything else. You can also use the if-else structure (this is the form used in the actual code) as follows:

```javascript
if (userName == "Tim Berners-Lee"){
    lblOutput.innerHTML = "Thanks for inventing HTML!";
} else {
    lblOutput.innerHTML = "Do I know you?";
} // end if
```

One more alternative lets you compare as many results as you want by adding new conditions:

```javascript
if (userName == "Tim Berners-Lee"){
    lblOutput.innerHTML = "Thanks for inventing HTML!";
} else if (userName == "Al Gore") {
    lblOutput.innerHTML = "Thanks for inventing the Internet";
} else if (userName == "Hakon Wium Lie") {
    lblOutput.innerHTML = "Thanks for inventing CSS";
} else {
    lblOutput.innerHTML = "Do I know you?";
} // end if
```

### Getting to know the conditional operators

The `==` operator checks to see if two values are identical, but JavaScript supports a number of other operators as well:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a == b</code></td>
<td><code>a</code> is equal to <code>b</code>.</td>
</tr>
<tr>
<td><code>a &lt; b</code></td>
<td><code>a</code> is less than <code>b</code>.</td>
</tr>
<tr>
<td><code>a &gt; b</code></td>
<td><code>a</code> is greater than <code>b</code>.</td>
</tr>
<tr>
<td><code>a &lt;= b</code></td>
<td><code>a</code> is less than or equal to <code>b</code>.</td>
</tr>
<tr>
<td><code>a &gt;= b</code></td>
<td><code>a</code> is greater than or equal to <code>b</code>.</td>
</tr>
<tr>
<td><code>a != b</code></td>
<td><code>a</code> is not equal to <code>b</code>.</td>
</tr>
</tbody>
</table>

If you're coming from another programming language like Java, C, or PHP, you might wonder how string comparisons work, because they require different operators in these languages. JavaScript uses exactly the same comparison operators for all types of data, so there's no need to learn different operators. Yeah JavaScript!
Managing Repetition with for Loops

Computers are well-known for repetitive behavior. It’s pretty easy to get a computer to do something many times. The main way to get this behavior is to use a mechanism called a loop. The for loop is a standard kind of loop that is used when you know how often something will happen. Figure BP1-9 shows the most basic form of the for loop.

![Figure BP1-9](image)

For loops

- count to ten
- count backwards
- count by fives

1
2
3
4
5
6
7
8
9
10

Arranging the Web page

The Web page in Figure BP1-9 is also used to demonstrate three different kinds of for loops. As usual, the HTML code sets everything up. Here’s the HTML code that creates the basic framework:

```html
<body onload = "init()">
  <h1>For loops</h1>
  <form>
    <fieldset>
      <!-- Other buttons -->
      <button type = "button"
              onclick = "count()"
              >
        count to ten
      </button>
    </fieldset>
  </form>
</body>
```
Managing Repetition with for Loops

```html
onclick = "back()"
    count backwards
</button>

<button type = "button"
    onclick = "byFive()"
    count by fives
</button>

</fieldset>
</form>

<div id = "output">Click a button to see some counting...
</div>
</body>
</html>

While the HTML is pretty straightforward, it does have some important features:

- **The body calls an initialization function.** Often, you’ll want some code to happen when the page first loads. One common way to do this is to attach a function call to the `onload` attribute of the `body` element. In this example, I call the `init()` function as soon as the body is finished loading. The contents of the `init()` function will be described in the next section.

- **The page is mostly an HTML form.** The most important part of this page is the form with three buttons on it. Each button calls a different JavaScript function.

- **A special div is created for output.** In previous examples, output was done in a label inside the form, but you can add output to any HTML element that has a beginning and ending tag. In this example, I create a div for output. It’s a good idea to put some default text in the div to ensure the div is actually changing when it’s supposed to.

From this example, it’s easy to see why it’s a good idea to write the HTML first. The HTML code gives me a solid base for the program, and it also provides a good outline of what JavaScript code I’ll need. Clearly, this page calls for four JavaScript functions, `init()`, `count()`, `back()`, and `byFive()`. The names of all the functions are pretty self-explanatory, so it’s pretty easy to see what each one is supposed to do. It’s also clear that the div named `output` is intended as an output area. When you design the HTML page well, the JavaScript code becomes very easy to start.

**Building global output**

This program illustrates a situation that frequently comes up in JavaScript programming: All three of the main functions will refer to the same output area. It
seems a waste to create a variable for output three different times. Instead, I make a single global output variable available to all functions and attach the variable to that element once when the page loads.

In order to understand why this is necessary, it’s important to discuss an idea called variable scope. Generally, variables are created inside functions. As long as the function is running, the variable still exists. However, when a function is done running, all the variables created inside that function are instantly destroyed. This prevents functions from accidentally changing the variables in other functions. Practically, it means you can think of each function as a separate program.

However, sometimes you want a variable to live in more than one function. The output variable in the forLoop.html page is a great example because all the functions will need it. One solution is to create the variable outside any functions. Then all the functions will have access to it.

You can create the output variable without being in a function, but you can’t attach it to the actual div in the Web page until the Web page has finished forming. The init() function is called when the body loads. Inside that function, I assign a value to the global output variable. Here’s how the main JavaScript and the init() method code look:

