Lecture 10
JavaScript: Functions

Instructor: Sudad H. Abed
Desgined By: Dabin Ding
UCM

OBJECTIVES

In this chapter you will:

- Construct programs modularly from small pieces called functions.
- Define new functions.
- Pass information between functions.
- Use simulation techniques based on random number generation.
- Use the new HTML5 `audio` and `video` elements
- Use additional global methods.
- See how the visibility of identifiers is limited to specific regions of programs.

9.1 Introduction

- To develop and maintain a large program
  - construct it from small, simple pieces
  - divide and conquer

9.2 Program Modules in JavaScript

- You'll combine new functions that you write with prepackaged functions and objects available in JavaScript
- The prepackaged functions that belong to JavaScript objects (such as `Math.pow`, introduced previously) are called **methods**.
- JavaScript provides several objects that have a rich collection of methods for performing common mathematical calculations, string manipulations, date and time manipulations, and manipulations of collections of data called arrays.

9.2 Program Modules in JavaScript (Cont.)

- You can define programmer-defined functions that perform specific tasks and use them at many points in a script
  - The actual statements defining the function are written only once and are hidden from other functions
- Functions are invoked by writing the name of the function, followed by a left parenthesis, followed by a comma-separated list of zero or more arguments, followed by a right parenthesis
- Methods are called in the same way as functions, but require the name of the object to which the method belongs and a dot preceding the method name
- Function (and method) arguments may be constants, variables or expressions

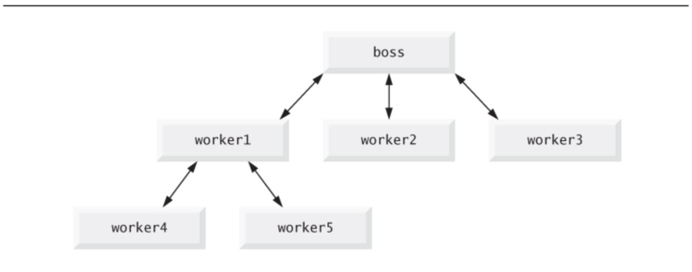

Fig. 9.1 | Hierarchical boss-function/worker-function relationship.

9.3.1 Programmer-Defined Function square

- `return` statement
  - passes information from inside a function back to the point in the program where it was called
- A function must be called explicitly for the code in its body to execute
- The format of a function definition is

function *function-name*(*parameter-list*)
{
    declarations and statements
}

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 9.2: SquareInt.html -->
4  <!-- Programmer-defined function square. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>A Programmer-Defined square Function</title>
9        <style type = "text/css">
10          p { margin: 0; }
11       </style>
12       <script>
13
14          document.writeln( "<h1>Square the numbers from 1 to 10</h1>" );
15
16          // square the numbers from 1 to 10
17          for ( var x = 1; x <= 10; ++x )
18             document.writeln( "<p>The square of " + x + " is " +
19                square( x ) + "</p>" );
20
```

Fig. 9.2 | Programmer-defined function `square`. (Part 1 of 3.)


```
21          // The following square function definition's body is executed
22          // only when the function is called explicitly as in line 19
23          function square( y )
24          {
25             return y * y;
26          } // end function square
27
28       </script>
29    </head><body></body> <!-- empty body element -->
30 </html>
```

Fig. 9.2 | Programmer-defined function `square`. (Part 2 of 3.)

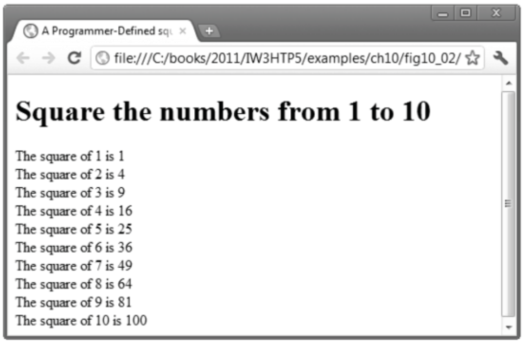

Fig. 9.2 | Programmer-defined function `square`. (Part 3 of 3.)

Common Programming Error 9.1

Forgetting to return a value from a function that's supposed to return a value is a logic error.

9.3.1 Programmer-Defined Function square (cont.)

- Three ways to return control to the point at which a function was invoked
  - Reaching the function-ending right brace
  - Executing the statement `return;`
  - Executing the statement "`return` *`expression`*;" to return the value of *expression* to the caller
- When a `return` statement executes, control returns immediately to the point at which the function was invoked

9.3.2 Programmer-Defined Function maximum (cont.)

- The script in our next example (Fig. 9.3) uses a programmer-defined function called `maximum` to determine and return the largest of three floating-point values.

```
<!DOCTYPE html>

<!-- Fig. 9.3: maximum.html -->
<!-- Programmer-Defined maximum function. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Maximum of Three Values</title>
      <style type = "text/css">
         p { margin: 0; }
      </style>
      <script>

         var input1 = window.prompt( "Enter first number", "0" );
         var input2 = window.prompt( "Enter second number", "0" );
         var input3 = window.prompt( "Enter third number", "0" );

         var value1 = parseFloat( input1 );
         var value2 = parseFloat( input2 );
         var value3 = parseFloat( input3 );

         var maxValue = maximum( value1, value2, value3 );

```

Fig. 9.3 | Programmer-defined `maximum` function. (Part 1 of 4.)


