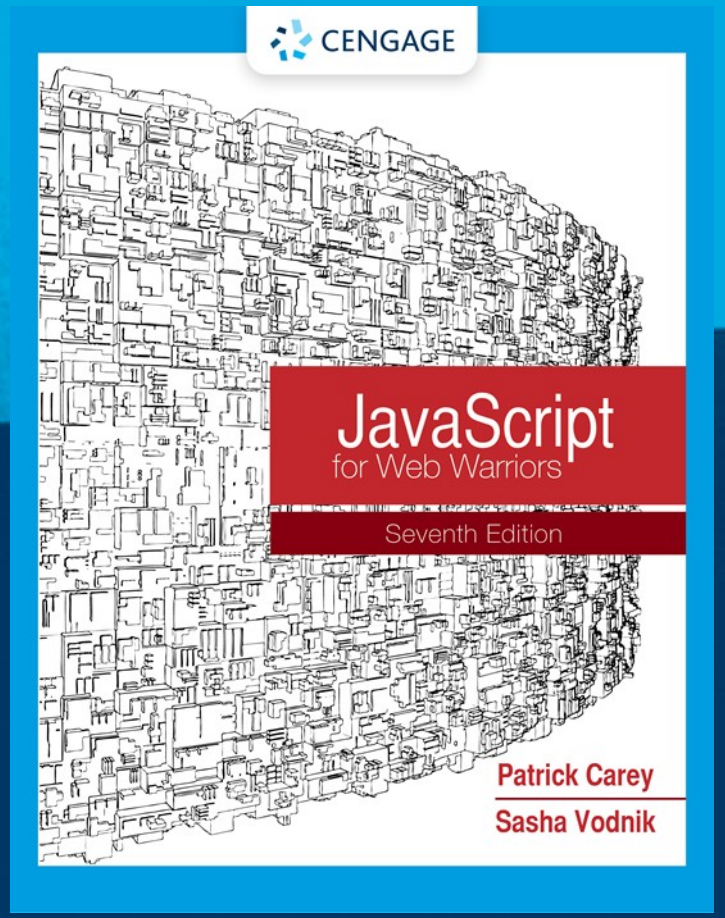



# JavaScript for Web Warriors, 7e

# Chapter 2: Working with Functions, Data Types, and Operators

CENGAGE

# Chapter Objectives (1 of 2)

By the end of this chapter, you should be able to:

- Write and call functions to perform actions and calculate values.
- Associate functions with events using event handlers and event listeners.
- Use built-in JavaScript functions.
- Understand the scope of variables and functions.
- Understand the data types supported by JavaScript and write expressions with numeric values, text strings, and Boolean values.

# Chapter Objectives (2 of 2)

By the end of this chapter, you should be able to:

- Create expressions using arithmetic, assignment, comparison, logical, string, and special operators.
- Understand order precedence and associativity of operations.
- Work with events and values associated with form controls.
- Access your browser's debugging console.

# Working with Functions (1 of 4)

- **Function:** a programming structure consisting of a collection of statements that share a common purpose or calculate a value
- Defining a function
  - Syntax for a **named function**:

```
function functionName(parameters) {
    statements
}
```

  - Syntax for an **anonymous function**:

```
function (parameters) {
    statements
}
```

  - Function's **parameters** are the variables it uses
  - Enclosed in a **command block** (opening and closing curly braces)

