3. **Place the cursor before the first line of code (the line that begins with *var luckyNumber*) and type:**

   ```
   do {
   ```

   This code creates the beginning of the loop. Next, you'll finish the loop and add the test condition.

4. **Click at the end of the last line of JavaScript code in that section and type: *} while (isNaN(luckyNumber));*. The completed code block should look like this:**

   ```
   do {
    var luckyNumber = prompt('What is your lucky number?','');
    luckyNumber = parseInt(luckyNumber, 10);
   } while (isNaN(luckyNumber));
   ```

Save this file and preview it in a Web browser. Try typing text and other nonnumeric symbols in the prompt dialog. That annoying dialog continues to appear until you actually type a number.

Here's how it works: the *do* keyword tells the JavaScript interpreter that it's about to enter a *do/while* loop. The next two lines are then run, so the prompt appears and the visitor's answer is converted to a whole number. It's only at this point that the condition is tested. It's the same condition as the script on page 89: it just checks to see if the input retrieved from the visitor is "not a number." If the input isn't a number, the loop repeats. In other words, the prompt will keep reappearing as long as a nonnumber is entered. The good thing about this approach is that it guarantees that the prompt appears at least once, so if the visitor does type a number in response to the question, there is no loop.

Functions: Turn Useful Code Into Reusable Commands

Imagine that at work you've just gotten a new assistant to help you with your every task (time to file this book under "fantasy fiction"). Suppose you got hungry for a piece of pizza, but since the assistant was new to the building and the area, you had to give him detailed directions: "Go out this door, turn right, go to the elevator, take the elevator to the first floor, walk out of the building…" and so on. The assistant follows your directions and brings you a slice. A couple hours later you're hungry again, and you want more pizza. Now, you don't have to go through the whole set of directions again— "Go out this door, turn right, go to the elevator…". By this time, your assistant knows where the pizza joint is, so you just say, "Get me a slice of pizza," and he goes to the pizza place and returns with a slice.

In other words, you only need to provide detailed directions a *single time*; your assistant memorizes those steps and with the simple phrase "Get me a slice" he instantly leaves and reappears a little while later with a piece of pizza. JavaScript has an equivalent mechanism called a *function*. A function is a series of programming steps that you set up at the beginning of your script—the equivalent of providing detailed directions to your assistant. Those steps aren't actually run when you create the function; instead, they're stored in the Web browser's memory, where you can call upon them whenever you need those steps performed.

Functions are invaluable for efficiently performing multiple programming steps repeatedly: for example, say you create a photo gallery Web page filled with 50 small thumbnail images. When someone clicks one of the small photos, you might want the page to dim, a caption to appear, and a larger version of that image to fill the screen (you'll learn to do just that on page 254). Each time someone clicks an image, the process repeats, so on a Web page with 50 small photos, your script might have to do the same series of steps 50 times. Fortunately, you don't have to write the same code 50 times to make this photo gallery work. Instead, you can write a function with all the necessary steps, and then, with each click of the thumbnail, you run the function. You write the code once, but run it any time you like.

The basic structure of a function looks like this:

```
function functionName() {
 // the JavaScript you want to run
}
```

The keyword *function* lets the JavaScript interpreter know you're creating a function—it's similar to how you use *if* to begin an *if/else* statement or *var* to create a variable. Next you provide a function name; as with a variable, you get to choose your own function name. Follow the same rules listed on page 44 for naming variables. In addition, it's common to include a verb in a function name like *calculateTax*, *getScreenHeight*, *updatePage*, or *fadeImage*. An active name makes it clear that it does something and makes it easier to distinguish between function and variable names.

Directly following the name, you add a pair of parentheses, which are another characteristic of functions. After the parentheses, there's a space followed by a curly brace, one or more lines of JavaScript and a final, closing curly brace. As with *if* statements, the curly braces mark the beginning and end of the JavaScript code that make up the function.

Tip: As with *if/else* statements, functions are more easily read if you indent the JavaScript code between the curly braces. Two spaces (or a tab) at the beginning of each line are common.

