CHAPTER OUTLINE

10.1 Introduction
10.2 while Loop
10.3 External JavaScript Files
10.4 Compound Interest Web Page
10.5 do Loop
10.6 Radio Buttons
10.7 Checkboxes
10.8 Job Skills Web Page
10.9 for Loop
10.10 fieldset and legend Elements
10.11 Manipulating CSS with JavaScript
10.12 Using z-index to Stack Elements on Top of Each Other
10.13 Textarea Controls
10.14 Dormitory Blog Web Page
10.15 Pull-Down Menus
10.16 List Boxes
10.17 Case Study: Collector Performance Details and Nonredundant Website Navigation

10.1 Introduction

In the previous two chapters, you learned about the basic building blocks needed to implement interactive web pages. Specifically, you learned about forms, buttons, text controls, number controls, event handlers, and JavaScript. In this chapter, you'll add to your tool bag, so you can implement a wider variety of web pages. You'll learn how to make those web pages look better and behave more dynamically.

In this chapter, we introduce controls that can be grouped together, such as radio buttons and checkboxes. You'll learn how to access and update those controls by using JavaScript loop statements to process the individual values within the control's group of values. You'll also use JavaScript to dynamically modify a web page's CSS formatting. You'll learn a new type of formatting with the CSS `z-index` property. It enables you to stack elements on top of each other, and you'll use JavaScript to modify the stacking order. Finally, you'll learn about pull-down menus and list boxes, which allow users to select one or more values from a list of choices.

10.2 `while` Loop

For many programming tasks, you'll need to perform the same operation repeatedly (e.g., adding a group of numbers to find their sum). To perform operations repeatedly, you'll need to use a loop statement. JavaScript provides three types of loop statements—`while` loop, `do` loop, and `for` loop. We'll cover the `while` loop in this section, and the `do` and `for` loops later in the chapter.

Syntax and Semantics

The `while` loop is the most flexible of the three types of loops. You can use it for any task that needs repetitive operations. **FIGURE 10.1** shows the syntax and semantics for the `while` loop. The syntax at the left of the figure should look familiar because it's similar to the `if` statement

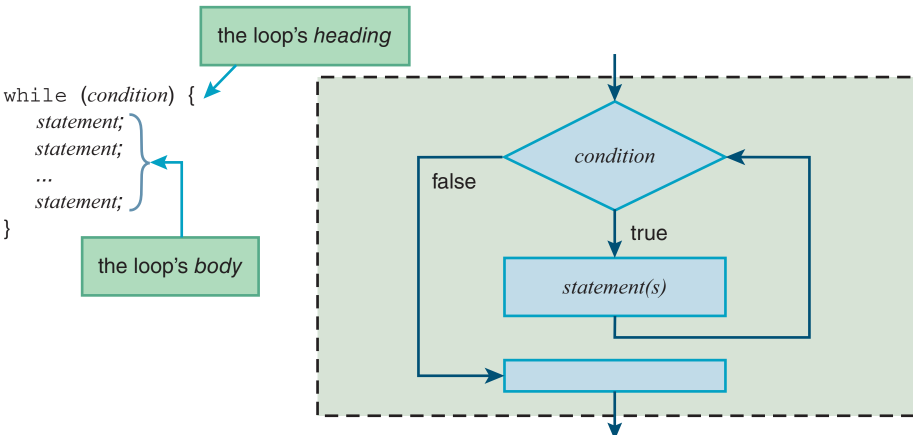


FIGURE 10.1 Syntax and semantics for the `while` loop

syntax. In the `while` loop's heading, after the reserved word `while`, you need parentheses, a condition, and an opening brace. As you know, a condition is a question that evaluates to true or false. In the `while` loop's body, you can have as many statements as you like. Below the body's statements, you indicate the end of the `while` loop with a closing brace.

In terms of style, the `while` loop is pretty much the same as the `if` statement. You should put a space between the condition and the opening brace. Also, don't forget to indent the statements within the braces and align the closing brace with the first character in the loop's heading.

In using loops, you'll need to get comfortable with the jargon. The number of times that a loop repeats is referred to as the number of *iterations*. It's possible for a loop to repeat forever. That's known as an *infinite loop*, and it's usually indicative of a bug. It's also possible for a loop to repeat zero times. There's no special term for the zero iteration occurrence, but it's important to be aware that this sometimes happens.