```javascript
var output;

function init(){
  output = document.getElementById("output");
} // end init
```

This code creates output as a global variable, and then attaches it to the output div after the page has finished loading.

Creating the basic for loop

The standard for loop counts the values between 1 and 10. The count to ten button triggers the count() function. Here’s the code for count():

```javascript
function count(){
  output.innerHTML = "";
  for (i = 1; i <= 10; i++){
    output.innerHTML += i + "<br ">
  } // end for loop
} // end count
```

Although the count() function clearly prints ten lines, it has only one line that modifies the output div. The main code repeats many times to create the long output. Here’s the steps to build a standard for loop:

1. You can use the output var immediately. Because output is a global variable and it has already been created, you can use it instantly. There’s no need to initialize it in the function.
2. Clear the output. Set `output.value` to the empty string (""") to clear the output. This will destroy whatever text is currently in the div.

3. Start a `for` loop. The `for` loop is a special loop used to repeat something a certain number of times. `for` loops have three components: initialization, comparison, and update.

4. Initialize your counting variable. A `for` loop works by changing the value of an integer many times. The first part of a `for` loop initializes this variable (often called `i`) to a starting value (usually zero or one).

5. Specify a condition for staying in the loop. The second part of a `for` statement is a condition. As long as the condition is true, the loop will continue. As soon as the condition is evaluated as false, the loop will exit.

6. Change the variable. The third part of a `for` statement somehow changes the counting variable. The most common way to change the variable is to add one to it. The `i++` syntax is a shortcut for “add one to i.”

7. Build a code block for repeated code. Use braces and indentation to indicate which code will repeat. All code inside the braces will repeat.

8. Inside the loop, write to the output. On each iteration of the loop, add the current value of `i` to the output div’s `innerHTML`. Also add a break `<br />` to make the output look better. Remember, when you add to an `innerHTML` property, you’re writing HTML code, so if you want the output to occur on different lines, you need to write the HTML to make this happen. (See “Introducing shortcut operators” for an explanation of the `+=` statement.)

9. Close the loop. Don’t forget to end the loop, or your program will not run correctly.

### Introducing shortcut operators

You might have noticed a couple of new operators in the code for `forLoops.html`. These are some shortcut tools that allow you to express common ideas more compactly. For example, consider the following code:

```javascript
i = i + 1;
```

This means “add one to `i`, and store the result back in `i`.” It’s a pretty standard statement, even if it does drive algebra teachers bananas. The statement is so common that it is often abbreviated, like this:

```javascript
i += 1;
```

This statement means exactly the same as the previous one; add one to `i` and store the result back in `i`. You can use this to add any amount to the variable `i`.

Since the `+` sign is used to concatenate (combine) strings, you can use the `+=` shortcut with string manipulation, so consider this variation:
var userName = "Andy";
userName += ", Benevolent Dictator for Life";

The second statement appends my official (I wish) title to the end of my name.

You can also use the -= operator to subtract from a variable. It's even possible to use *=- and /=, but they are not commonly used.

Moving back to numbers; since adding one is extremely common, there's another shortcut that's even more brief:

i++;

This statement also means “add one to i.” In the standard for loop, I use that variation because it's very easy.

When programmers decided to make a new variation of the C language, they called the new language C++. Get it? It's one better than C! Those guys are a hoot!

Making a loop that counts backwards

Once you understand basic for loops, it's not difficult to make a loop that counts backwards. Here's the back() function (called by the count backwards button):