# Working with Functions (2 of 4)

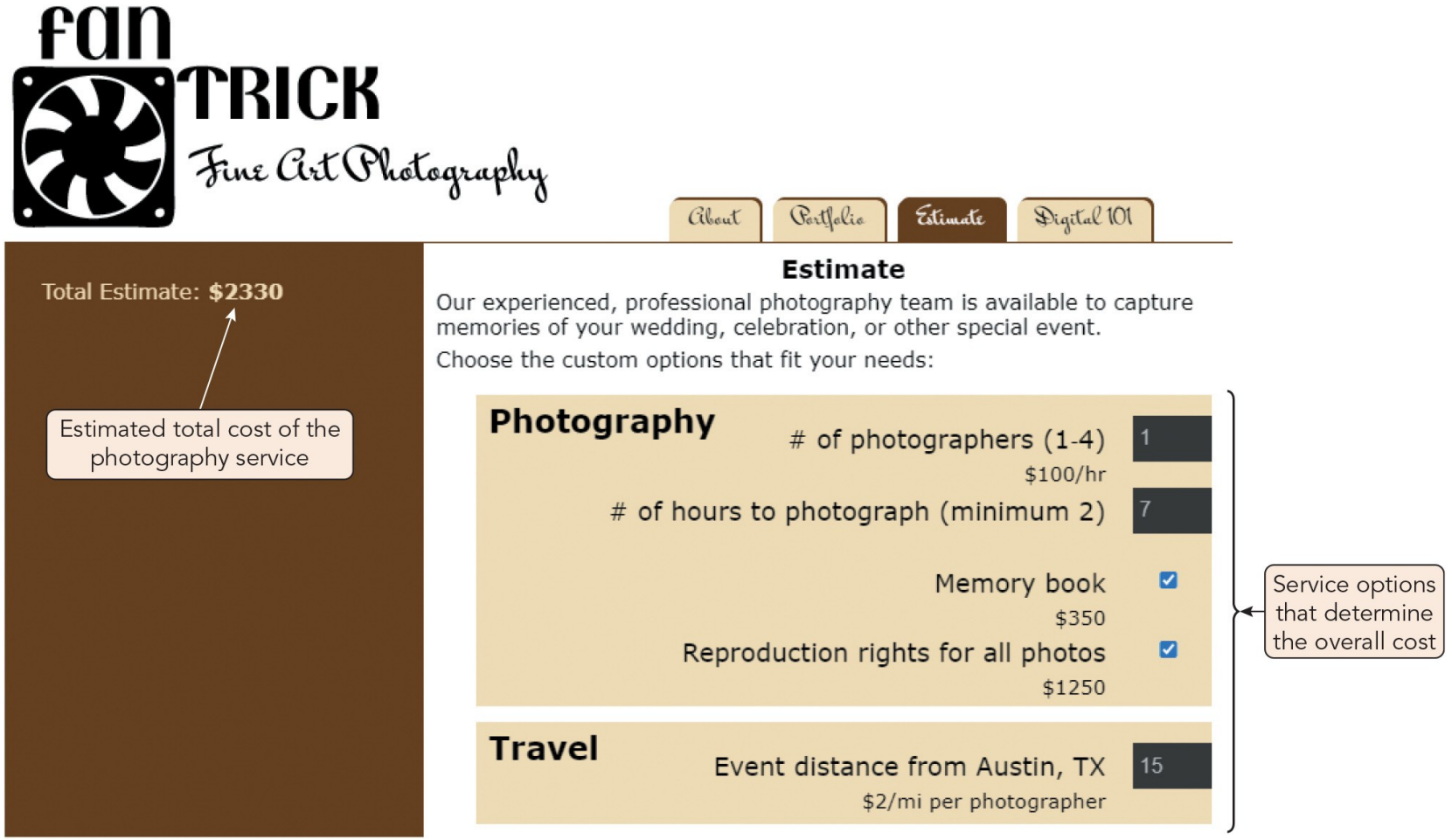



Figure 2-1 Estimating the total photography charge

# Working with Functions (3 of 4)

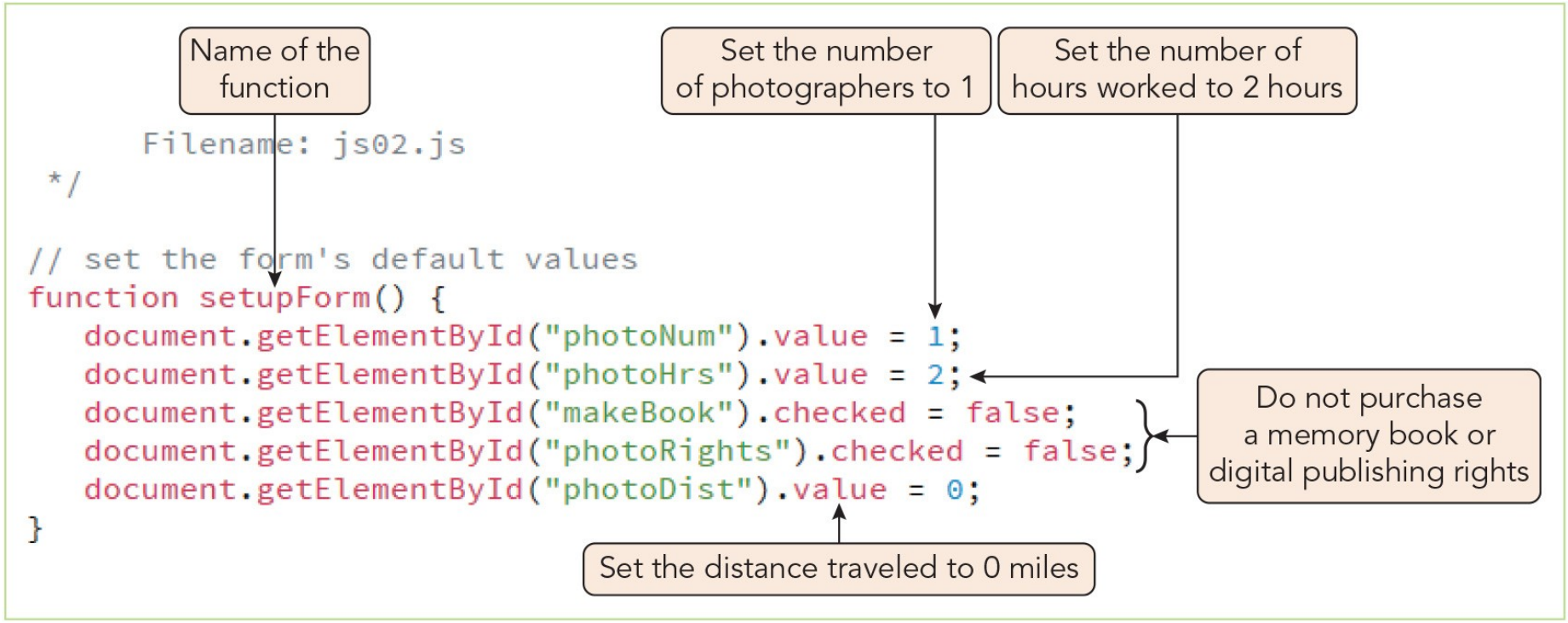



Figure 2-2 Creating the `setupForm()` function

# Working with Functions (4 of 4)

- Calling a function
  - JavaScript expression for **calling** a function:
    ```
    functionName(paramValues);
    ```
  - paramValues passed to a function are the **arguments** (**actual parameters**)
- Returning a value from a function
  - Syntax for a function that returns a value:
    ```
    function functionName(parameters) {
        statements
        return value;
    }
    ```
  - `return` statement ends execution and returns a single value

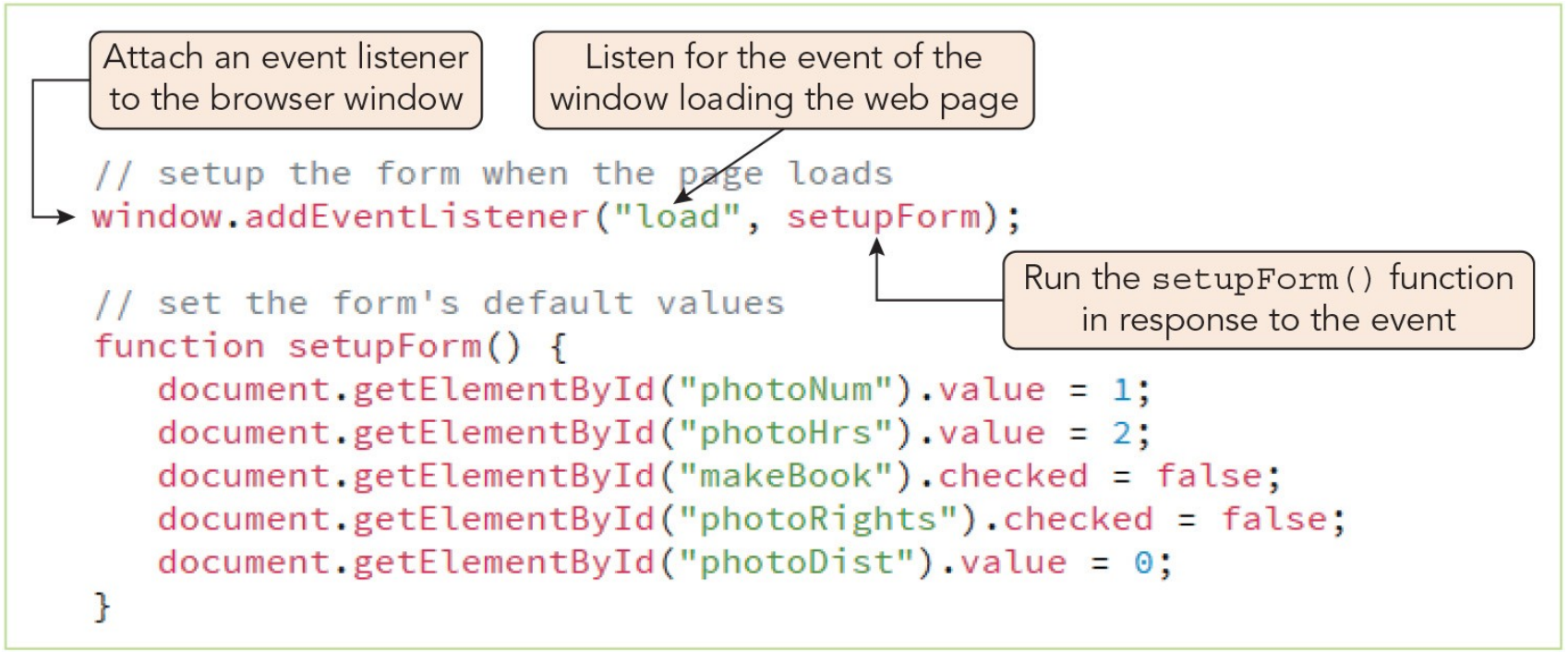
# Managing Events with Functions (1 of 3)