```
         document.writeln( "<p>First number: " + value1 + "</p>" +
            "<p>Second number: " + value2 + "</p>" +
            "<p>Third number: " + value3 + "</p>" +
            "<p>Maximum is: " + maxValue + "</p>" );

         // maximum function definition (called from line 22)
         function maximum( x, y, z )
         {
            return Math.max( x, Math.max( y, z ) );
         } // end function maximum

      </script>
   </head><body></body>
</html>
```

Fig. 9.3 | Programmer-defined `maximum` function. (Part 2 of 4.)

Fig. 9.3 | Programmer-defined `maximum` function. (Part 3 of 4.)

Fig. 9.3 | Programmer-defined `maximum` function. (Part 4 of 4.)

9.4 Notes on Programmer-Defined Functions

- All variables declared with the keyword var in function definitions are local variables—this means that they can be accessed only in the function in which they're defined.
- A function's parameters are also considered to be local variables.
- There are several reasons for modularizing a program with functions.
  - Divide-and-conquer approach makes program development more manageable.
  - Software reusability.
  - Avoid repeating code in a program.

Software Engineering Observation 9.1

If a function's task cannot be expressed concisely, perhaps the function is performing too many different tasks. It's usually best to break such a function into several smaller functions.

Common Programming Error 9.2

Redefining a function parameter as a local variable in the function is a logic error.

Good Programming Practice 9.1

Do not use the same name for an argument passed to a function and the corresponding parameter in the function definition. Using different names avoids ambiguity.

Software Engineering Observation 9.2

To promote software reusability, every function should be limited to performing a single, well-defined task, and the name of the function should describe that task effectively. Such functions make programs easier to write, debug, maintain and modify.

9.5 Random Number Generation

- `random` method generates a floating-point value from 0.0 up to, but *not* including, 1.0
- Random integers in a certain range can be generated by scaling and shifting the values returned by `random`, then using `Math.floor` to convert them to integers
  - The scaling factor determines the size of the range (i.e. a scaling factor of 4 means four possible integers)
  - The shift number is added to the result to determine where the range begins (i.e. shifting the numbers by 3 would give numbers between 3 and 7)
- Method `Math.floor` rounds its argument down to the closest integer

```
<!DOCTYPE html>

<!-- Fig. 9.4: RandomInt.html -->
<!-- Random integers, shifting and scaling. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Shifted and Scaled Random Integers</title>
      <style type = "text/css">
         p, ol { margin: 0; }
         li    { display: inline; margin-right: 10px; }
      </style>
      <script>

         var value;

         document.writeln( "<p>Random Numbers</p><ol>" );

         for ( var i = 1; i <= 30; ++i )
         {
            value = Math.floor( 1 + Math.random() * 6 );
            document.writeln( "<li>" + value + "</li>" );
         } // end for

```

Fig. 9.4 | Random integers, shifting and scaling. (Part 1 of 2.)


```
         document.writeln( "</ol>" );

      </script>
   </head><body></body>
</html>
```

Fig. 9.4 | Random integers, shifting and scaling. (Part 2 of 2.)

9.5.2 Displaying Random Images

- In the next example, we build a **random image generator**—a script that displays four randomly selected die images every time the user clicks a Roll Dice button on the page.
- For the script in Fig. 9.5 to function properly, the directory containing the file `RollDice.html` must also contain the six die images—these are included with this chapter's examples.


```
<!DOCTYPE html>

<!-- Fig. 9.5: RollDice.html -->
<!-- Random dice image generation using Math.random. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Random Dice Images</title>
      <style type = "text/css">
         li { display: inline; margin-right: 10px; }
         ul { margin: 0; }
      </style>
      <script>
         // variables used to interact with the i mg elements
         var die1Image;
         var die2Image;
         var die3Image;
         var die4Image;

```

Fig. 9.5 | Random dice image generation using `Math.random`. (Part 1 of 4.)

```
        // register button listener and get the img elements
        function start()
        {
           var button = document.getElementById( "rollButton" );
           button.addEventListener( "click", rollDice, false  );
           die1Image = document.getElementById( "die1" );
           die2Image = document.getElementById( "die2" );
           die3Image = document.getElementById( "die3" );
           die4Image = document.getElementById( "die4" );
        } // end function rollDice

        // roll the dice
        function rollDice()
        {
           setImage( die1Image );
           setImage( die2Image );
           setImage( die3Image );
           setImage( die4Image );
        } // end function rollDice

```

Fig. 9.5 | Random dice image generation using `Math.random`. (Part 2 of 4.)


```
        // set image source for a die
        function setImage( dieImg )
        {
           var dieValue = Math.floor( 1 + Math.random() * 6 );
           dieImg.setAttribute( "src", "die" + dieValue + ".png" );
           dieImg.setAttribute( "alt",
              "die image with " + dieValue + " spot(s)" );
        } // end function setImage

        window.addEventListener( "load", start, false );
      </script>
   </head>
```

Fig. 9.5 | Random dice image generation using `Math.random`. (Part 3 of 4.)