```javascript
function back(){
    output.innerHTML = "";
    for (i = 10; i > 0; i--){
        output.innerHTML += i + "<br/>";
    } // end for loop
} // end back
```

This code is almost exactly like the first loop, but look carefully at how the loop is created:

1. Initialize i to a high value. This time, I want to count backward from ten to one, so start i with the value 10.
2. Keep going as long as i is greater than 0. It's important to note that the logic changes here. If i is greater than zero, the loop should continue. If i becomes 0 or less, the loop will exit.
3. Subtract one from i on each pass. The -- operator works much like ++, but it subtracts one from the variable.

Making a loop that counts by fives

Counting by fives (or any other value) is pretty trivial once you know how for loops work. Here's the byFive() code called by the count by five button:
function byFive()
    output.innerHTML = "";
    for (i = 5; i <= 25; i += 5){
        output.innerHTML += i + "<br />";
    } // end for loop
} // end byFive

It is remarkably similar to the other looping code you've seen.

1. Initialize `i` to 5. The first value I want is 5, so that is the initial value for `i`.
2. Continue as long as `i` is less than or equal to 25. Since I want the value 25 to appear, I set the condition to be less than or equal to 25.
3. Add 5 to `i` on each pass. Each time through the loop, I add 5 to `i` using the `+=` operator.

Manufacturing while Loops

For loops are useful when you know how often a loop will continue, but sometimes you need a more flexible type of loop. The while loop is based on a simple idea. It contains a condition. When the condition is true, the loop continues; if the condition is false, the loop exits.

Creating a basic while loop

Figure BP1-10 shows one in a series of dialog boxes asking for a password. The program keeps asking for a password until the user enters the correct password.

function getPassword()
    //from while.html
    var correct = "HTML5";
    var guess = "";
    while (guess != correct){
        guess = prompt("Password?");
A **while** loop for passwords is not hard to build.

1. **Store the correct password in a variable.** Variable names are important because they can make your code easier to follow. I use the names `correct` and `guess` to differentiate the two types of password. Beginning programmers will often call one of these variables `password`, but that can be confusing because there are actually *two* passwords (the correct password and the guessed password) in play here.

2. **Initialize the `guess` to an empty value.** The key variable for this loop will be `guess`. It will start as an empty string. It’s critical to initialize the key variable before the loop begins.

3. **Set up the **while** statement.** The **while** statement has extremely simple syntax: the keyword **while** followed by a condition, followed by a block of code.

4. **Build the condition.** The condition is the heart of a **while** loop. The condition must be constructed so the loop happens at least once. (Ensure this by comparing the condition to the variable initialization.) When the condition is true, the loop will continue. When the condition is evaluated to false, the loop will exit. This condition compares `guess` to `correct`. If `guess` is not equal to `correct`, the code will continue.

5. **Write the code block.** Use braces and indentation to indicate the block of code that will be repeated in the loop. The only code in this particular loop asks the user for a password.

6. **Add code to change the key variable inside the loop.** Somewhere inside the loop, you need code that changes the value of the key variable. In this example, the `prompt` statement changes the password. As long as the user eventually gets the right password, the loop will end.

**Getting your loops to behave**

**while** loops can be dangerous. It’s quite easy to write a **while** loop that works incorrectly, and these can be an exceptionally difficult kind of bug to find and fix. If a **while** loop is incorrectly designed, it can refuse to ever run or run forever. These endless loops are especially troubling in JavaScript because they will crash the entire browser. If a JavaScript program gets into an endless loop, often the only solution is to use the operating system task manager (Ctrl-Alt-Delete on Windows) to shut down the entire browser.

The easy way to make sure your loop works is to remember that **while** loops need all the same features as **for** loops. (These ideas are built into the structure
of a for loop; you’re responsible for them yourself in a while loop.) If your loop doesn’t work, check that you’ve followed these steps:

1. Identify a key variable. A while loop is normally based on a condition, which is usually a comparison (though it might also be a variable or function that returns a Boolean value). In a for loop, the key variable is almost always an integer. while loops can be based on any type of variable.

2. Initialize the variable before the loop. Before the loop begins, set up the initial value of the key variable to ensure the loop happens at least once.

3. Identify the condition for the loop. A while loop is based on a condition. Define the condition so the loop will continue while the condition is true, and will exit when the condition is evaluated to false.

4. Change the condition inside the loop. Somewhere inside the loop code, you need to have statements that will eventually make the condition false. If you forget this part, your loop will never end.

This example is a terrible way to handle security. The password is shown in the clear, and anybody could view the source code to see the correct password. There are far better ways to handle security, but this is the cleanest example of a while loop I could think of.

Managing more complex loops

It won’t take long before you find situations where the standard for or while loops do not seem adequate. For example, consider the password example again. This time, you want to ask for a password until the user gets the password correct or guesses incorrectly three times. Think about how you would build that code. There are a number of ways to do it, but here’s the cleanest approach:

```javascript
function threeTries(){
   // continues until user is correct or has three // incorrect guesses
   // from while.html

   var correct = "HTML5";
   var guess = ";
   var keepGoing = true;
   var tries = 0;

   while (keepGoing){
      guess = prompt("Password?");
      if (guess == correct){
         alert("You may proceed");
         keepGoing = false;
      } else {
```
Bonus Part 1: Using JavaScript

```javascript
tries++;  
if (tries >= 3){
    alert("Too many tries. Launching missiles...");
    keepGoing = false;
} // end if
} // end if
} // end while
} // end threetries
```

This code is a little more complex, but it uses a nice technique to greatly simplify loops:

1. Initialize correct and guess. As in the previous example, initialize the correct and guess passwords.

2. Build a counter to indicate the number of tries. The `tries` variable will count how many attempts have been made.

3. Build a Boolean sentry variable. The `keepGoing` variable is special. Its entire job is to indicate whether the loop should continue. It is a Boolean variable, meaning it will contain only the values `true` or `false`.

4. Use `keepGoing` as the condition. A condition doesn’t have to be a comparison. It just has to be true or false. Use the Boolean variable as the condition! As long as `keepGoing` has the value `true`, the loop will continue. Anytime you want to exit the loop, set `keepGoing` to `false`.

5. Ask for the password. You still need the password, so get this information from the user.

6. Check to see if the password is correct. Use an `if` statement to see if the password is correct.

7. State what to do if the password is correct. Provide feedback to the user, and set `keepGoing` to `false`. The next time the while statement is executed, the loop will end. (Remember, you want the loop to end when the password is correct.)

8. State what to do if the password is incorrect. If the `(guess == correct)` condition is `false`, that means the user did not get the password correct. In this case, add one to the number of tries.

9. Check the number of tries. Build another `if` statement to check the number of tries.

10. If it’s been three turns, provide feedback (threatening annihilation is always fun) and set `keepGoing` to `false`.

The basic idea of this strategy is quite straightforward: Create a special Boolean variable with the singular job of indicating whether the loop continues. Anytime you want the loop to exit, change the value of that variable.
If you change most of your `while` loops to this format (using a boolean variable as the condition), you'll generally eliminate most `while` loop issues. Most beginners (like me, and I've been doing this for thirty years) make their loops way too complicated.

### Sending Data To and From Functions

Functions make your code safe because variables created inside a function are destroyed when the function dies. Sometimes, though, you want data to move from one function to another. One solution is the global variable, but this is a kind of crude option. A better solution is to allow data to pass into and out of functions. As an example, look at the program in Figure BP1-11.

![Figure BP1-11](image)

**Using functions, parameters, and return values**

The ants go marching 1 by 1, hurray, hurray
The ants go marching 1 by 1, hurray, hurray
The ants go marching 1 by 1, hurray, hurray
The little one stops to suck his thumb

- and they all came marching down, to the ground, to get out, of the rain.
boom boom boom boom

The ants go marching 2 by 2, hurray, hurray
The ants go marching 2 by 2, hurray, hurray
The ants go marching 2 by 2, hurray, hurray
The little one stops to tie his shoe

- and they all came marching down, to the ground, to get out, of the rain.
boom boom boom boom

Of course, this program could be written by creating a really long string variable and then copying it to the `innerHTML` attribute of the `output` div, but that would be quite inefficient. Instead, I used functions to simplify the work. Begin by looking over the main function: `makeSong()`.

```javascript
function makeSong(){
    //create output variable
    //from param.html

    var output = document.getElementById("output");
    ```
This code demonstrates one of the primary advantages of functions; they allow you to break a complex problem into a series of smaller problems. There's a number of interesting things going on here, as follows:

1. The program writes to a div called `output`. I make a variable called `output` that corresponds to a div called `output` on the page.

2. I'm writing text to `output`. That's not surprising, but it is interesting because there are no text variables or values in the `makeSong()` function.

3. All the text is created by other functions. There are two other functions in this program: `verse()` and `chorus()`. Both of these functions create string values.

4. Verse can be fed a numeric value. The `verse` function is especially important because it can be passed a value. The verse changes behavior based on the value passed to it.