Study Figure 10.1's flowchart and make sure you understand the `while` loop's semantics. If the `while` loop's condition evaluates to true, then the statements within the loop are executed and control then flows back to the top of the loop, where the condition gets checked again. That continues until the condition becomes false. At that point, control flows down to whatever is below the `while` loop.

Tracing an Example

The code fragment in **FIGURE 10.2** uses a `while` loop to calculate the factorial of a user-entered number. The code comes from an exercise at the end of this chapter. The exercise shows the code as part of a complete web page, so go there now if you're curious. The main point of the code fragment is to show how a `while` loop can be used to implement a solution that requires repetitive operations. As you probably recall from a middle school math class, the factorial of a nonnegative integer x is denoted by $x!$ To calculate the factorial of x, you multiply all the integers from 1 up to x. So 4! equals 1 • 2 • 3 • 4, which equals 24. In the code fragment, the `while` loop uses a `count` variable as the multiplicand for each loop iteration multiplication operation.

```
num = form.elements["number"].valueAsNumber;
factorial = 1;
count = 2;

while (count <= num) {
  factorial *= count;
  count++;
}

form.elements["result"].value = num + "! = " + factorial;
```

The `while` loop performs multiple multiplication operations: 1 * 2 * 3 * … * `num`

FIGURE 10.2 Code fragment that uses a `while` loop to calculate a factorial

Let's trace the code fragment. *Tracing* is where you essentially pretend you're the computer. You step through the program line by line and carefully record what happens. For many traces, the outcome is dependent on user input. For the factorial web page, the user enters a value into a number box, and when the user clicks the form's button, the JavaScript retrieves the user's entered value. We don't know what a real user will enter, but we need to provide an assumed input value so we can proceed with the trace. Let's assume the user enters 3 into the number box.

The code fragment's first statement uses `form.elements["number"]` to retrieve a control that has an `id` value of `number`. If you look at the complete web page code in the exercise at the end of this chapter, you can see that the number box has an `id` value of `number`. The number box's input value, which we assume to be 3, gets assigned into the `num` variable. The second and third statements assign 1 and 2 to the `factorial` and `count` variables, respectively. Remember, the point of a trace is to carefully record what happens. Here's how you should record the trace after executing the first three lines:

`num`	`factorial`	`count`	`result` element
3	1	2	

The `while` loop's heading checks the condition `count <= num`. Because 2 is less than 3, the condition is true, and the loop's body gets executed. The loop body's first statement multiplies `factorial` times `count` and puts the result back into the `factorial` variable. The loop body's second statement increments `count`. Here's what your trace should look like after executing those two statements:

`num`	`factorial`	`count`	`result` element
3	~~1~~	~~2~~	
	2	3	

Note how 1 and 2 are crossed off. After a variable's value changes, you should cross off its old value so the old value doesn't accidentally get reused later.

Continuing with the trace, the next step is to jump back to the `while` loop's heading and check the condition again. Is the condition `count <= num` still true? Yes, `count`, 3, is equal to `num`, 3. So the loop's body gets executed again, and here's what your trace should look like after that execution:

`num`	`factorial`	`count`	`result` element
3	~~1~~	~~2~~	
	~~2~~	~~3~~	
	6	4	

Going back to the top of the loop, and checking the condition, `count` is 4 and `num` is 3, so the condition is false. The next step is to jump below the loop and execute the bottom statement. The bottom statement concatenates three entities: `num`'s value, "! = ", and `factorial`'s value. The concatenated result is "3! = 6". That string is then assigned into the control specified by `form.elements["result"]`. In the trace, we represent that control with the heading "`result` element." Here's what your final trace should look like:

`num`	`factorial`	`count`	`result` element
3	~~1~~	~~2~~	
	~~2~~	~~3~~	
	6	4	
			`3! = 6`

Generating the Table with JavaScript

FIGURE 10.5 shows the JavaScript file for the Compound Interest web page. Note the prologue section at the top of the file. You should include a prologue at the top of every one of your JavaScript files. A *prologue* is a block comment that provides information about the file, so someone who's interested in the file can quickly get an idea of what the file is all about.

In the figure, note how the prologue begins with `/*` and ends with `*/`. As you know, those characters are required to mark the beginning and end of a JavaScript block comment. To make the prologue's information stand out, it's common to enclose the prologue's information in a box of asterisks. Note how the "box" is formed with an asterisk line at the top, single asterisks at the left edge, and an asterisk line at the bottom. Within the box, include the filename, the programmer's name, a blank line, and a description of the file's code.