- Using event handlers
  - Most direct method of associating a function with an event
  - Drawback: places JavaScript code in the HTML file
  - Syntax for creating an event handler as an attribute of the HTML element:
    ```
    <elem onevent = "function()">
    ```
- Events as object properties
  - Places the event handler within the JavaScript code file
  - Can only specify function name, not parameter values
  - Only one function can handle an event at a time
  - Syntax for an event as an object property:
    ```
    object.onevent = function;
    ```

# Managing Events with Functions (2 of 3)

- Event listeners
  - An **event listener** listens for an event as it propagates through a web page, during either:
    - The **capture phase** (event moves down the object hierarchy) or
    - The **bubbling phase** (event moves back up the object hierarchy)
  - Can attach multiple functions to the same event
  - Syntax for method that attaches an event listener to an object:
    `object.addEventListener("event", function, capture)`
- Events and anonymous functions
  - Include entire structure of anonymous function in place of function name in an event handler or event listener
  - Can pass in parameter values with this approach

# Managing Events with Functions (3 of 3)




```
// setup the form when the page loads
window.addEventListener("load", setupForm);

// set the form's default values
function setupForm() {
   document.getElementById("photoNum").value = 1;
   document.getElementById("photoHrs").value = 2;
   document.getElementById("makeBook").checked = false;
   document.getElementById("photoRights").checked = false;
   document.getElementById("photoDist").value = 0;
}
```

Figure 2-3 Creating an event listener

# Using Built-in JavaScript Functions (1 of 2)

| Function | Description |
| --- | --- |
| `decodeURI (string)` | Decodes text strings encoded with `encodeURI()` |
| `decodeURIComponent (string)` | Decodes text strings encoded with `encodeURIComponent()` |
| `encodeURI (string)` | Encodes a text string so it becomes a valid URI |
| `encodeURIComponent (string)` | Encodes a text string so it becomes a valid URI component |
| `eval (string)` | Evaluates expressions contained within strings |
| `isFinite (number)` | Determines whether a number is finite |
| `isNaN (number)` | Determines whether a value is the special value `NaN` (Not a Number) |
| `parseFloat (string)` | Converts string literals to floating-point numbers |
| `parseInt (string)` | Converts string literals to integers |

# Using Built-in JavaScript Functions (2 of 2)

- Example of using built-in function to verify the `socialSecurityNumber` variable is not a number:

```
let socialSecurityNumber = "123-45-6789";
let checkVar = isNaN(socialSecurityNumber);
document.write(checkVar);
```

# Understanding Variable Scope (1 of 3)

- **Scope**: where a variable or function can be called within the program
  - Variable/function is only recognized within scope
  - Referencing elsewhere results in an error
- `let` and `var` declaration scopes
  - Variables declared with `let` are **block scoped**: scope is limited to the command block
  - Variables declared with `var` are **function scoped**: scope is limited to the function
- Local and global scope
  - Variables/functions with **local scope** (e.g., **local variables**) are accessible within the command block or function where they are defined
    - Includes block scope and function scope
  - Those with **global scope** (e.g., **global variables**) are defined outside a block/function and thus accessible throughout the program

# Understanding Variable Scope (2 of 3)

- Local and global scope (continued)
  - Can create local and global variables with the same name but different values
    - Local variable takes precedence when in scope
    - Assigning a value to the local variable does not affect the global variable's value outside the local variable's scope
  - Global variables most useful for small applications and variables used as constants
  - Local variables preferable for values used within and changed by functions

# Understanding Variable Scope (3 of 3)

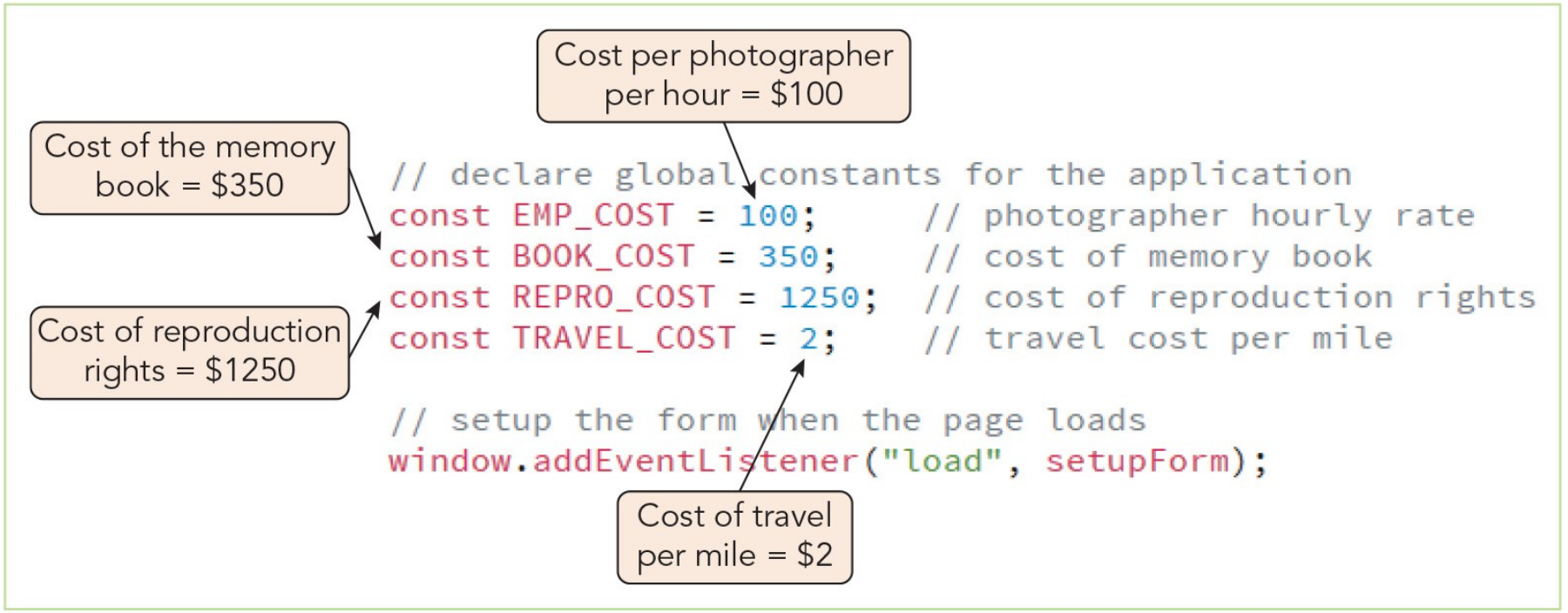



Figure 2-5 Declaring constant variables

# Activity 2.1: Knowledge Check

1. Describe the syntax for creating and then calling a function.

2. What type(s) of scope do the variables in this code sample have?

```
let quantityPerBox = 12;
function describeCandy(productName, numberOfBoxes) {
   var numberCandies = quantityPerBox * numberOfBoxes;
   document.write("You would like " + numberCandies + " of our
   " + productName + " candies!");
}
```

# Activity 2.1: Knowledge Check Answers (1 of 2)

1. Describe the syntax for creating and then calling a function.

   Syntax for a named function:

```
function functionName(parameters) {
   statements
}
```

   Syntax for an anonymous function:

```
function (parameters) {
   statements
}
```

   Syntax for calling a function:

```
functionName(paramValues);
```

# Activity 2.1: Knowledge Check Answers (2 of 2)

2. What type(s) of scope do the variables in this code sample have?

```
let quantityPerBox = 12;
function describeCandy(productName, numberOfBoxes) {
   var numberCandies = quantityPerBox * numberOfBoxes;
   document.write("You would like " + numberCandies + " of our
   " + productName + " candies!");
}
```

`quantityPerBox` has global scope, unless this code appears within a command block in the larger program, in which case it would have block scope, a type of local scope. `numberCandies` has function scope, a type of local scope. You can't determine the scope of `productName` or `numberOfBoxes` from this sample since they aren't declared here.

