

# JavaScript

---

## INTRODUCTION

# Objectives

---

Insert a script element

Write JavaScript comments

Display an alert dialog box

Use browser debugging tools

Reference browser and page objects

Use JavaScript properties and methods

# Objectives (continued)

---

Write HTML code and text content into a page

Work with a Date object

Use JavaScript operators

Create a JavaScript function

Create timed commands

# Server-Side and Client-Side Programming

---

**Server-side programming:** Program code runs from the server hosting the website

Advantage

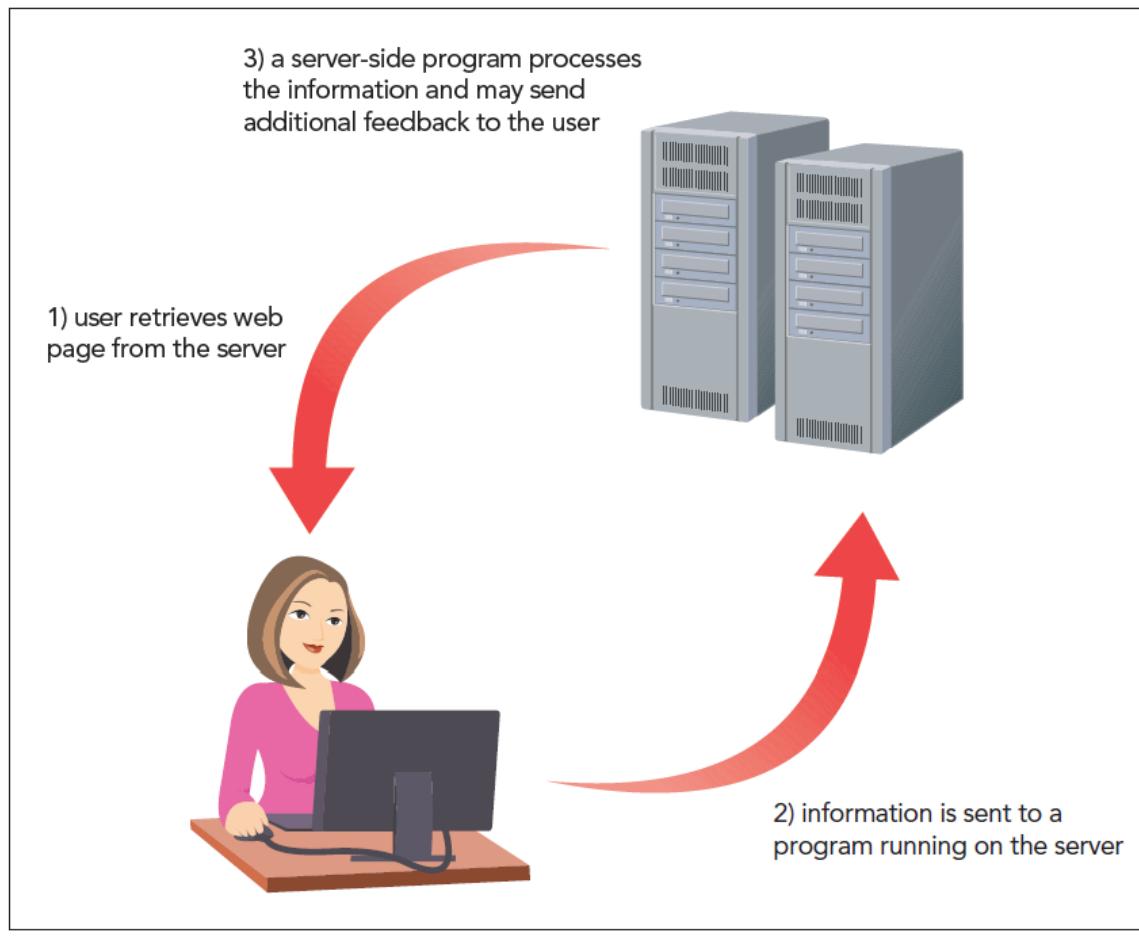
- Connects a server to an online database containing information not directly accessible to end users
- Disadvantages

- Use server resources and requires Internet access
- Long delays in cases of system over-load