Below the prologue, you can see that the file contains just one thing—the `generateTable` function definition. It's common for a JavaScript file to contain multiple function definitions, where the functions are called from different web pages, but in this relatively simple example, there's just one function definition and one web page.

The `generateTable` function starts by retrieving the values from the three number controls. The rest of the function is all about building the code that implements the table that displays the compound interest results. That code gets built by assigning and concatenating code strings into

start of the block comment

end of the block comment

prologue

```
/******************************************************
* compoundInterest.js
* John Dean
*
* This file contains a function that supports the
* compound interest web page.
******************************************************/

// This function generates a compound interest table.

function generateTable(form) {
  var amount;    // accumulated value for each new year
  var rate;      // interest rate
  var years;     // years for principal to grow
  var interest;  // interest earned each year
  var table;     // compound interest table
  var year = 1;  // the year being calculated

  amount = form.elements["deposit"].valueAsNumber;
  rate = form.elements["rate"].valueAsNumber;
  years = form.elements["years"].valueAsNumber;

  table =
    "<table>" +
    "<tr><th>Year</th><th>Starting Value</th>" +
    "<th>Interest Earned</th><th>Ending Value</th></tr>";

  while (year <= years) {
    table += "<tr>";
    table += "<td>" + year + "</td>";
    table += "<td>$" + amount.toFixed(2) + "</td>";
    interest = amount * rate / 100;
    table += "<td>$" + interest.toFixed(2) + "</td>";
    amount += interest;
    table += "<td>$" + amount.toFixed(2) + "</td>";
    table += "</tr>";
    year++;
  } // end while

  table += "</table>";
  document.getElementById("result").innerHTML = table;
} // end generateTable
```

FIGURE 10.5 External JavaScript file for Compound Interest web page

the `table` variable. The first such assignment takes care of the `table` start tag and the first `tr` element with its four `th` cells:

```
table =
  "<table>" +
  "<tr><th>Year</th><th>Starting Value</th>" +
    "<th>Interest Earned</th><th>Ending Value</th></tr>";
```

Go back to Figure 10.3's browser window and verify that the displayed table's first row matches the `<tr>` content in the preceding code.

The subsequent assignments use the `+=` compound assignment operator to concatenate additional code onto the end of the `table` variable. Note how those compound assignment operators are inside a loop. Each loop iteration implements a new `tr` container with its `td` cells. More specifically, each loop iteration starts by concatenating a `tr` start tag and ends by concatenating a `tr` end tag. In between those `+=` operations, you can see `+=` operations for a row's four `td` elements. Those `td` elements contain values for the current year, the year's starting balance, the calculated interest, and the calculated ending balance. Get used to the technique exhibited here when you need to build a rather complicated string value. Start by initializing a string variable and then incrementally append to it by using the `+=` operator. Very useful!

The function's last statement assigns the just-built `table` variable to the placeholder `div` element at the bottom of the web page. Here's that statement:

```
document.getElementById("result").innerHTML = table;
```

The `getElementById` method retrieves the "result" element. Going back to the web page's body container, you can see `id="result"` in the bottom `div` element. As you know, the `innerHTML` property is how you access the interior between the element's start and end tags. So assigning the `table` variable there causes the compound interest table to display. Yay!

JavaScript Debugging

All of the major browsers have debugging tools built in. Chrome's debugging tool is called "Developer Tools." To load it while viewing a web page with Chrome, you press ctrl+shift+i for Windows computers or cmd+opt+i for Mac computers. To get practice using Developer Tools, you should download the Chrome Developer Tools tutorial on the book's website and work your way through the tutorial's instructions. It uses the Compound Interest web page to illustrate how to use the debugger tool to find bugs and fix them.

If you don't have time to learn all of the debugger's features, that's OK, but you should at least open the debugger and take advantage of its *console frame*. As you execute JavaScript on a web page, if there's a syntax error, the console frame displays a message that describes the error and provides a link to the errant line in the source code. That can be very helpful. Also, to help with debugging, you can call `console.log` with a message as an argument and the message gets displayed in the debugger's console frame. Very helpful again! For example, suppose you've got an event handler that calculates the total cost of a user's purchase, and you want to display the `cost` variable's value. The following `console.log` method call displays the `cost` variable's value in the console frame:

```
console.log("cost = " + cost);
```

10.5 do Loop

As mentioned earlier, JavaScript has three types of loops—the `while` loop, `do` loop, and `for` loop. Next up—the `do` loop.