# Working with Data Types (1 of 6)

- **Data type**: the specific category of information that a variable contains
- **Primitive types**: data types that can be assigned only a single value

| Data Type | Description |
|---|---|
| number | A positive or negative number with or without decimal places, or a number written using exponential notation |
| Boolean | A logical value of `true` or `false` |
| string | Text such as “Hello World!” |
| undefined | An unassigned, undeclared, or nonexistent value |
| null | An empty value |

# Working with Data Types (2 of 6)

- **Strongly typed** (**statically typed**) programming languages require that you declare the type of data that a variable contains and do not allow you to alter that type
- **Loosely typed** (**duck typed**, **dynamically typed**) programming languages do not require you to declare the data type and allow data types to be change
- JavaScript is loosely typed
  - Data types cannot be declared when variables are created
  - JavaScript interpreter determines and assigns or reassigns the variable's data type based on the type of data stored
  - ```
    diffTypes = "Hello World!"; // String
    diffTypes = 8;              // Integer number
    diffTypes = 5.367;          // Floating-point number
    diffTypes = true;           // Boolean
    diffTypes = null;           // Null
    ```
    ```

# Working with Data Types (3 of 6)

- Working with numeric values
  - **Integer:** positive or negative number without decimal places
  - **Floating point number:** positive or negative numbers containing decimal places
    - Can be written in **exponential notation** (**scientific notation**): `2.0e6` = $2 \times 10^6$ = 2,000,000
  - Use integers to calculate monetary values
    - Calculations on integer values, but not on floating point numbers, are the same in binary and decimal
- Working with Boolean values
  - **Boolean value:** a logical value of true or false
  - Most often used for controlling program flow or for data comparisons

# Working with Data Types (4 of 6)

- Working with strings
  - Text string: zero or more characters surrounded by double or single quotation marks
  - **Empty string**: zero-length string value
  - Can use quotation marks within strings:

```
document.write("Welcome to 'Fan Trick Photography'");
document.write('Welcome to "Fan Trick Photography"');
```

  - To split a text string onto a new line without causing an error:
    - Use two or more strings concatenated by the addition operator (+)
    - For some browsers, end a line with the \ character to indicate the string continues
    - Create a **template literal** by enclosing the string in backtick characters (`)

# Working with Data Types (5 of 6)

- Escape characters and sequences
  - An **escape character** is placed before characters within strings to indicate that they are to be treated as regular keyboard characters, not as syntax
  - JavaScript's escape character is the backslash (\)
  - **Escape sequence**: combination of an escape character with a specific character, usually to carry out a special function

# Working with Data Types (6 of 6)

| Escape Sequence | Character |
|---|---|
| `\\` | Backslash |
| `\b` | Backspace |
| `\r` | Carriage return |
| `\"` | Double quotation mark |
| `\f` | Form feed |
| `\t` | Horizontal tab |
| `\n` | Newline |
| `\0` | Null character |
| `\'` | Single quotation mark (apostrophe) |
| `\v` | Vertical tab |
| `\x`*XX* | Latin-1 character specified by the *XX* characters, which represent two hexadecimal digits |
| `\u`*XXXX* | Unicode character specified by the *XXXX* characters, which represent four hexadecimal digits |

# Using Operators to Build Expressions (1 of 9)

- Two types of JavaScript operators: binary and unary
- **Binary operator:** requires an operand before and after the operator
- **Arithmetic operators:** operators used to perform mathematical operations

| Operator | Description | Expression | Returns |
|---|---|---|---|
| + | Combines or adds two items | 12 + 3 | 15 |
| – | Subtracts one item from another | 12 – 3 | 9 |
| * | Multiplies two items | 12*3 | 36 |
| / | Divides one item by another | 12/3 | 4 |
| % | Returns the remainder (**modulus**) after dividing one integer by another integer | 18%5 | 3 |
| ** | Raising a value to a power | 3**2 | 9 |

# Using Operators to Build Expressions (2 of 9)

- Arithmetic operators (continued)
  - **Unary operator:** requires just a single operand either before or after the operator

| Operator | Description | Expression | Returns |
|---|---|---|---|
| ++ | Increases a value by 1 | 12++ | 13 |
| – – | Decreases a value by 1 | 12– – | 11 |
| – | Changes the sign of a value | –12 | –12 |

# Using Operators to Build Expressions (3 of 9)

- Arithmetic operators (continued)
  - Two types of unary operators:
    - **Prefix operators**, which are placed before the variable
    - **Postfix operators**, which are placed after the variable
  - Prefix operator is applied before assignment operator:

```
let x = 5;
let y = ++x // x = 6 and y = 6
```

  - Postfix operator is applied after assignment operator:

```
let x = 5;
let y = x++ // x = 6 and y = 5
```

# Using Operators to Build Expressions (4 of 9)

- Assignment operators
  - An **assignment operator** (e.g., =) is used for assigning a value to a variable
  - **Compound assignment operators** both assign a value and perform a calculation
    - Interpreter will attempt to convert a nonnumeric to a numeric operand

| Operator | Example | Equivalent to |
|---|---|---|
| = | x = y | x = y |
| += | x += y | x = x + y |
| −= | x −= y | x = x − y |
| *= | x *= y | x = x * y |
| /= | x /= y | x = x/y |
| %= | x %= y | x = x % y |
| **= | x **= y | x = x**y |

# Using Operators to Build Expressions (5 of 9)

- **Comparison operators** (**relational operators**): used to compare two operands
  - Two nonnumerical operands are compared in lexicographical order
  - String plus number: interpreter converts string to number or returns `false`

| Operator | Example | Description |
|---|---|---|
| == | x == y | Tests whether x is equal in value to y |
| === | x === y | Tests whether x is equal in value to y and has the same data type |
| != | x != y | Tests whether x is not equal to y or has a different data type |
| !== | x !== y | Tests whether x is not equal to y and/or doesn't have the same data type |
| > | x > y | Tests whether x is greater than y |
| >= | x >= y | Tests whether x is greater than or equal to y |
| < | x < y | Tests whether x is less than y |
| <= | x <= y | Tests whether x is less than or equal to y |

# Using Operators to Build Expressions (6 of 9)

- **Conditional operators** (**ternary operators**) return one of two possible values given the Boolean value of comparison
  - Syntax: `condition ? trueValue : falseValue;`
  - Condition can be any expression that equals `true` or `false`, including a Boolean variable
  - Can return an expression instead of a value
- Understanding falsy and truthy values
  - **Falsy values**, equivalent to `false`: `""` (empty string), `-0`, `0`, `NaN`, `null`, `undefined`
  - Everything else is a **truthy value**, equivalent to `true`
  - Can often use truthy and falsy values to make comparison operations more compact by omitting the comparison operator

# Using Operators to Build Expressions (7 of 9)

- **Logical operators**
  - Used to combine expressions that will result in a Boolean value of `true` or `false`
  - Used for negating (swapping) a Boolean value
  - Multiple conditions can be grouped within parentheses to create more complicated statements

| Operator | Definition | Example | Description |
|---|---|---|---|
| `&&` | and | `(x === 5) && (y === 8)` | Tests whether x is equal to 5 and y is equal to 8 |
| `\|\|` | or | `(x === 5) \|\| (y === 8)` | Test whether x is equal to 5 or y is equal to 8 |
| `!` | not | `! (x < 5)` | Test whether x is not less than 5 |