```
    <body>
       <form action = "#">
          <input id = "rollButton" type = "button" value = "Roll Dice">
       </form>
       <ol>
          <li><img id = "die1" src = "blank.png" alt = "die 1 image"></li>
          <li><img id = "die2" src = "blank.png" alt = "die 2 image"></li>
          <li><img id = "die3" src = "blank.png" alt = "die 3 image"></li>
          <li><img id = "die4" src = "blank.png" alt = "die 4 image"></li>
       </ol>
    </body>
</html>
```

Fig. 9.5 | Random dice image generation using `Math.random`. (Part 4 of 4.)

9.5.2 Displaying Random Images

User Interactions Via Event Handling

- Until now, all user interactions with scripts have been through
  - a prompt dialog or
  - an alert dialog.
- These dialogs are valid ways to receive input from a user and to display messages, but they're fairly limited in their capabilities.
- A prompt dialog can obtain only one value at a time from the user, and a message dialog can display only one message.
- Inputs are typically received from the user via an HTML5 form.
- Outputs are typically displayed to the user in the web page.
- This program uses an HTML5 form and a new graphical user interface concept—GUI **event handling**.
- The JavaScript executes in response to the user's interaction with an element in a form. This interaction causes an event.
- Scripts are often used to respond to user initiated events.

9.5.2 Displaying Random Images

The body Element

- The elements in the body are used extensively in the script.

The form Element

- The HTML5 standard requires that every form contain an action attribute, but because this form does not post its information to a web server, the string "#" is used simply to allow this document to validate.
- The # symbol by itself represents the current page.

9.5.2 Displaying Random Images

The button input Element and Event-Driven Programming

- **Event-driven programming**
  - the user interacts with an element in the web page, the script is notified of the event and the script processes the event.
- The user's interaction with the GUI "drives" the program.
- The button click is known as the **event**.
- The function that's called when an event occurs is known as an **event handler**.
- When a GUI event occurs in a form, the browser calls the specified event-handling function.
- Before any event can be processed, each element must know which event-handling function will be called when a particular event occurs.

9.5.2 Displaying Random Images

The img Elements

- The four `img` elements will display the four randomly selected dice.
- Their `id` attributes (`die1`, `die2`, `die3` and `die4`, respectively) can be used to apply CSS styles and to enable script code to refer to these element in the HTML5 document.
- Because the `id` attribute, if specified, must have a unique value among all id attributes in the page, Java-Script can reliably refer to any single element via its `id` attribute.
- Each `img` element displays the image `blank.png` (an empty white image) when the page first renders.

9.5.2 Displaying Random Images

Specifying a Function to Call When the Browser Finishes Loading a Document

- Many examples will execute a JavaScript function when the document finishes loading.
- This is accomplished by handling the window object's load event.
- To specify the function to call when an event occurs, you registering an event handler for that event.
- Method `addEventListener` is available for every DOM node. The method takes three arguments:
  - the first is the name of the event for which we're registering a handler
  - the second is the function that will be called to handle the event
  - the last argument is typically false—the true value is beyond this book's scope

9.5.3 Rolling Dice Repeatedly and Displaying Statistics

- To show that the random values representing the dice occur with approximately equal likelihood, let's allow the user to roll 12 dice at a time and keep statistics showing the number of times each face occurs and the percentage of the time each face is rolled (Fig. 9.6).


```
<!DOCTYPE html>

<!-- Fig. 9.6: RollDice.html -->
<!-- Rolling 12 dice and displaying frequencies. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Die Rolling Frequencies</title>
      <style type = "text/css">
         img           { margin-right: 10px; }
         table         { width: 200px;
                         border-collapse: collapse;
                         background-color: lightblue; }
         table, td, th { border: 1px solid black;
                         padding: 4px;
                         margin-top: 20px; }
         th            { text-align: left;
                         color: white;
                         background-color: darkblue; }
      </style>
```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 1 of 9.)

```
      <script>
         var frequency1 = 0;
         var frequency2 = 0;
         var frequency3 = 0;
         var frequency4 = 0;
         var frequency5 = 0;
         var frequency6 = 0;
         var totalDice = 0;

         // register button event handler
         function start()
         {
            var button = document.getElementById( "rollButton" );
            button.addEventListener( "click", rollDice, false );
         } // end function start

```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 2 of 9.)


```
         // roll the dice
         function rollDice()
         {
            var face;  // face rolled

            // loop to roll die 12 times
            for ( var i = 1; i <= 12; ++i )
            {
               face = Math.floor( 1 + Math.random() * 6 );
               tallyRolls( face ); // increment a frequency counter
               setImage( i, face ); // display appropriate die image
               ++totalDice; // increment total
            } // end die rolling loop

            updateFrequencyTable();
         } // end function rollDice

```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 3 of 9.)

©1992-2012 by Pearson Education, Inc. All Rights Reserved.

```
// increment appropriate frequency counter
function tallyRolls( face )
{
   switch ( face )
   {
      case 1:
         ++frequency1;
         break;
      case 2:
         ++frequency2;
         break;
      case 3:
         ++frequency3;
         break;
      case 4:
         ++frequency4;
         break;
      case 5:
         ++frequency5;
         break;
      case 6:
         ++frequency6;
         break;
   } // end switch
} // end function tallyRolls
```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 4 of 9.)


```

// set image source for a die
function setImage( dieNumber, face )
{
   var dieImg = document.getElementById( "die" + dieNumber );
   dieImg.setAttribute( "src", "die" + face + ".png" );
   dieImg.setAttribute( "alt", "die with " + face + " spot(s)" );
} // end function setImage

```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 5 of 9.)