Syntax and Semantics

Note the `do` loop's syntax template at the left of **FIGURE 10.6**. It shows the `do` loop's condition at the bottom. That contrasts with the `while` loop, where the condition is at the top. Having the condition tested at the bottom guarantees that the `do` loop executes at least one time. After all, the condition is the only way to terminate the loop, and the JavaScript engine won't check the condition (at the bottom) until after executing the lines above it. In the syntax template, note the semicolon at the right of the condition. That's also different from the `while` loop. Finally, note that the `while` part is on the same line as the closing brace—that's good style. It's legal to put `while(`*condition*`);` on the line after the closing brace, but that would be bad style because it would look like you're trying to start a new `while` loop.

With a `while` loop, with its condition at the top, if the condition starts out with a false value, the JavaScript engine will execute the loop body zero times. For most looping situations, it's appropriate to accommodate the possibility of zero iterations. But for those situations where zero iterations doesn't make sense—that is, when you're sure that the loop body should be executed at least one time—it's more efficient to use a `do` loop.

Powers of 2 Web Page

FIGURE 10.7 shows a web page that uses a `do` loop as part of its event handler code. Let's start by figuring out why using a `do` loop is appropriate. The web page asks the user to enter the largest power of 2 he or she can think of. If the user enters a number less than 10, the web page tells the user to enter a larger number. After the user submits an answer, the button's event handler determines whether the entered number is indeed a power of 2. It does so by repeatedly dividing by 2 until the result is

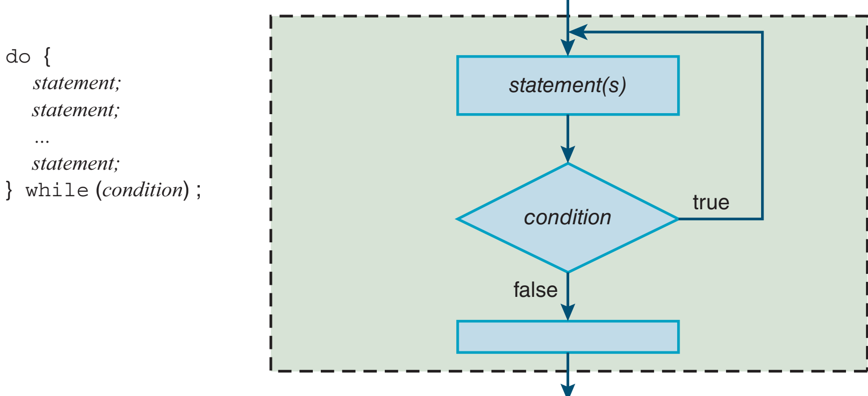
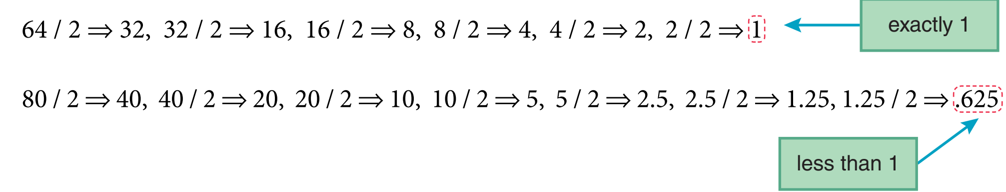


FIGURE 10.6 Syntax and semantics for the `do` loop

© Dawn Hudson/ShutterStock, Inc.

FIGURE 10.7 Powers of 2 web page—initial display and what happens after the user enters a correct value and clicks the button

less than or equal to 1. If the result is exactly 1, that means the entered number is a power of 2. For example, the following division operations show that 64 is a power of 2, and 80 is not a power of 2:



Because the user is forced to enter a number 10 or greater, you're guaranteed to need to divide by 2 at least once. That means a `do` loop is appropriate.

We'll get to the `do` loop code soon enough, but let's examine the HTML and CSS code first. **FIGURE 10.8A** shows the web page's `body` container. As you skim through it, note the `body` container's two child elements—a form and an image. By default, a form is a block element, so it would normally span the width of the web page's viewport, causing the image to display below the form. To get the image to display at the right of the form, we use a little CSS magic….