# Server-Side and Client-Side Programming (continued 1)

Figure 9-1

Server-side programming



# Server-Side and Client-Side Programming (continued 2)

---

**Client-side programming:** Programs run on the user's computer using downloaded scripts with HTML and CSS files

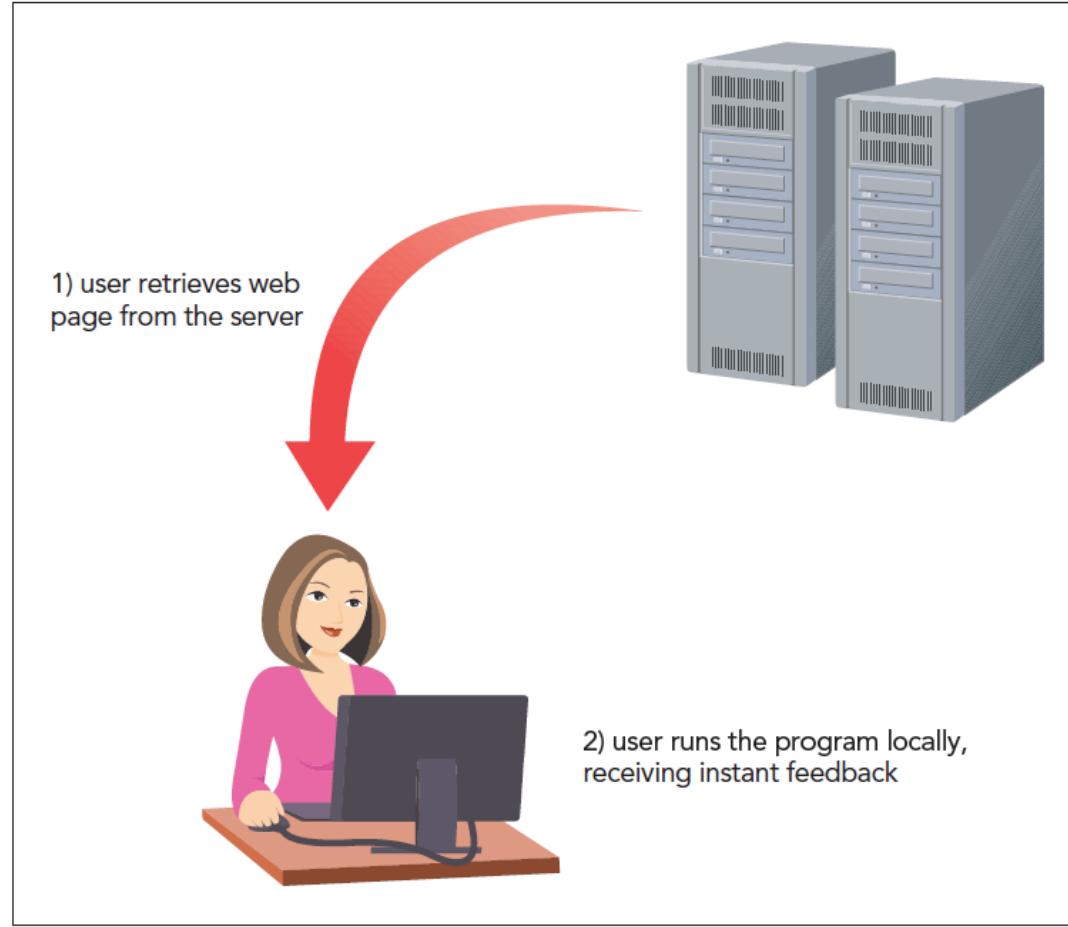
Distributes load to avoid overloading of program-related requests

Client-side programs can never replace server-side programming

# Server-Side and Client-Side Programming (continued 3)

Figure 9-2

Client-side programming



# The Development of JavaScript

---

**JavaScript** is a programming language for client-side programs

It is an **interpreted language** that executes a program code without using an application

**Compiler** is an application that translates a program code into machine language

JavaScript code can be directly inserted into or linked to an HTML file

# Working with the script Element

---

JavaScript code is attached to an HTML file using the `script` element

```
<script src ="url"></script>
```

where `url` is the URL of the external file containing the JavaScript code

- An **embedded script** can be used instead of an external file by omitting the `src` attribute

```
<script>  
    code  
</script>
```

# Loading the script Element

---

script element can be placed anywhere within an HTML document

When a browser encounters a script, it immediately stops loading the page and begins loading and then processing the script commands

async and defer attributes can be added to script element to modify its sequence of processing

# Loading the script Element (continued)

---

`async` attribute tells a browser to parse the HTML and JavaScript code together

`defer` attribute defers script processing until after the page has been completely parsed and loaded