```
      // update frequency table in the page
      function updateFrequencyTable()
      {
         var tableDiv = document.getElementById( "frequencyTableDiv" );

         tableDiv.innerHTML = "<table>" +
            "<caption>Die Rolling Frequencies</caption>" +
            "<thead><th>Face</th><th>Frequency</th>" +
            "<th>Percent</th></thead>" +
            "<tbody><tr><td>1</td><td>" + frequency1 + "</td><td>" +
            formatPercent(frequency1 / totalDice) + "</td></tr>" +
            "<tr><td>2</td><td>" + frequency2 + "</td><td>" +
            formatPercent(frequency2 / totalDice)+ "</td></tr>" +
            "<tr><td>3</td><td>" + frequency3 + "</td><td>" +
            formatPercent(frequency3 / totalDice) + "</td></tr>" +
            "<tr><td>4</td><td>" + frequency4 + "</td><td>" +
            formatPercent(frequency4 / totalDice) + "</td></tr>" +
            "<tr><td>5</td><td>" + frequency5 + "</td><td>" +
            formatPercent(frequency5 / totalDice) + "</td></tr>" +
            "<tr><td>6</td><td>" + frequency6 + "</td><td>" +
            formatPercent(frequency6 / totalDice) + "</td></tr>" +
            "</tbody></table>";
      } // end function updateFrequencyTable

```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 6 of 9.)


```
      // format percentage
      function formatPercent( value )
      {
         value *= 100;
         return value.toFixed(2);
      } // end function formatPercent

      window.addEventListener( "load", start, false );
   </script>
</head>
```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 7 of 9.)

```
    <body>
       <p><img id = "die1" src = "blank.png" alt = "die 1 image">
          <img id = "die2" src = "blank.png" alt = "die 2 image">
          <img id = "die3" src = "blank.png" alt = "die 3 image">
          <img id = "die4" src = "blank.png" alt = "die 4 image">
          <img id = "die5" src = "blank.png" alt = "die 5 image">
          <img id = "die6" src = "blank.png" alt = "die 6 image"></p>
       <p><img id = "die7" src = "blank.png" alt = "die 7 image">
          <img id = "die8" src = "blank.png" alt = "die 8 image">
          <img id = "die9" src = "blank.png" alt = "die 9 image">
          <img id = "die10" src = "blank.png" alt = "die 10 image">
          <img id = "die11" src = "blank.png" alt = "die 11 image">
          <img id = "die12" src = "blank.png" alt = "die 12 image"></p>
       <form action = "#">
          <input id = "rollButton" type = "button" value = "Roll Dice">
       </form>
       <div id = "frequencyTableDiv"></div>
    </body>
 </html>
```

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 8 of 9.)

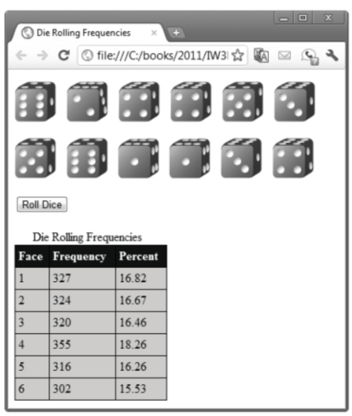

Fig. 9.6 | Rolling 12 dice and displaying frequencies. (Part 9 of 9.)

9.5.3 Rolling Dice Repeatedly and Displaying Statistics

Generalized Scaling and Shifting of Random Values

- The values returned by random are always in the range

  0.0 £ Math.random() < 1.0

- Previously, we demonstrated the statement

  face = Math.floor(1 + Math.random() * 6);

- which simulates the rolling of a six-sided die. This statement always assigns an integer (at random) to variable face, in the range 1 £ face £ 6.
- Referring to the preceding statement, we see that the width of the range is determined by the number used to scale random with the multiplication operator (6 in the preceding statement) and that the starting number of the range is equal to the number (1 in the preceding statement) added to Math.random() * 6.

9.5.3 Rolling Dice Repeatedly and Displaying Statistics (cont.)

- We can generalize this result as

  face = Math.floor(a + Math.random() * b);

- where a is the shifting value (which is equal to the first number in the desired range of consecutive integers) and b is the scaling factor (which is equal to the width of the desired range of consecutive integers).

©1992-2012 by Pearson Education, Inc.
All Rights Reserved.

9.6 Example: Game of Chance; Introducing the HTML5 audio and video Elements

- The script in Fig. 9.7 simulates the game of craps.


```
<!DOCTYPE html>

<!-- Fig. 9.7: Craps.html -->
<!-- Craps game simulation. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Craps Game Simulation</title>
      <style type = "text/css">
         p.red  { color: red }
         img    { width: 54px; height: 54px; }
         div    { border: 5px ridge royalblue;
                  padding: 10px; width: 120px;
                  margin-bottom: 10px; }
         .point { margin: 0px; }
      </style>
```

Fig. 9.7 | Craps game simulation. (Part 1 of 12.)

```
      <script>
         // variables used to refer to page elements
         var pointDie1Img; // refers to first die point img
         var pointDie2Img; // refers to second die point img
         var rollDie1Img; // refers to first die roll img
         var rollDie2Img; // refers to second die roll img
         var messages; // refers to "messages" paragraph
         var playButton; // refers to Play button
         var rollButton; // refers to Roll button
         var dicerolling; // refers to audio clip for dice

         // other variables used in program
         var myPoint; // point if no win/loss on first roll
         var die1Value; // value of first die in current roll
         var die2Value; // value of second die in current roll

```

Fig. 9.7 | Craps game simulation. (Part 2 of 12.)