### Returning a value from a function

To truly understand what's happening here, begin with the `chorus()` function (because it's a little simpler than `verse()`):

```javascript
function chorus(){
    //from param.html
    var result = "-and they all came marching down, <br />";
    result += "to the ground, to get out, of the rain. <br />"
    result += "boom boom boom boom <br />
    result += "boom boom boom boom <br />
    result += "boom boom boom <br />
    return result;
} // end chorus
```

The `chorus()` function is extremely simple.

1. Create a variable called `result`. This variable will hold the result of the function’s work (which will be a string value containing the chorus of the song).
2. Append HTML code to the result variable. This code has several lines that build up the result. Note that I’m using HTML formatting because this code will be printed in an HTML div.

3. Return result. The last statement of the function is special. The return statement allows you to specify a value that the function returns to whatever code called it.

4. Use the function like a variable. When a function has a return value, you can treat it like a variable. Since this function returns a string, you can use the function like a string variable in the makeSong() function. In that function, I said `output.innerHTML += chorus()`. That means “run the chorus() function and then add whatever comes out of that function to the innerHTML of the output element.”

Sending arguments to a function

The verse() function also returns a value, but it has another trick up its sleeve. While the chorus is always the same, the verse changes a bit each time. The little one (who appears to have attention issues, but we love him anyway) gets distracted in a different way on every verse.

The verse() function uses an important idea called parameter-passing to allow this variation in behavior. Begin by looking at the code for the function:

```javascript
function verse(verseNumber){
    // from param.html
    var distraction = "";
    if (verseNumber == 1){
        distraction = "suck his thumb";
    } else if (verseNumber == 2){
        distraction = "tie his shoe";
    } else {
        distraction = "there's a problem here...";
    } // end if

    var result = "The ants go marching ";
    result += verseNumber + " by " + verseNumber + " , ";
    result += "hurrah, hurrah <br />";
    result += "The ants go marching ";
    result += verseNumber + " by " + verseNumber + " , ";
    result += "hurrah, hurrah <br />";
    result += "The ants go marching ";
    result += verseNumber + " by " + verseNumber + "<br />";
    result += "The little one stops to ";
}```
result += distraction + "<br /> <br />";

return result;
} // end verse

The `verse()` function is very similar to the `chorus()` function, except it is more flexible because it can accept a parameter.

1. Call the function with a value inside the parentheses. When a function is intended to accept a parameter, it must be called with a value inside the parentheses. In `makeSong()`, you’ll see calls to `verseNum(1)` and `verseNum(2)`, but never `verseNum()`. That’s because `verseNum` is designed to always accept a single integer value.

2. Define the function with a variable name inside the parentheses. If you look at the function definition for `verse()`, you see it contains the variable `verseNumber` between the parentheses. Whenever the `verse()` function is called, it must be fed a value, and that value will be placed in the special variable `verseNum`.

3. Use `verseNum` to find the distraction. Analyze the `verseNumber` variable and use it to find the appropriate distraction. Put this in a variable named `distraction`.

4. Build the verse. Incorporate the `verseNumber` and `distraction` variables in the `result` variable.

5. Return the result. The main function will use the returned value as a string, printing out the verse.

Using Arrays to Simplify Data

Computer programs are about data. Often, you’re working with a lot of data. Programmers have a number of tools for managing large amounts of data, but the most basic is the array. JavaScript supports a simple yet very powerful and flexible array mechanism that lets you do plenty with arrays. To see arrays in action, look at Figure BP1-12.

An array is actually a very simple idea; it’s simply a list. You’ve already used lists many times in HTML coding, but in programming, lists are called arrays, and have special characteristics. This example features two arrays — a list of books written by a certain charming and devilishly handsome author, and some of the topics said author writes about.
This page has four main components:

- **HTML structure:** It has a form with two buttons. The body calls an initialization function when it loads, and each button calls its own function. The page also has a `div` named `output`.

- **An `init()` function:** This function provides access to the output `div`, and it also loads up the two arrays described in this example. Arrays usually require some kind of initialization.

- **The `showBooks()` function:** You’ll be amazed and surprised that this function displays a series of book titles.

- **The `showTitles()` function:** This function demonstrates another way to walk through the elements of an array.

### Building the arrays

Arrays are frequently created as global variables because they are often used throughout the program. (And in some languages, passing an array as a parameter can be kind of complicated.)

In this program, I create a number of variables in the global space, and initialize them all in the `init()` function called with `body.onload`:

```javascript
var output;
var books;
```
var topics;

function init(){
  //initialize output and arrays
  //from basicArrays.html
  output = document.getElementById("output");
  books = Array("Flash Game Programming For Dummies",
  "Game Programming: the L Line",
  "HTML, XHTML, and CSS All-in-One For Dummies",
  "JavaScript and AJAX For Dummies",
  "HTML5 Quick Reference For Dummies");
  topics = Array(5);
  topics[0] = "HTML5";
  topics[1] = "CSS3";
  topics[2] = "JavaScript";
  topics[4] = "AJAX";
} // end init

Setting up the arrays is the most important part of the process.

1. Create arrays for the arrays. I have two arrays in this example, books and topics. Each is created just like any other variable, with the var statement. I also create an output variable to hold a reference to the output div.

2. Build an init() function to initialize variables. As programs become more complex, it is common to have an initialization function to set everything up. This function is called with body.onload.

3. Build the output variable. Since all the other functions will use output, I create it in init().

4. Use the Array() function to create the array of books. This special function is used to create an array of elements. Note that it uses an uppercase A. (If you must be technical, this is a constructor for an Array object, but in JavaScript, that’s a function too.)

5. Simply list each book as a parameter in the Array() function. If you feed the Array() function a series of values, they become the values of the array. This technique is great if you already know what will go into each element when you build the array.

6. Build the topics array differently. The topics array is built with a different technique. In this array, I specified a single integer, which is the number of elements the array will contain.
7. Use the index operator to add elements to the array. All array elements have the same name, but they have a different number (corresponding to where they fit on the list). Use square braces with an integer to refer to a specific element in the array. Note that array elements always begin with element zero.

If you've used arrays in other programming languages, you'll find JavaScript arrays to be very forgiving. Be careful, though, because arrays are one of those features that every language supports, but they all work a little bit differently. You'll actually find the JavaScript arrays are more like the ArrayList in Java, or the Vector class in C++, than the traditional array in either of these languages.

Stepping through the books array

Arrays are wonderful because they allow you to pack a lot of data into a single variable. Very often when you have an array, you'll want to do something with each element in the array. The most common structure to do this is a for loop. Look at showBooks() for an example:

```javascript
function showBooks() {
    // from basicArrays.html
    output.innerHTML = "";
    for (i = 0; i < books.length; i++) {
        output.innerHTML += books[i] + "<br "/>";
    } // end for loop
} // end showBooks
```

This function steps through the list of books and prints the name of each one in the output area.

1. Clear the output div. output has already been defined in the init() function, so it's pretty easy to clear out its value in the function.

2. Build a loop for the length of the array. Arrays and for loops are natural partners. In this loop, I have i count from zero to the number of elements in the array.

3. Begin with zero. Array indices always begin with zero, so your counting variable should also start at zero (in most cases).

4. Use the length property to determine how long the array is. When you build a for loop to step through an array, what you really want to know is how many elements are in the array. Use arrayName.length to determine the number of elements in the current array, where arrayName is the name of the current array. This way, even if the number of elements in the array changes, the loop will still work correctly.
5. Process data inside the loop. If the counting variable is \( i \), each element of the array will be \( \text{arrayName}[i] \) inside the loop. You can do what you want with the data. In my example, I'm simply printing it out.

Using the for . . . in loop to access array elements
The `showTopics()` function uses a special variation of the for loop to print out the contents of an array.

```javascript
function showTopics(){
    // from basicArrays.html
    output.innerHTML = "";
    for (topicID in topics){
        output.innerHTML += topics[topicID] + "<br "/>";
    } // end for
} // end showTopics
```

Because loops and arrays are so commonly linked, JavaScript provides a special shortcut version of the for loop just for working with array elements.

1. Clear the output area. If there's already data in the `output` div, clean it out before printing anything new there.

2. Use the for...in loop variant. The for loop in this function is quite a bit simpler than most. It simply has a variable and an array. This loop will repeat once per element in the list.

3. Use the `topicID` variable to refer to each element in the loop. The `topicID` variable will contain each index used in the array. Use this index to determine the value associated with that index.

The for...in loop is really great, and if you know PHP, it looks at first glance just like the PHP `foreach` loop. However, they are not exactly the same. The JavaScript version returns the key, and the PHP version returns the associated value. I get confused every time I switch between the two languages.

I've added another variation of the ants program to my Web site that uses arrays. Look over this code (`antsArray.html`) if you want to see an example of how arrays can further simplify that program.