# Using Operators to Build Expressions (8 of 9)

| Name | Special Operator | Description |
|---|---|---|
| Property access | `.` | Appends an object, method, or property to another object |
| Array index | `[]` | Accesses an element of an array |
| Function call | `()` | Calls up functions or changes the order in which individual operations in an expression are evaluated |
| Comma | `,` | Separates multiple expressions in the same statement |
| Conditional expression | `?:` | Executes one of two expressions based on the results of a conditional expression |
| Delete | `delete` | Deletes array elements, variables created without the `var` keyword, and properties of custom objects |

# Using Operators to Build Expressions (9 of 9)

| Name | Special Operator | Description |
|---|---|---|
| Property exists | `in` | Returns a value of `true` if a specified property is contained within an object |
| Object type | `instanceof` | Returns `true` if an object is of a specified object type |
| New object | `new` | Creates a new instance of a user-defined object type or a predefined JavaScript object type |
| Data type | `typeof` | Determines the data type of a variable |
| Void | `void` | Evaluates an expression without returning a result |

# Activity 2.2: Think, Pair, and Share

1. Form pairs/groups of two to four class members.
2. Your group will be working with a floating point number variable called `totalDue`, which represents money owed by a customer. Work together to write a function to perform each of the following tasks:

   Add a $5 handling charge to `totalDue`.

   Add both a $5 handling charge and a 15% heavy item charge. The heavy item charge should be calculated *before* the handling charge is added to `totalDue`.
3. Now write code that calls the first function if the `totalWeight` variable's value is less than 20 or the second function if `totalWeight` is greater than or equal to 20.

# Understanding Operator Precedence (1 of 3)

- **Operator precedence** determines the order in which operations in an expression are evaluated
- **Associativity** determines precedence for operators with equal intrinsic precedence
- Examples:
  - `5 + 2 * 8` evaluates to 21
  - `30 / 5 * 2` evaluates to 12
  - 
```
let x = 3;
let y = 2;
x = y *= ++x;  // Value of both x and y is 8
```
  - `(5 + 2) * 8` evaluates to 56

# Understanding Operator Precedence (2 of 3)

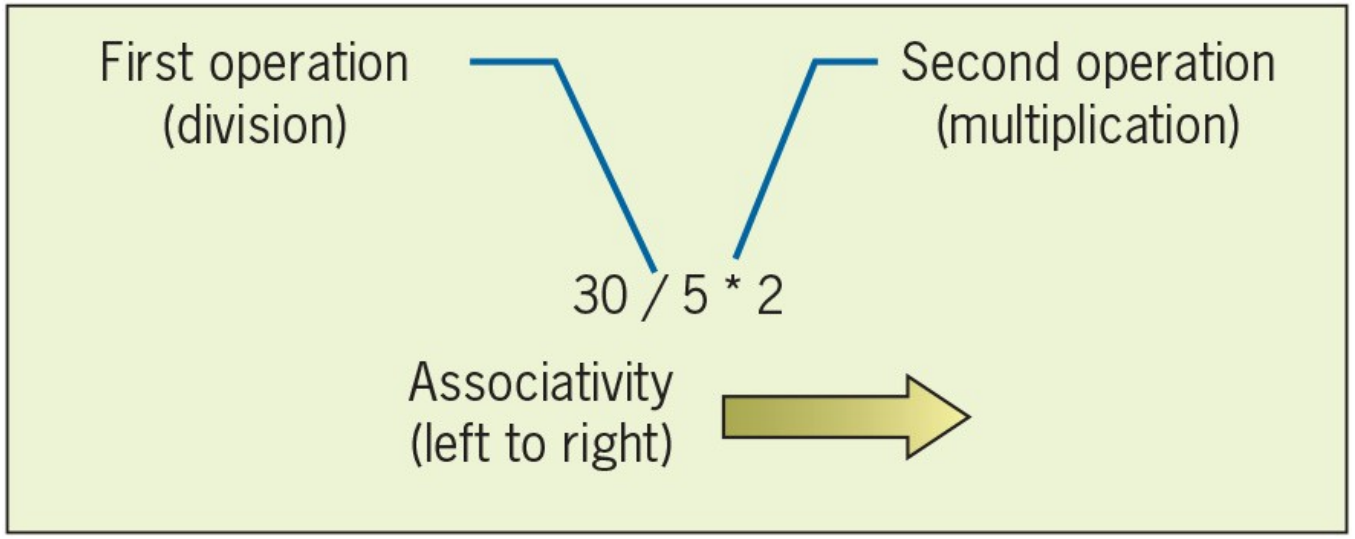



Figure 2-16 Left-to-right associativity

# Understanding Operator Precedence (3 of 3)

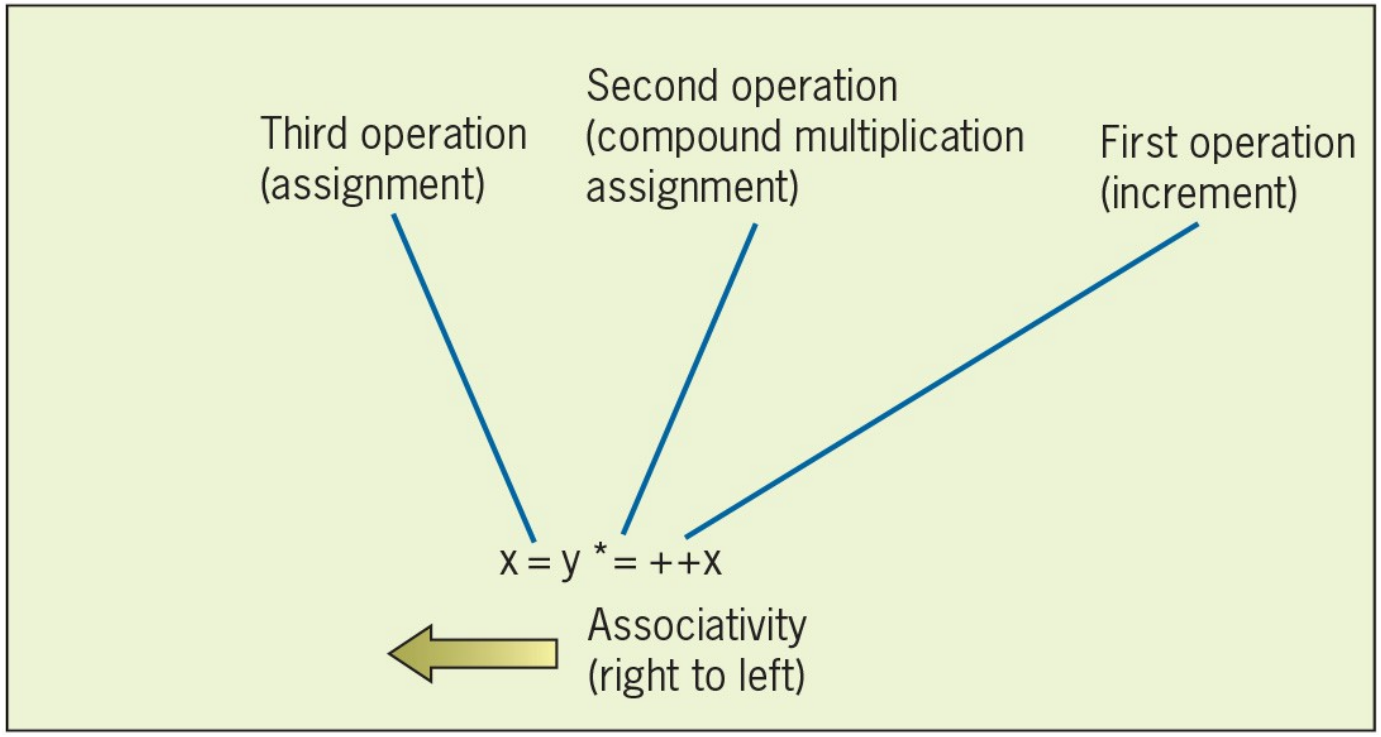



Figure 2-17 Right-to-left associativity

# Using Expressions with Web Form Controls (1 of 7)

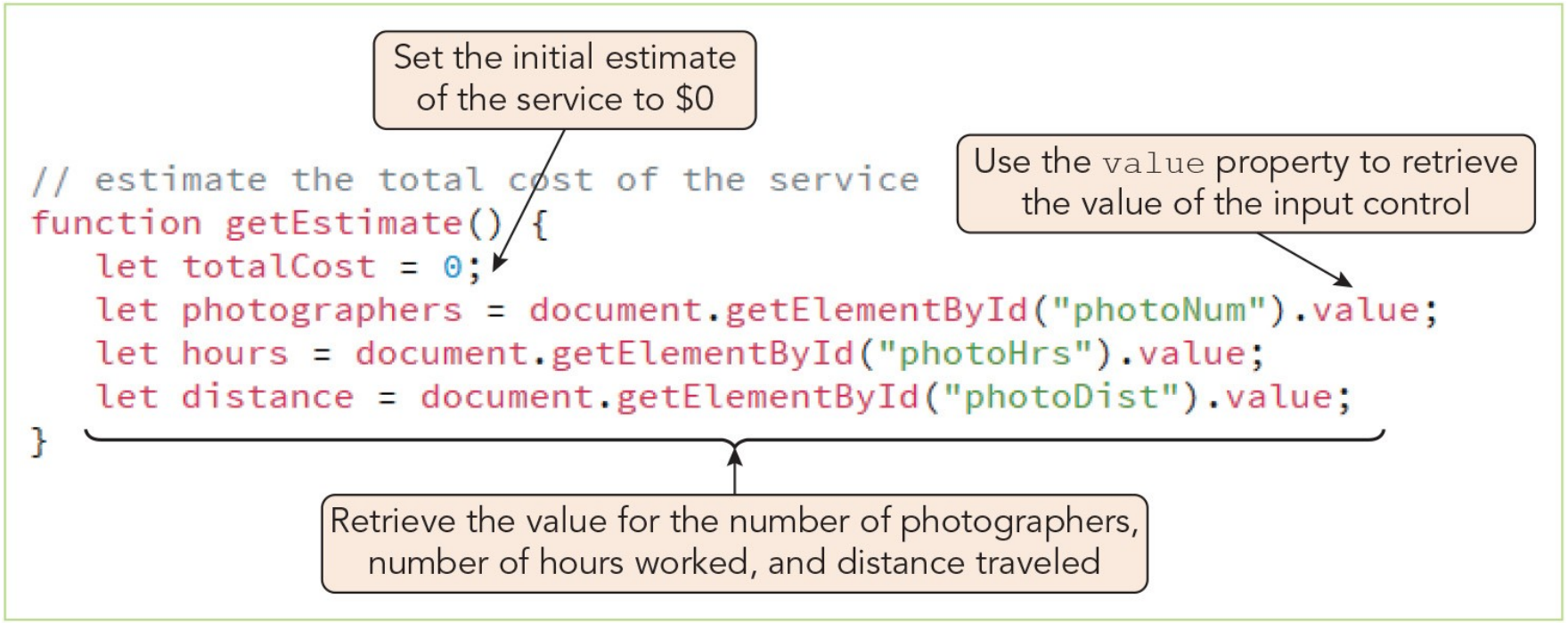
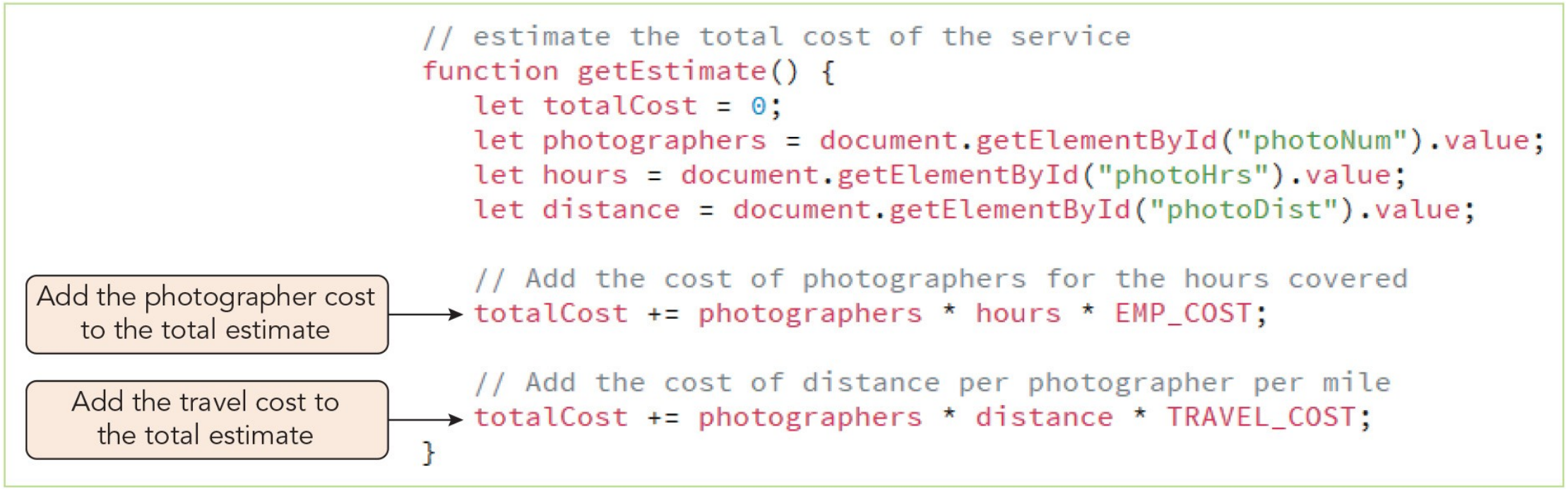



Figure 2-18 Retrieving input control values

# Using Expressions with Web Form Controls (2 of 7)



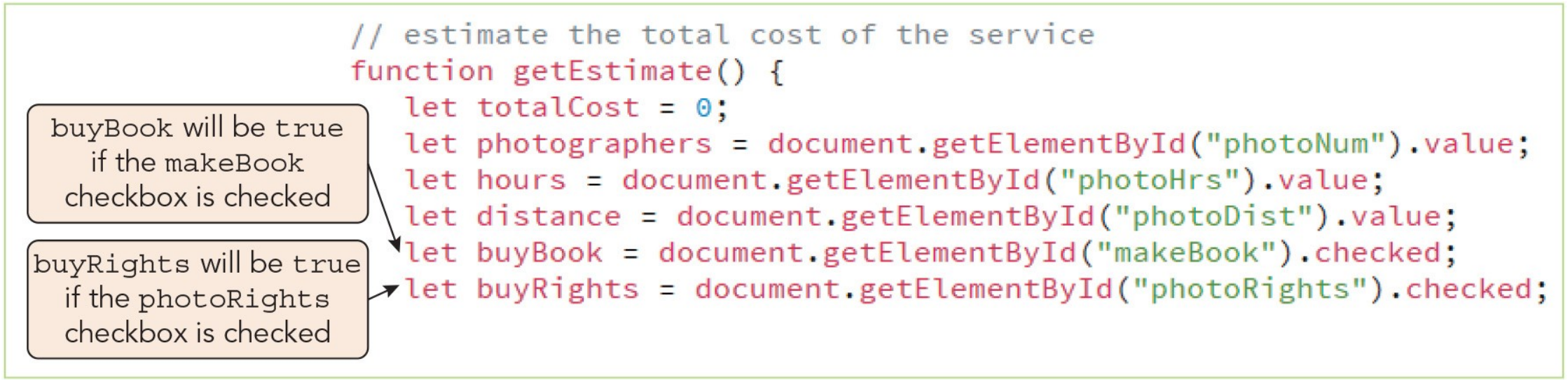