`async` and `defer` attributes are ignored for embedded scripts

# Inserting the script Element

---

Figure 9-4

Inserting the script element

```
<title>Tulsa's New Year's Bash</title>
<link href="tny_reset.css" rel="stylesheet" />
<link href="tny_styles.css" rel="stylesheet" />
<script src="tny_script.js" defer></script>
</head>
```

source of the  
JavaScript file

defers loading the script file  
until after the rest of the page  
is loaded by the browser

# Creating a JavaScript Program

---

JavaScript programs are created using a standard text editor

## Adding Comments to your JavaScript Code

- Comments help understand the design and purpose of programs
- JavaScript comments can be entered on single or multiple lines

# Creating a JavaScript Program (continued 1)

---

- Syntax of a single-line comment is as follows:

*// comment text*

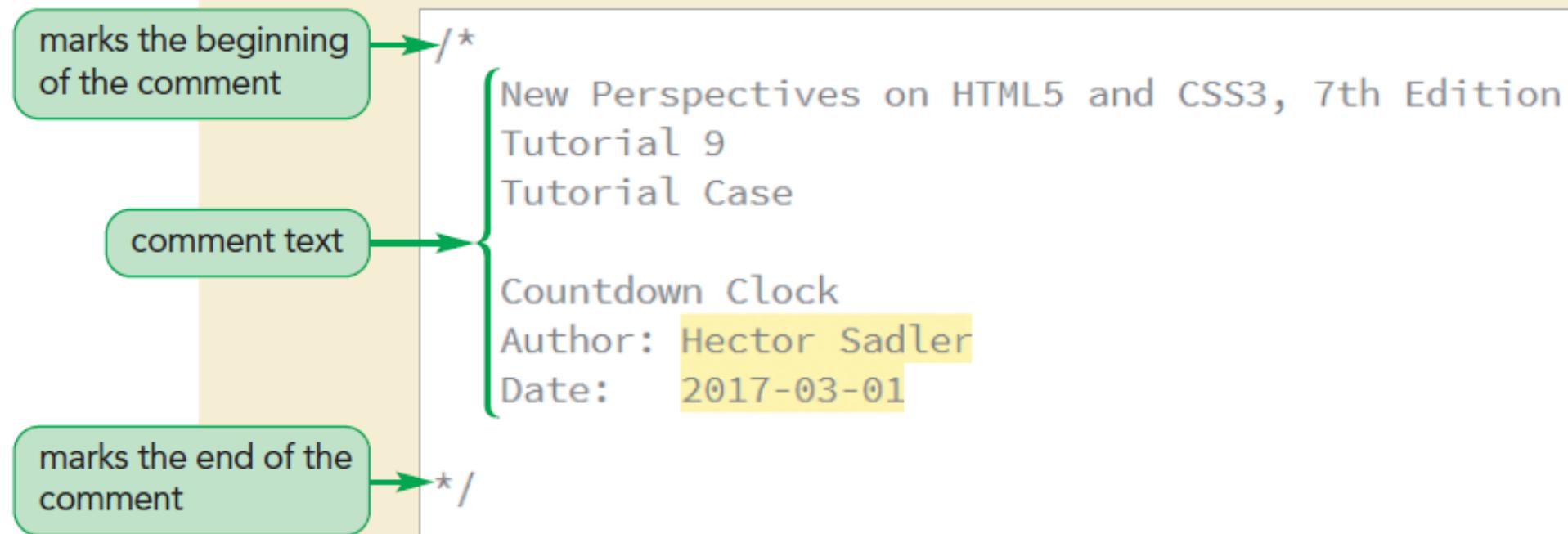
- Syntax of multiple-line comments is as follows:

*/\*  
comment text spanning  
several lines  
\*/*

# Creating a JavaScript Program (continued 2)

---

**Figure 9-6** Adding a JavaScript comment



# Creating a JavaScript Program (continued 3)

---

## Writing a JavaScript Command

- A command indicates an action for a browser to take
- A command should end in a semicolon