Here's a very simple function to print out the current date in a format like "Sun May 12 2008":

```
function printToday( ) {
 var today = new Date( );
 document.write(today.toDateString( ));
}
```

The function's name is *printToday*. It has just two lines of JavaScript code that retrieve the current date, convert the date to a format we can understand (that's the *toDateString( )* part), and then print the results to the page using our old friend the *document.write( )* command. Don't worry about how all of the date stuff works—you'll find out in the next chapter.

Programmers usually put their functions at the beginning of a script, which sets up the various functions that the rest of the script will use later. Remember that a function doesn't run when it's first created—it's like telling your assistant how to get to the pizza place without actually sending him there. The JavaScript code is merely stored in the browser's memory, waiting to be run later, when you need it.

But how do you run a function? In programming-speak you *call* the function whenever you want the function to perform its task. Calling the function is just a matter of writing the function's name, followed by a pair of parentheses. For example, to make our *printToday* function run, you'd simply type:

```
printToday( );
```

As you can see, making a function run doesn't take a lot of typing—that's the beauty of functions. Once they're created, you don't have to add much code to get results.

Note: When calling a function, don't forget the parentheses following the function. That's the part that makes the function run. For example, *printToday* won't do anything, but *printToday()* executes the function.

Mini-Tutorial

Because functions are such an important concept, here's a series of steps for you to practice creating and using a function on a real Web page:

1. **In a text editor, open the file *3.2.html*.**

   You'll start by adding a function in the head of the document.

2. **Locate the code between the <script> tags in the <head> of the page, and type the following code:**

   ```
   function printToday( ) {
    var today = new Date( );
    document.write(today.toDateString( ));
   }
   ```

   The basic function is in place, but it doesn't do anything yet.

3. **Save the file and preview it in a Web browser.**

   Nothing happens. Well, actually something does happen; you just don't see it. The Web browser read the function statements into memory, and was waiting for you to actually call the function, which you'll do next.

4. **Return to your text editor and the *3.2.html* file. Locate the <p> tag that begins with "Today is", and between the two <strong> tags, add the following bolded code:**

```
<p>Today is <strong><script type="text/javascript">printToday( );↵
</script></strong></p>
```

Note: Remember, when you see the ↵ character, that just means the full line of code wouldn't fit across the page of this book. You just type the code on one line in your text editor. Don't start a new line, and don't attempt to type a ↵ character.

Save the page and preview it in a Web browser. The current date is printed to the page. If you wanted to print the date at the bottom of the Web page as well, all you'd need to do is call the function a second time.

Giving Information to Your Functions

Functions are even more useful when they receive information. Think back to your assistant—the fellow who fetches you slices of pizza. The original "function" described on page 97 was simply directions to the pizza parlor and instructions to buy a slice and return to the office. When you wanted some pizza, you "called" the function by telling your assistant "Get me a slice!" Of course, depending on how you're feeling, you might want a slice of pepperoni, cheese, or olive pizza. To make your instructions more flexible, you can tell your assistant what type of slice you'd like. Each time you request some pizza, you can specify a different type.

JavaScript functions can also accept information, called *parameters*, which the function uses to carry out its actions. For example, if you want to create a function that calculates the total cost of a person's shopping cart, then the function needs to know how much each item costs, and how many of each item was ordered.

To start, when you create the function, place the name of a new variable inside the parentheses—this is the *parameter*. The basic structure looks like this:

```
function functionName(parameter) {
 // the JavaScript you want to run
}
```

The parameter is just a variable, so you supply any valid variable name (see page 44 for tips on naming variables). For example, let's say you want to save a few keystrokes each time you print something to a Web page. You create a simple function that lets you replace the Web browser's *document.write( )* function with a shorter name:

```
function print(message) {
 document.write(message);
}
```

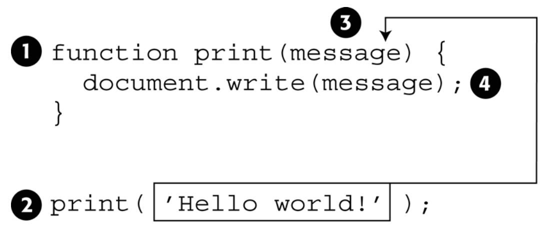
The name of this function is *print* and it has one parameter, named *message*. When this function is called, it receives some information (the message to be printed) and then it uses the *document.write( )* function to write the message to the page. Of course, a function doesn't do anything until it's called, so somewhere else on your Web page, you can call the function like this:

```
print('Hello world.');
```

When this code is run, the print function is called and some text—the string 'Hello world.'—is sent to the function, which then prints "Hello World." to the page. Technically, the process of sending information to a function is called "passing an argument." In this example, the text—'Hello world.'—is the *argument*.