```
         // starts a new game
         function startGame()
         {
            // get the page elements that we'll interact with
            dicerolling = document.getElementById( "dicerolling" );
            pointDie1Img = document.getElementById( "pointDie1" );
            pointDie2Img = document.getElementById( "pointDie2" );
            rollDie1Img = document.getElementById( "rollDie1" );
            rollDie2Img = document.getElementById( "rollDie2" );
            messages = document.getElementById( "messages" );
            playButton = document.getElementById( "play" );
            rollButton = document.getElementById( "roll" );

            // prepare the GUI
            rollButton.disabled = true; // disable rollButton
            setImage( pointDie1Img ); // reset image for new game
            setImage( pointDie2Img ); // reset image for new game
            setImage( rollDie1Img ); // reset image for new game
            setImage( rollDie2Img ); // reset image for new game

            myPoint = 0; // there is currently no point
            firstRoll(); // roll the dice to start the game
         } // end function startGame

```

Fig. 9.7 | Craps game simulation. (Part 3 of 12.)

```
         // perform first roll of the game
         function firstRoll()
         {
            var sumOfDice = rollDice(); // first roll of the dice

            // determine if the user won, lost or must continue rolling
            switch (sumOfDice)
            {
               case 7: case 11: // win on first roll
                  messages.innerHTML =
                     "You Win!!! Click Play to play again.";
                  break;
               case 2: case 3: case 12: // lose on first roll
                  messages.innerHTML =
                     "Sorry. You Lose. Click Play to play again.";
                  break;
               default: // remember point
                  myPoint = sumOfDice;
                  setImage( pointDie1Img, die1Value );
                  setImage( pointDie2Img, die2Value );
                  messages.innerHTML = "Roll Again!";
                  rollButton.disabled = false; // enable rollButton
                  playButton.disabled = true; // disable playButton
                  break;
            } // end switch
         } // end function firstRoll
```

Fig. 9.7 | Craps game simulation. (Part 4 of 12.)


```

         // called for subsequent rolls of the dice
         function rollAgain()
         {
            var sumOfDice = rollDice(); // subsequent roll of the dice

            if (sumOfDice == myPoint)
            {
               messages.innerHTML =
                  "You Win!!! Click Play to play again.";
               rollButton.disabled = true; // disable rollButton
               playButton.disabled = false; // enable playButton
            } // end if
            else if (sumOfDice == 7) // craps
            {
               messages.innerHTML =
                  "Sorry. You Lose. Click Play to play again.";
               rollButton.disabled = true; // disable rollButton
               playButton.disabled = false; // enable playButton
            } // end else if
         } // end function rollAgain

```

Fig. 9.7 | Craps game simulation. (Part 5 of 12.)

```
// roll the dice
function rollDice()
{
   dicerolling.play(); // play dice rolling sound

   // clear old die images while rolling sound plays
   die1Value = NaN;
   die2Value = NaN;
   showDice();

   die1Value = Math.floor(1 + Math.random() * 6);
   die2Value = Math.floor(1 + Math.random() * 6);
   return die1Value + die2Value;
} // end function rollDice

```

Fig. 9.7 | Craps game simulation. (Part 6 of 12.)


```
// display rolled dice
function showDice()
{
   setImage( rollDie1Img, die1Value );
   setImage( rollDie2Img, die2Value );
} // end function showDice

// set image source for a die
function setImage( dieImg, dieValue )
{
   if ( isFinite( dieValue ) )
      dieImg.src = "die" + dieValue + ".png";
   else
      dieImg.src = "blank.png";
} // end function setImage

```

Fig. 9.7 | Craps game simulation. (Part 7 of 12.)

```
         // register event liseners
         function start()
         {
            var playButton = document.getElementById( "play" );
            playButton.addEventListener( "click", startGame, false );
            var rollButton = document.getElementById( "roll" );
            rollButton.addEventListener( "click", rollAgain, false );
            var diceSound = document.getElementById( "dicerolling" );
            diceSound.addEventListener( "ended", showDice, false );
         } // end function start

         window.addEventListener( "load", start, false );
      </script>
   </head>
   <body>
      <audio id = "dicerolling" preload = "auto">
         <source src = "http://test.deitel.com/dicerolling.mp3"
            type = "audio/mpeg">
         <source src = "http://test.deitel.com/dicerolling.ogg"
            type = "audio/ogg">
         Browser does not support audio tag</audio>
```

Fig. 9.7 | Craps game simulation. (Part 8 of 12.)