```
// estimate the total cost of the service
function getEstimate() {
   let totalCost = 0;
   let photographers = document.getElementById("photoNum").value;
   let hours = document.getElementById("photoHrs").value;
   let distance = document.getElementById("photoDist").value;

   // Add the cost of photographers for the hours covered
   totalCost += photographers * hours * EMP_COST;

   // Add the cost of distance per photographer per mile
   totalCost += photographers * distance * TRAVEL_COST;
}
```

Figure 2-19 Adding to the total cost estimate

# Using Expressions with Web Form Controls (3 of 7)




```
// estimate the total cost of the service
function getEstimate() {
   let totalCost = 0;
   let photographers = document.getElementById("photoNum").value;
   let hours = document.getElementById("photoHrs").value;
   let distance = document.getElementById("photoDist").value;
   let buyBook = document.getElementById("makeBook").checked;
   let buyRights = document.getElementById("photoRights").checked;
```

Figure 2-20 Retrieving the checked status of web form checkboxes

# Using Expressions with Web Form Controls (4 of 7)

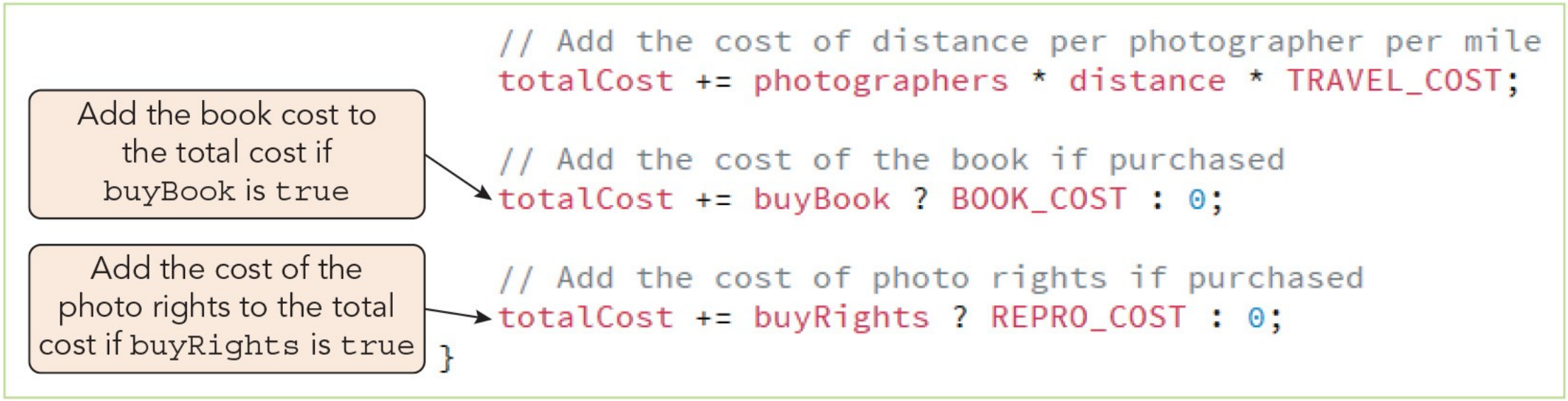



Figure 2-21 Adding the cost of the memory book and the photo rights

# Using Expressions with Web Form Controls (5 of 7)

```
// estimate the total cost of the service
function getEstimate() {
   let totalCost = 0;
   let photographers = document.getElementById("photoNum").value;
   let hours = document.getElementById("photoHrs").value;
   let distance = document.getElementById("photoDist").value;
   let buyBook = document.getElementById("makeBook").checked;
   let buyRights = document.getElementById("photoRights").checked;

   // Add the cost of photographers for the hours covered
   totalCost += photographers * hours * EMP_COST;

   // Add the cost of distance per photographer per mile
   totalCost += photographers * distance * TRAVEL_COST;

   // Add the cost of the book if purchased
   totalCost += buyBook ? BOOK_COST : 0;

   // Add the cost of photo rights if purchased
   totalCost += buyRights ? REPRO_COST : 0;

   // Display the total cost estimate
   document.getElementById("estimate").innerHTML = "$" + totalCost;
}
```

Display `totalCost` prefaced by the `$` character

Figure 2-22 Displaying the total cost estimate in the web page

# Using Expressions with Web Form Controls (6 of 7)

```
// set the form's default values
function setupForm() {
   document.getElementById("photoNum").value = 1;
   document.getElementById("photoHrs").value = 2;

   document.getElementById("makeBook").checked = false;
   document.getElementById("photoRights").checked = false;
   document.getElementById("photoDist").value = 0;

   getEstimate();
}
```

Run the `getEstimate()` function when the browser loads the page

Figure 2-23 Calling the `getEstimate()` function when the page loads

# Using Expressions with Web Form Controls (7 of 7)

```
// set the form's default values
function setupForm() {
   document.getElementById("photoNum").value = 1;
   document.getElementById("photoHrs").value = 2;

   document.getElementById("makeBook").checked = false;
   document.getElementById("photoRights").checked = false;
   document.getElementById("photoDist").value = 0;

   getEstimate();