Even with a really simple function like this, the logic of when and how things work can be a little confusing if you're new to programming. Here's how each step breaks down, as shown in the diagram in Figure 3-5:

1. **The function is read by the JavaScript interpreter and stored in memory. This step just prepares the Web browser to run the function later.**
2. **The function is called and information—"Hello world."—is passed to the function.**
3. **The information passed to the function is stored in a new variable named *message*. This step is equivalent to *var message = 'Hello World.';***
4. **Finally, the function runs, printing the value stored in the variable *message* to the Web page.**



```
function print(message) {
  document.write(message);
}

print('Hello world!');
```

Figure 3-5:
When working with functions, you usually create the function before you use it. The print( ) *function here is created in the first three lines of code, but the code inside the function doesn't actually run until the last line.*

A function isn't limited to a single parameter, either. You can pass any number of arguments to a function. You just need to specify each parameter in the function, like this:

```
function functionName(parameter1, parameter2, parameter3) {
 // the JavaScript you want to run
}
```

And then call the function with the same number of arguments in the same order:

```
functionName(argument1, argument2, argument3);
```

In this example, when *functionName* is called, *argument1* is stored in *parameter1*, *argument2* in *parameter2,* and so on. Expanding on the print function from above, suppose in addition to printing a message to the Web page, you want to specify an HTML tag to wrap around the message. This way, you can print the message as a headline or a paragraph. Here's what the new function would look like:

```
function print(message,tag) {
 document.write('<' + tag + '>' + message +'</' + tag + '>');
}
```

The function call would look like this:

```
print('Hello world.', 'p');
```

In this example, you're passing two arguments—'Hello world.' and 'p'—to the function. Those values are stored in the function's two variables—*message* and *tag*. The result is a new paragraph—*<p>Hello world.</p>*—printed to the page.

You're not limited to passing just strings to a function either: you can send any type of JavaScript variable or value to a function. For example, you can send an array, a variable, a number, or a Boolean value as an argument.

Retrieving Information from Functions

Sometimes a function simply does something like write a message to a page, move an object across the screen, or validate the form fields on a page. Other times, you'll want to get something back from a function: after all, the "Get me a slice of pizza" function wouldn't be much good if you didn't end up with some tasty pizza at the end. Likewise, a function that calculates the total cost of items in a shopping cart isn't very useful unless the function lets you know the final total.

Some of the built-in JavaScript functions we've already seen return values. For example the *prompt( )* command (see page 55) pops up a dialog box with a text field, whatever the user types in to the box is returned. As you've seen, you can then store that return value into a variable and do something with it:

```
var answer = prompt('What month were you born?', '');
```

The visitor's response to the prompt dialog is stored in the variable *answer*; you can then test the value inside that variable using conditional comments or do any of the many other things JavaScript lets you do with variables.

To return a value from your own functions, you use *return* followed by the value you wish to return:

```
function functionName(parameter1, parameter2) {
 // the JavaScript you want to run
 return value;
}
```

For example, say you want to calculate the total cost of a sale including sales tax. You might create a script like this:

```
var TAX = .08; // 8% sales tax
function calculateTotal(quantity, price) {
 var total = quantity * price * (1 + TAX);
 var formattedTotal = total.toFixed(2);
 return formattedTotal;
}
```

The first line stores the tax rate into a variable named *TAX* (which lets you easily change the rate simply by updating this line of code). The next three lines define the function. Don't worry too much about what's happening inside the function—you'll learn more about working with numbers in the next chapter. The important part is the fourth line of the function—the return statement. It returns the value stored in the variable *formattedTotal*.

To make use of the return value you usually store it inside a variable, so in this example, you could call the function like this:

```
var saleTotal = calculateTotal(2, 16.95);
document.write('Total cost is: $' + saleTotal);
```

In this case, the values 2 and 16.95 are passed to the function. The first number represents the number of items purchased, and the second their individual cost. The result is returned from the function and stored into a new variable—*sale-Total*—which is then used as part of a *document.write( )* command to print the total cost of the sale including tax.