```
      <p><a href = "CrapsRules.html">Click here for a short video
         explaining the basic Craps rules</a></p>
      <div id = "pointDiv">
         <p class = "point">Point is:</p>
         <img id = "pointDie1" src = "blank.png"
            alt = "Die 1 of Point Value">
         <img id = "pointDie2" src = "blank.png"
            alt = "Die 2 of Point Value">
      </div>
      <div class = "rollDiv">
         <img id = "rollDie1" src = "blank.png"
            alt = "Die 1 of Roll Value">
         <img id = "rollDie2" src = "blank.png"
            alt = "Die 2 of Roll Value">
      </div>
      <form action = "#">
         <input id = "play" type = "button" value = "Play">
         <input id = "roll" type = "button" value = "Roll">
      </form>
      <p id = "messages" class = "red">Click Play to start the game</p>
   </body>
</html>
```

Fig. 9.7 | Craps game simulation. (Part 9 of 12.)

a) Win on the first roll. In this case, the `pointDiv` does not show any dice and the **Roll** button

b) Loss on the first roll. In this case, the `pointDiv` does not show any dice and the **Roll** button

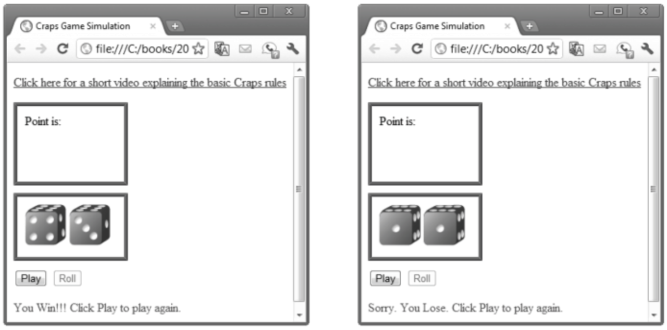

Fig. 9.7 | Craps game simulation. (Part 10 of 12.)

c) First roll is a 5, so the user's point is 5. The **Play** button is disabled and the **Roll** button is enabled.

d) User won on a subsequent roll. The **Play** button is enabled and the **Roll** button is disabled.

Fig. 9.7 | Craps game simulation. (Part 11 of 12.)

e) First roll is a 6, so the user's point is 6. The **Play** button is disabled and the **Roll** button is enabled.

f) User lost on a subsequent roll. The **Play** button is enabled and the **Roll** button is disabled.

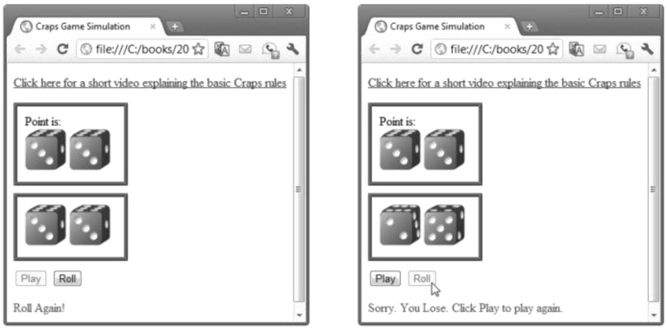

Fig. 9.7 | Craps game simulation. (Part 12 of 12.)

9.6 Example: Game of Chance; Introducing the HTML5 audio and video Elements (Cont.)

The HTML5 audio Element

- An HTML5 audio element is used to embed audio into a web page.
- We specify an `id` for the element, so that we can programmatically control when the audio clip plays, based on the user's interactions with the game.
- Setting the preload attribute to `"auto"` indicates to the browser that it should consider downloading the audio clip so that it's ready to be played when the game needs it.

9.6 Example: Game of Chance; Introducing the HTML5 audio and video Elements (Cont.)

- Most browsers support MP3, OGG and/or WAV format.
- Each source element specifies a src and a type attribute.
  - The src attribute specifies the location of the audio clip.
  - The type attribute specifies the clip’s MIME type—audio/mpeg for the MP3 clip and audio/ogg for the OGG clip (WAV would be audio/x-wav; MIME types for these and other formats can be found online).
- When a web browser that supports the audio element encounters the source elements, it will chose the first audio source that represents one of the browser’s supported formats.

Software Engineering Observation 9.3

Variables declared inside the body of a function are known only in that function. If the same variable names are used elsewhere in the program, they’ll be entirely separate variables in memory.

Error-Prevention Tip 9.1

Initializing variables when they're declared in functions helps avoid incorrect results and interpreter messages warning of uninitialized data.

9.6 Example: Game of Chance; Introducing the HTML5 audio and video Elements (Cont.)

CrapsRules.html and the HMTL5 video Element

- When the user clicks the hyperlink in `Craps.html`, the `CrapsRules.html` is displayed in the browser.
- This page consists of a link back to Craps.html (Fig. 9.8) and an HTML5 video element that displays a video explaining the basic rules for the game of Craps.

```
<!DOCTYPE html>

<!-- Fig. 9.8: CrapsRules.html -->
<!-- Web page with a video of the basic rules for the dice game Craps. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Craps Rules</title>
   </head>
   <body>
      <p><a href = "Craps.html">Back to Craps Game</a></p>
      <video controls>
         <source src = "CrapsRules.mp4" type = "video/mp4">
         <source src = "CrapsRules.webm" type = "video/webm">
         A player rolls two dice. Each die has six faces that contain
         one, two, three, four, five and six spots, respectively. The
```

Fig. 9.8 | Web page that displays a video of the basic rules for the dice game Craps. (Part 1 of 3.)