   // Add event handlers for each input control
   document.getElementById("photoNum").onchange = getEstimate;
   document.getElementById("photoHrs").onchange = getEstimate;
   document.getElementById("photoDist").onchange = getEstimate;
   document.getElementById("makeBook").onchange = getEstimate;
   document.getElementById("photoRights").onchange = getEstimate;
}
```

Run the `getEstimate()` function when the values of any of the 5 input controls are changed

Figure 2-25 Adding event handlers for each input control

# Locating Errors with the Browser Console (1 of 3)

- Accessing the **browser console** (**console**) displays error messages from the browser
- Locating an error in your program
  - Browser console reports the line where detected each error is located
  - Also reports lines that failed to run
  - Be sure to make permanent corrections to code within your code editor

# Locating Errors with the Browser Console (2 of 3)

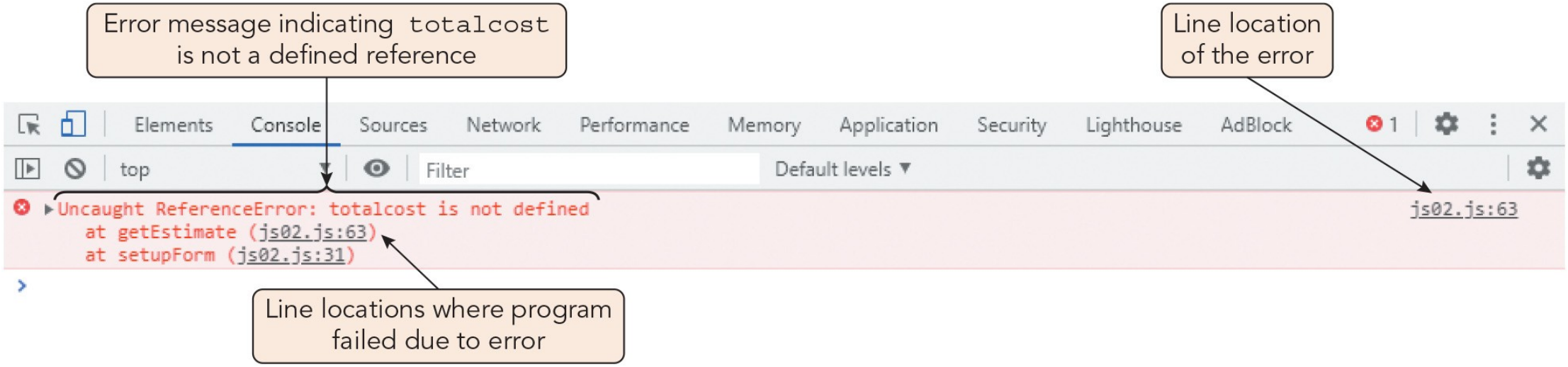



Figure 2-28 Browser console message in the Google Chrome browser

# Locating Errors with the Browser Console (3 of 3)

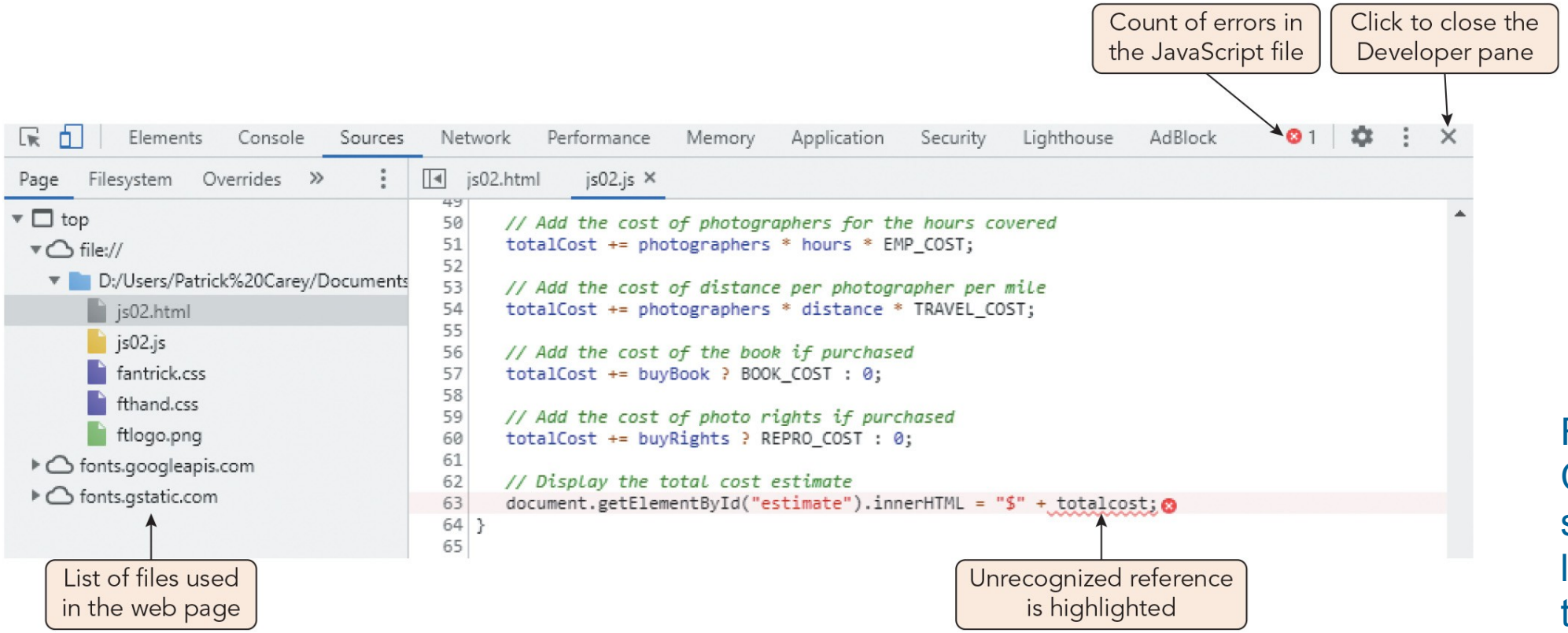



Figure 2-29 Console showing the location of the error

# Activity 2.3: Discussion Questions

1. What are the similarities between a function and a procedure?

2. Why does there need to be a set order of precedence for the JavaScript operators?

# Self-Assessment

1. What role will functions play in the JavaScript programs that you write?

2. How do you plan to use your knowledge of variable and function scope to avoid errors and keep your code easy to understand and maintain?

3. Name some ways you can use JavaScript operators that were new to you when you read this chapter or that you find especially useful.

# Summary

- Now that the lesson has ended, you should have learned to:
  - Write and call functions to perform actions and calculate values.
  - Associate functions with events using event handlers and event listeners.
  - Use built-in JavaScript functions.
  - Understand the scope of variables and functions.
  - Understand the data types supported by JavaScript and write expressions with numeric values, text strings, and Boolean values.
  - Create expressions using arithmetic, assignment, comparison, logical, string, and special operators.
  - Understand order precedence and associativity of operations.
  - Work with events and values associated with form controls.
  - Access your browser's debugging console.