You don't have to store the return value into a variable, however. You can use the return value directly within another statement like this:

```
document.write('Total: $' + calculateTotal(2, 16.95));
```

In this case, the function is called and its return value is added to the string 'Total: $', which is then printed to the document. At first, this way of using a function may be hard to read, so you might want to take the extra step of just storing the function's results into a variable and then using that variable in your script.

Tip: A function can only return one value. If you want to return multiple items, store the results in an array and return the array.

Keeping Variables from Colliding

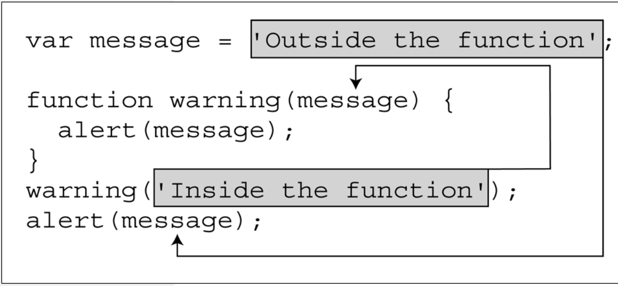
One great advantage of functions is that they can cut down the amount of programming you have to do. You'll probably find yourself using a really useful function time and time again on different projects. For example, a function that helps calculate shipping and sales tax could come in handy on every order form you create, so you might copy and paste that function into other scripts on your site or on other projects.

One potential problem arises when you just plop a function down into an already created script. What happens if the script uses the same variable names as the function? Will the function overwrite the variable from the script, or vice versa? For example:

```
var message = 'Outside the function';
function warning(message) {
    alert(message);
}
warning('Inside the function'); // 'Inside the function'
alert(message); // 'Outside the function'
```

Notice that the variable *message* appears both outside the function (the first line of the script) and as a parameter in the function. A parameter is really just a variable that's filled with data when the function's called. In this case, the function call—*warning('Inside the function');*—passes a string to the function and the function stores that string in the variable *message*. It looks like there are now two versions of the variable *message*. So what happens to the value in the original *message* variable that's created in the first line of the script?

You might think that the original value stored in *message* is overwritten with a new value, the string 'Outside the function'; it's not. When you run this script, you'll see two alert dialogues: the first will say "Inside the function" and the second "Outside the function." There are actually two variables named *message*, but they exist in separate places (see Figure 3-6).



```
var message = 'Outside the function';

function warning(message) {
  alert(message);
}
warning('Inside the function');
alert(message);
```

Figure 3-6:
*A function parameter is only visible inside the function, so the first line of this function–*function warning(message)*–will create a new variable named message that can only be accessed inside the function. Once the function is done, that variable disappears.*

The JavaScript interpreter treats variables inside of a function differently than variables declared and created outside of a function. In programming-speak, each function has its own *scope*. A function's scope is like a wall that surrounds the function—variables inside the wall aren't visible to the rest of the script outside the wall. Scope is a pretty confusing concept when you first learn about it, but it's very useful. Because a function has its own scope, you don't have to be afraid that the names you use for parameters in your function will overwrite or conflict with variables used in another part of the script.

So far, the only situation we've discussed is the use of variables as parameters. But what about a variable that's created inside the function, but not as a parameter, like this:

```
var message = 'Outside the function';
function warning( ) {
 var message ='Inside the function';
 alert( message );
}
warning( ); // 'Inside the function'
alert( message ); //'Outside the function'
```

Here, the code creates a *message* variable twice—in the first line of the script, and again in the first line inside the function. This situation is the same as with parameters—by typing *var message* inside the function, you've created a new variable inside the function's scope. This type of variable is called a *local variable*, since it's only visible within the walls of the function—the main script and other functions can't see or access this variable.