We introduced the flexible box layout in an earlier chapter and used it to center a web page's contents horizontally. This time, we use it to "flex" the size of the `body` container's two child elements so they conform to the size of their contents. Take a look at **FIGURE 10.8B**'s `style` container, and note this flexbox CSS rule:

```
body {display: flex; align-items: flex-start;}
```

The `display: flex` property-value pair converts the `body` container into a flexbox and causes the form's width to conform to the size of its content (and not span the width of the web page). The `align-items: flex-start` property value causes the flex container's child elements to be aligned at the top. The `style` container's next two rules tweak the layout's margins to further improve the layout:

```
form, img {margin: 20px 20px 0;}
h1 {margin-top: 0;}
```

The `style` container's last CSS rule applies a very light shade of pink to the web page's background:

```
body {background-color: rgb(255, 246, 250);}
```

```
<body>
<form>
  <h1>MATH IS FUN!</h1>
  <label for="number">What is the largest power of 2
    you can think of?</label>
  <br><br>
  <input type="number" id="number"
     min="10" step="1" required>
  <br><br>
  <input type="button" value="Submit"
    onclick="checkForPowerOf2(this.form);">
  <br><br>
  <output id="result"></output>
</form>
<img src="../images/girlJuggling.png" width="128" height="212" alt="">
</body>
</html>
```

FIGURE 10.8A **`body` container for Powers of 2 web page**

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author" content="John Dean">
<title>Powers of 2</title>
<style>
  body {display: flex; align-items: flex-start;}
  form, img {margin: 20px 20px 0;}
  h1 {margin-top: 0;}
  body {background-color: rgb(255, 246, 250);}
</style>
<script>
  // This function checks whether the user entered a power of 2.

  function checkForPowerOf2(form) {
    var numBox;       // number control
    var output;       // output element that displays the response
    var num;          // user-entered number
    var quotient;     // number that is repeatedly divided by 2
    var wholeNumber; // is the quotient a whole number?
```

FIGURE 10.8B `head` container for Powers of 2 web page

The web page's background color applies to the background parts of the image (the parts surrounding the girl and her juggled numbers) because the image's background uses transparent bits there.

Now let's examine the web page's `script` container with its `checkForPowerOf2` function definition. Figure 10.8B shows trivial stuff—the function heading and the variable declarations. **FIGURE 10.8C** shows the good stuff. The function checks the number box and displays an error message for invalid input. If the user enters valid input, the function uses a `do` loop to repeatedly divide by 2 while the resulting quotient is greater than 1. After the loop, if the final quotient is exactly 1, that means the user entered a power of 2, and the web page displays a congratulatory message.

Note the condition at the bottom of the `do` loop:

```
} while (wholeNumber && quotient > 1);
```

Previously, we said the loop repeats as long as the resulting quotient is greater than 1. Well, almost. As you can see in the `do` loop's condition, there's a second thing that must also be true for the loop to repeat—the `wholeNumber` variable must have a value of true. If you start with a power of 2 and you repeatedly divide by 2, each resulting quotient will be a whole number (e.g., $16 / 2 \Rightarrow 8$, $8 / 2 \Rightarrow 4$, $4 / 2 \Rightarrow 2$, $2 / 2 \Rightarrow 1$). On the other hand, if you start with a number that's not a power of 2 and you repeatedly divide by 2, you'll eventually get a quotient that's not a whole number. So to make the function more efficient, with each loop iteration, you can check the resulting quotient to see if it's a whole number. If it's not a whole number, you can immediately terminate the loop and tell the user that his or her entry was not a power of 2. To keep track of whether the resulting quotient is a whole number, we use the `wholeNumber` variable.

```
    numBox = form.elements["number"];
    output = form.elements["result"];

    if (!numBox.checkValidity()) {
      output.value =
        "Invalid input. You must enter an integer 10 or greater.";
    }
    else {
      num = quotient = numBox.valueAsNumber;
      wholeNumber = true;
      do {
        quotient /= 2;
        if (quotient != Math.floor(quotient)) {
          wholeNumber = false;
        }
      } while (wholeNumber && quotient > 1);