```
         sum of the spots on the two upward faces is calculated. If the
         sum is 7 or 11 on the first throw, the player wins. If the sum
         is 2, 3 or 12 on the first throw (called "craps"), the player
         loses (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or
         10 on the first throw, that sum becomes the player's "point."
         To win, you must continue rolling the dice until you "make your
         point" (i.e., roll your point value). You lose by rolling a 7
         before making the point.
      </video>
   </body>
</html>
```

Fig. 9.8 | Web page that displays a video of the basic rules for the dice game Craps. (Part 2 of 3.)

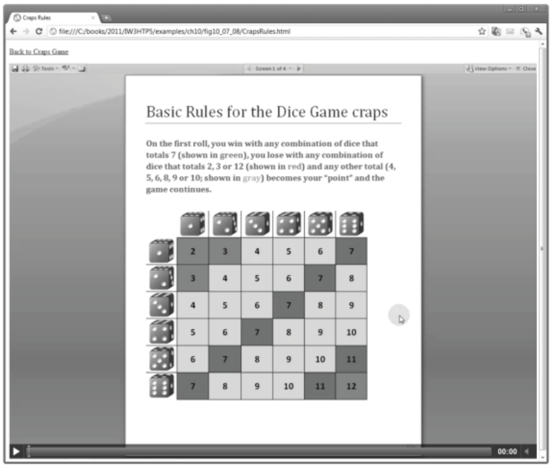

Fig. 9.8 | Web page that displays a video of the basic rules for the dice game Craps. (Part 3 of 3.)

9.7 Scope Rules

- Each identifier in a program has a scope
- The **scope** of an identifier for a variable or function is the portion of the program in which the identifier can be referenced
- **Global variables** or **script-level variables** are accessible in any part of a script and are said to have **global scope**
  - Thus every function in the script can potentially use the variables

9.7 Scope Rules (Cont.)

- Identifiers declared inside a function have function (or local) scope and can be used only in that function
- Function scope begins with the opening left brace ({) of the function in which the identifier is declared and ends at the terminating right brace (})
- Local variables of a function and function parameters have function scope
- If a local variable in a function has the same name as a global variable, the global variable is "hidden" from the body of the function.

Good Programming Practice 9.2

Avoid local-variable names that hide global-variable names. This can be accomplished by simply avoiding the use of duplicate identifiers in a script.

```
<!DOCTYPE html>

<!-- Fig. 9.9: scoping.html -->
<!-- Scoping example. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Scoping Example</title>
      <style type = "text/css">
         p       { margin: 0px; }
         p.space { margin-top: 10px; }
      </style>
```

Fig. 9.9 | Scoping example. (Part 1 of 4.)


```
      <script>
         var output; // stores the string to display
         var x = 1; // global variable

         function start()
         {
            var x = 5; // variable local to function start

            output = "<p>local x in start is " + x + "</p>";

            functionA(); // functionA has local x
            functionB(); // functionB uses global variable x
            functionA(); // functionA reinitializes local x
            functionB(); // global variable x retains its value

            output += "<p class='space'>local x in start is " + x +
               "</p>";
            document.getElementById( "results" ).innerHTML = output;
         } // end function start

```

Fig. 9.9 | Scoping example. (Part 2 of 4.)

```
        function functionA()
        {
           var x = 25; // initialized each time functionA is called

           output += "<p class='space'>local x in functionA is " + x +
              " after entering functionA</p>";
           ++x;
           output += "<p>local x in functionA is " + x +
              " before exiting functionA</p>";
        } // end functionA

        function functionB()
        {
           output += "<p class='space'>global variable x is " + x +
              " on entering functionB";
           x *= 10;
           output += "<p>global variable x is " + x +
              " on exiting functionB</p>";
        } // end functionB

        window.addEventListener( "load", start, false );
     </script>
  </head>
```

Fig. 9.9 | Scoping example. (Part 3 of 4.)