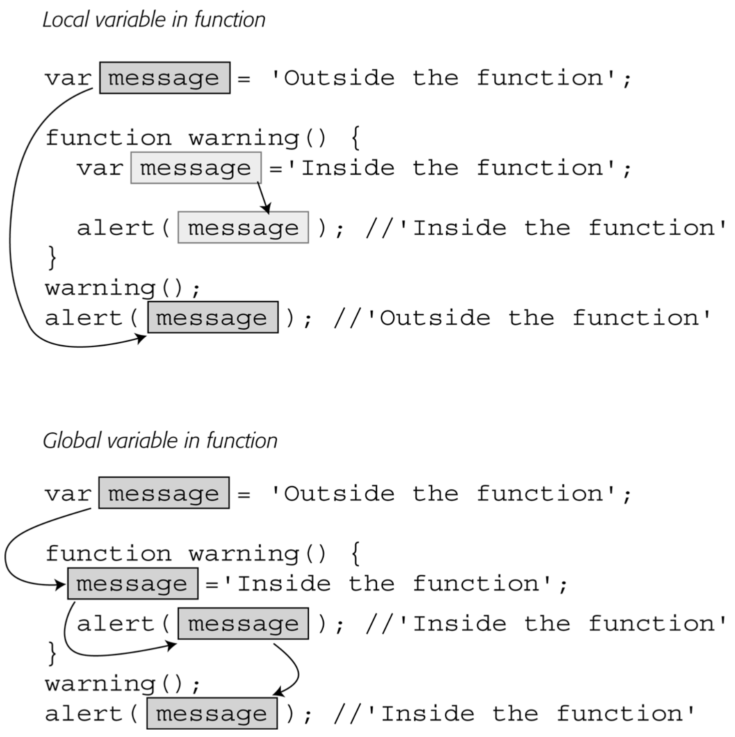
However, variables created in the main part of a script (outside a function) exist in *global scope*. All functions in a script can access variables that are created in its main body. For example, in the code below, the variable message is created on the first line of the script—it's a *global variable*, and it can be accessed by the function.

```
var message = 'Global variable';
function warning( ) {
 alert( message );
}
warning( ); // 'Global variable'
```

This function doesn't have any parameters and doesn't define a *message* variable, so when the *alert(message)* part is run, the function looks for a global variable named *message*. In this case, that variable exists, so an alert dialog with the text "Global variable" appears.

There's one potential gotcha with local and global variables—a variable only exists within the function's scope if it's a parameter, or if the variable is created inside the function with the *var* keyword. Figure 3-7 demonstrates this situation. The top chunk of code demonstrates how both a global variable named *message* and a function's local variable named *message* can exist side-by-side. The key is the first line inside the function—*var message ='Inside the function';*. By using *var*, you create a local variable.

Compare that to the code in the bottom half of Figure 3-7. In this case, the function doesn't use the *var* keyword. Instead, the line of code *message='Inside the function';* doesn't create a new local variable; it simply stores a new value inside the global variable *message*. The result? The function clobbers the global variable, replacing its initial value.



```
var message = 'Outside the function';

function warning() {
  var message ='Inside the function';

  alert( message ); //'Inside the function'
}
warning();
alert( message ); //'Outside the function'
```

```
var message = 'Outside the function';

function warning() {
  message ='Inside the function';
  alert( message ); //'Inside the function'
}
warning();
alert( message ); //'Inside the function'
```

Figure 3-7: *There's a subtle yet crucial difference when assigning values to variables within a function. If you want the variable to only be accessible to the code inside the function, make sure to use the* var *keyword to create the variable inside the function (top). If you don't use* var*, you're just storing a new value inside the global variable (bottom).*

The notion of variable scope is pretty confusing, so the preceding discussion may not make a lot of sense for you right now. But just keep one thing in mind: if the variables you create in your scripts don't seem to be holding the values you expect, you might be running into a scope problem. If that happens, come back and reread this section.

Tutorial: A Simple Quiz

Now it's time to bring together the lessons from this chapter and create a complete program. In this tutorial, you'll create a simple quiz system for asking questions and evaluating the quiz-taker's performance. First, this section will look at a couple of ways you could solve this problem, and discuss efficient techniques for programming.

As always, the first step is to figure out what exactly the program should do. There are a few things you want the program to accomplish:

- **Ask questions.** If you're going to quiz people, you need a way to ask them questions. At this point, you know one simple way to get feedback on a Web page: the *prompt( )* command. In addition, you'll need a list of questions; since arrays are good for storing lists of information, you'll use an array to store your quiz questions.