      if (quotient == 1) {
        output.value = "Yes, " + num + " is a power of 2." +
          " You're so awesome!";
      }
      else {
        output.value = "Sorry, " + num + " is not a power of 2.";
      }
    } // end else
  } // end checkForPowerOf2
</script>
</head>
```


FIGURE 10.8C `head` **container for Powers of 2 web page**

Using a Boolean Variable to Terminate the Loop

In the past, we've used variables to store numbers, strings, and objects. The JavaScript language supports those *data types* as well as a few others. The Boolean data type is for variables that hold the value true or the value false, and those variables are referred to as *Boolean variables*. As you might have guessed by now, the `wholeNumber` variable is a Boolean variable. It holds the value true if the most recently generated quotient is a whole number and false otherwise. In the `checkForPowerOf2` function, note how we assign true to `wholeNumber` above the loop and then inside the loop, we assign false to `wholeNumber` if the new quotient is not a whole number. Note how we use `Math.floor` to see if the new quotient is not a whole number:

```
if (quotient != Math.floor(quotient)) {
```

Remember that the `floor` method rounds down, so if the quotient is not a whole number, rounding down returns a value different from the original value. And the `!=` operator evaluates to

true if the values are different. At the bottom of the loop, we use `wholeNumber` in the `do` loop's condition:

```
} while (wholeNumber && quotient > 1);
```

If `wholeNumber` has the value false, then the condition pares down to `false && quotient > 1`. Remember that if you use false with the `&&` operator, the result is false, regardless of the other operand's value.

By the way, we didn't have to use a Boolean variable in the `checkForPowerOf2` function. This `do` loop provides the same functionality without using a Boolean variable:

```
do {
  quotient /= 2;
} while (quotient != Math.floor(quotient) && quotient > 1);
```

So, what's the benefit of using a Boolean variable? It can lead to more readable code, as is the case in the `do` loop condition in the Powers of 2 web page. Readability can be improved even more dramatically in other cases. In general, a Boolean variable can be used to keep track of a situation in which there's a state with one of two possible values. For example, if you're writing a program that plays a game against the computer, you can keep track of the "state" of whose turn it is by using a Boolean variable named `userTurn`. If `userTurn` holds the value true, it's the user's turn. If `userTurn` holds the value false, it's the computer's turn. In an end-of-chapter exercise, you'll be asked to trace a code fragment that uses such a `userTurn` Boolean variable.

In case you were wondering, the term Boolean comes from George Boole, a 19th-century English mathematician. He invented Boolean algebra, which describes operations that can be performed with true and false values.

10.9 `for` Loop

In the previous section, we used a `while` loop to access all the checkboxes in the collection of job skills checkboxes. Using a `while` loop works OK, but in this section, we use a `for` loop to access the checkboxes, which leads to a more compact implementation.

FIGURE 10.13 shows the `while` loop used in the Job Skills web page and a functionally equivalent `for` loop. Both versions use a counter variable, `i`, that gets initialized to 0 and gets incremented each time through the loop. With a `for` loop, the counter mechanism is implemented within the loop's heading. It's such a foundational part of a `for` loop that the counter variable is given a special name—an *index variable*. Sound familiar? Yep, an index variable is also the name we use for the variable inside `[]`'s when referring to an individual object within a collection. So in Figure 10.13's `for` loop body, the `i` in `jobSkillsCBs[i]` is an index variable not only for the `for` loop, but also for the `jobSkillsCBs` collection.

```
var i = 0;
while (i < jobSkillsCBs.length) {
  if (!jobSkillsCBs[i].checkValidity()) {
    message += "<br>" + jobSkillsCBs[i].value;
  }
  i++;
} // end while
```

```
for (let i=0; i<jobSkillsCBs.length; i++) {
  if (!jobSkillsCBs[i].checkValidity()) {
    message += "<br>" + jobSkillsCBs[i].value;
  }
} // end for
```

FIGURE 10.13 **`while` loop versus `for` loop comparison for Job Skills web page**

With a `for` loop, all the looping mechanism code is stuffed into the `for` loop heading. That can make for a rather complicated looking heading when you're new to `for` loops. But for veteran `for` loop users, the `for` loop heading's structure can be comforting because it's compact and it follows a standard format. The `for` loop heading is formed with three components—the initialization, condition, and update components—with the components separated by semicolons:

```
for (initialization; condition; update) {
```

In Figure 10.13, identify those three components in the `for` loop heading. And note the arrows, which show the corresponding component code embedded in the `while` loop. Hopefully, seeing how the `while` loop incorporates the components makes it clear how the components work, but if not, the following list explains how the `for` loop uses the three components:

1. Initialization component

   Before the first pass through the body of the loop, execute the initialization component.
2. Condition component

   Before each loop iteration, evaluate the condition component:
   - If the condition is true, execute the body of the loop.
   - If the condition is false, terminate the loop (exit to the statement below the loop's closing brace).
3. Update component

   After each pass through the body of the loop, return to the loop heading and execute the update component. Then, recheck the continuation condition in the second component, and if it's satisfied, go through the body of the loop again.

Perhaps the hardest part of the `for` loop mechanism to remember is that you have to execute the update component's code after you're done with each loop iteration. It can be hard to remember because the code appears at the top of the loop, even though you execute it after executing the bottom of the loop.

In Figure 10.13, note that we declare the `i` index variable with `let` in the `for` loop heading. When you declare a `for` loop index variable with `let`, that limits the scope of the index variable to just the loop. In other words, whenever a variable is declared in the `for` loop heading, it exists and can be recognized and used only by code that is within the heading or body of that `for` loop. By limiting the index variable's scope, you can redeclare the same-named index variable in a second loop with no fear of one loop's index variable messing up the other loop's index variable. You might think that using `var` instead of `let` for declaring your index variable would accomplish the same thing. Nope. If you declare a variable with `var`, the variable's scope is the entire function. Normally, that won't create problems, but you should do more than strive for acceptable normalcy. You should strive for maximum elegance, and that means using `let` for your index variable declarations.

In the following `for` loop heading (copied from Figure 10.13 for your convenience), note that there are no spaces surrounding the = operator and the < operator:

```
for (let i=0; i<jobSkillsCBs.length; i++) {
```

Why is that good practice? The `for` loop header is inherently complex, so in order to temper that complexity, we add visual cues to compartmentalize the `for` loop header. More specifically, we omit spaces within each of the three sections, and we insert a space after each semicolon to keep the three sections separate.

When to Use Each Type of Loop

Although you can use any of the three loops for any looping task, you should strive to use the type of loop that fits best for your particular task at hand. If you have a task where you know the exact number of loop iterations before the loop begins, use a `for` loop. For the Job Skills web page, the task was to loop through the checkboxes in a checkbox collection. The number of loop iterations came from the checkbox collection's `length` property. We knew that value before executing the loop, so using a `for` loop worked out nicely. Remember the compound interest web page? The task was to repeatedly generate projected interest and balance values for upcoming years. The number of loop iterations came from the user's input in the "Years to grow" text control (see Figure 10.3 for a refresher). We used a `while` loop for that implementation, but because we knew the number of loop iterations before executing the loop, we could have used a `for` loop and the result would have been slightly more compact. On the other hand, what about the Powers of 2 web page? The task was to repeatedly divide by 2 until the quotient became 1 or less than 1. Before the loop began, we did not know how many times the loop would repeat, so using a `for` loop would have been inappropriate. We knew that the loop should be executed at least one time, so we used a `do` loop, and the JavaScript programming gods smiled down upon us and said, "It was good."

`for…of` Loop

Now that you know when to use a `for` loop, let's get fancy and introduce another version of the `for` loop. The `for…of` loop uses a more compact heading than the traditional `for` loop by eliminating the index variable. For example, here's a `for…of` loop that is functionally equivalent to the loop used in the Job Skills web page:

```
for (let skill of jobSkillsCBs) {
  if (!skill.checkValidity()) {
    message += "<br>" + skill.value;
  }
} // end for
```

As promised, you can see that the `for…of` loop uses no index variable. The `for…of` loop is more compact than the traditional `for` loop because there are no initialization, condition, and update parts you need to worry about. With a traditional `for` loop, those parts implement the loop's counting mechanism. With a `for…of` loop, the counting functionality is taken care of automatically behind the scenes without you (the developer) having to do anything.

Before you get too excited about the for…of loop, you need to realize that it's not as general purpose as the standard for loop. The for…of loop works only if you have a collection. Here's the syntax for the for…of loop's heading:

```
for (let variable of collection) {
```

In the Job Skills for…of loop shown here, jobSkillsCBs is the collection and skill is the variable. The skill variable serves as a repository for each object in the jobSkillsCBs collection as the loop traverses through the objects. So in the loop's body, to access a value in the collection, you don't need []'s around an index variable. Instead, you simply use the variable declared in the loop's heading. In the Job Skills web page, to access a checkbox object within the loop, we simply use the skill variable. "Accessing" the checkbox object means you can read its values or update its values.