```
  <body>
     <div id = "results"></div>
  </body>
</html>
```

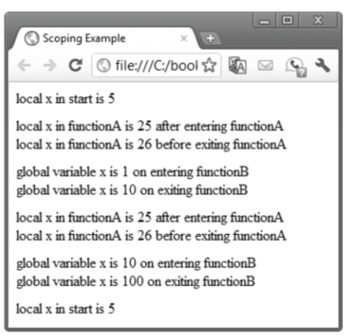

Fig. 9.9 | Scoping example. (Part 4 of 4.)

9.8 JavaScript Global Functions

- JavaScript provides nine global functions as part of a `Global` object
- This object contains
  - all the global variables in the script
  - all the user-defined functions in the script
  - all the built-in global functions listed in the following slide
- You do not need to use the `Global` object directly; JavaScript uses it for you

Global function	Description
`isFinite`	Takes a numeric argument and returns `true` if the value of the argument is not `NaN`, `Number.POSITIVE_INFINITY` or `Number.NEGATIVE_INFINITY` (values that are not numbers or numbers outside the range that JavaScript supports)—otherwise, the function returns `false`.
`isNaN`	Takes a numeric argument and returns `true` if the value of the argument is not a number; otherwise, it returns `false`. The function is commonly used with the return value of `parseInt` or `parseFloat` to determine whether the result is a proper numeric value.
`parseFloat`	Takes a string argument and attempts to convert the *beginning* of the string into a floating-point value. If the conversion is unsuccessful, the function returns `NaN`; otherwise, it returns the converted value (e.g., `parseFloat( "abc123.45" )` returns `NaN`, and `parseFloat( "123.45abc" )` returns the value `123.45`).

Fig. 9.10 | JavaScript global functions. (Part 1 of 2.)

Global function	Description
`parseInt`	Takes a string argument and attempts to convert the beginning of the string into an integer value. If the conversion is unsuccessful, the function returns `NaN`; otherwise, it returns the converted value (for example, `parseInt( "abc123" )` returns `NaN`, and `parseInt( "123abc" )` returns the integer value `123`). This function takes an optional second argument, from 2 to 36, specifying the radix (or base) of the number. Base 2 indicates that the first argument string is in binary format, base 8 that it's in octal format and base 16 that it's in hexadecimal format. See Appendix E, for more information on binary, octal and hexadecimal numbers.

Fig. 9.10 | JavaScript global functions. (Part 2 of 2.)

9.9 Recursion

- A recursive function calls itself, either directly, or indirectly through another function.
- A recursive function knows how to solve only the simplest case, or base case
  - If the function is called with a base case, it returns a result
  - If the function is called with a more complex problem, it divides the problem into two conceptual pieces—a piece that the function knows how to process (the base case) and a simpler or smaller version of the original problem.
- The function invokes (calls) a fresh copy of itself to go to work on the smaller problem; this invocation is referred to as a recursive call, or the recursion step.

9.10 Recursion (Cont.)

- The recursion step executes while the original call to the function is still open (i.e., it has not finished executing)
- For recursion eventually to terminate, each time the function calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case
  - At that point, the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues up the line until the original function call eventually returns the final result to the caller

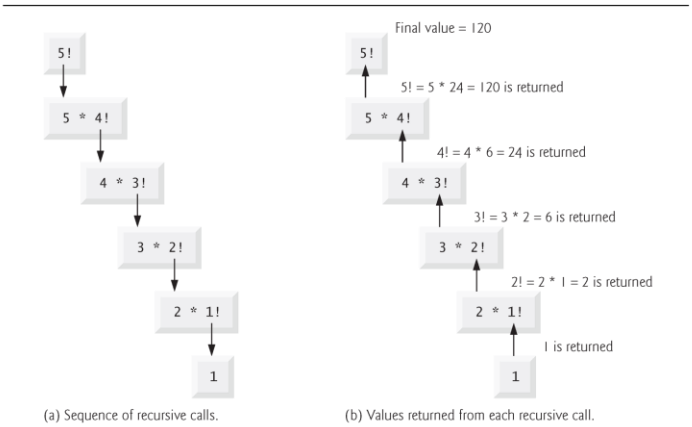

Fig. 9.11 | Recursive evaluation of 5!.

```
<!DOCTYPE html>

<!-- Fig. 9.12: FactorialTest.html -->
<!-- Factorial calculation with a recursive function. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Recursive Factorial Function</title>
      <style type = "text/css">
         p       { margin: 0px; }
      </style>
      <script>
         var output = ""; // stores the output

         // calculates factorials of 0 - 10
         function calculateFactorials()
         {
            for ( var i = 0; i <= 10; ++i )
               output += "<p>" + i + "! = " + factorial( i ) + "</p>";

            document.getElementById( "results" ).innerHTML = output;
         } // end function calculateFactorials

```

Fig. 9.12 | Factorial calculation with a recursive function. (Part 1 of 3.)


```
         // Recursive definition of function factorial
         function factorial( number )
         {
            if ( number <= 1 )  // base case
               return 1;
            else
               return number * factorial( number - 1 );
         } // end function factorial

         window.addEventListener( "load", calculateFactorials, false );
      </script>
   </head>
   <body>
      <h1>Factorials of 0 to 10</h1>
      <div id = "results"></div>
   </body>
</html>
```

Fig. 9.12 | Factorial calculation with a recursive function. (Part 2 of 3.)

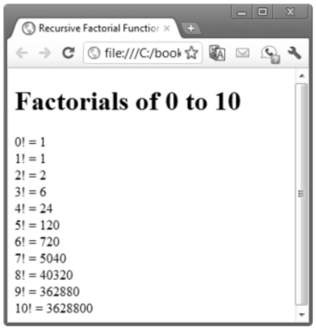

Fig. 9.12 | Factorial calculation with a recursive function. (Part 3 of 3.)

Common Programming Error 9.3

Omitting the base case and writing the recursion step incorrectly so that it does not converge on the base case are both errors that cause infinite recursion, eventually exhausting memory. This situation is analogous to the problem of an infinite loop in an iterative (non-recursive) solution.

©1992-2012 by Pearson Education, Inc. All Rights Reserved.

Error-Prevention Tip 9.2

Internet Explorer displays an error message when a script seems to be going into infinite recursion. Firefox simply terminates the script after detecting the problem. This allows the user of the web page to recover from a script that contains an infinite loop or infinite recursion.

9.10 Recursion vs. Iteration

- Both iteration and recursion involve repetition
  - Iteration explicitly uses a repetition statement
  - Recursion achieves repetition through repeated function calls
- Iteration and recursion each involve a termination test
  - Iteration terminates when the loop-continuation condition fails
  - Recursion terminates when a base case is recognized

9.11 Recursion vs. Iteration

- Iteration both with counter-controlled repetition and recursion gradually approach termination
  - Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail
  - Recursion keeps producing simpler versions of the original problem until the base case is reached
- Both iteration and recursion can occur infinitely:
  - An infinite loop occurs with iteration if the loop-continuation test never becomes false;
  - infinite recursion occurs if the recursion step does not reduce the problem each time via a sequence that converges on the base case or if the base case is incorrect.

Software Engineering Observation 9.4

Any problem that can be solved recursively can also be solved iteratively (non-recursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

Performance Tip 9.1

Avoid using recursion in performance-critical situations. Recursive calls take time and consume additional memory.