*JavaScript command;*

# Creating a JavaScript Program (continued 4)

Figure 9-7

Displaying a dialog box

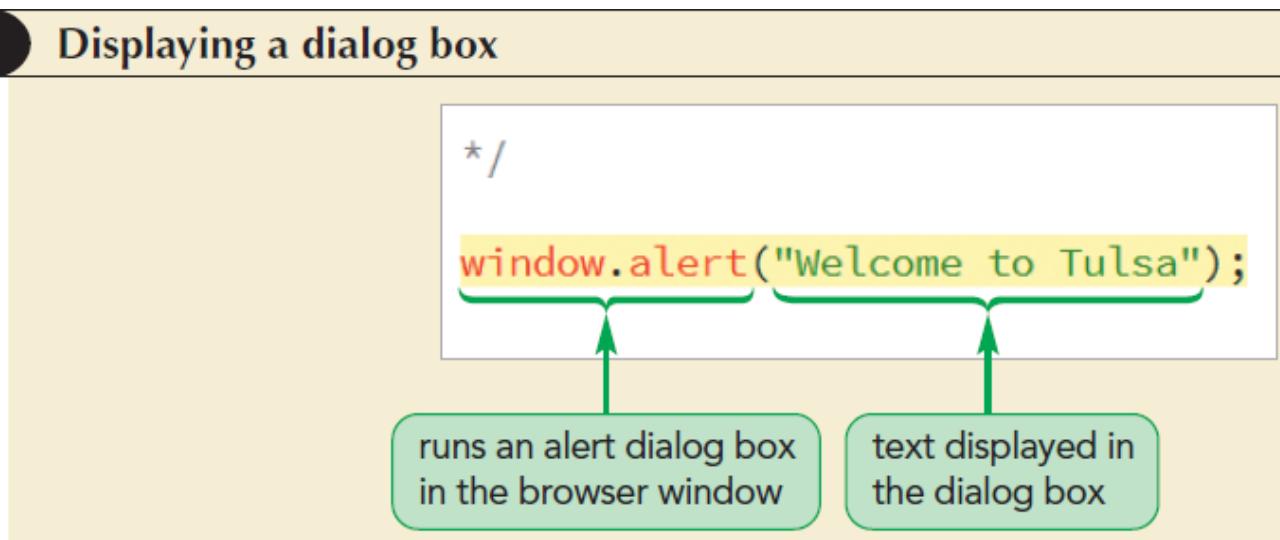
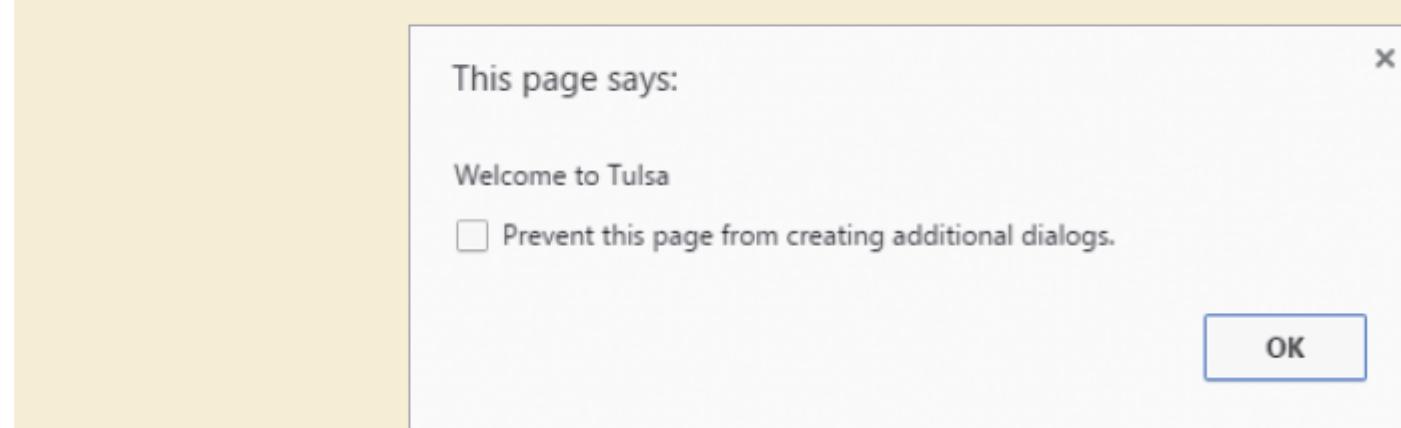


Figure 9-8

Google Chrome dialog box



# Creating a JavaScript Program (continued 5)

---

## Understanding JavaScript Syntax

- JavaScript is case sensitive
- Extra white space between commands is ignored
- Line breaks placed within the name of a JavaScript command or a quoted text string cause an error

# Debugging your Code

---

**Debugging:** Process of locating and fixing a programming error

Types of errors

- Load-time errors – occur when a script is first loaded by a browser
- Run-time errors – occur during execution of a script without syntax errors
- Logical errors – are free from syntax and executable mistakes but result in an incorrect output

# Opening a Debugger

---

Debugging tools locate and fix errors in JavaScript codes

Shortcut to open a debugging tool is F12 key

The tools can also be opened by selecting Developer Tools from the browser menu

# Inserting a Breakpoint

---

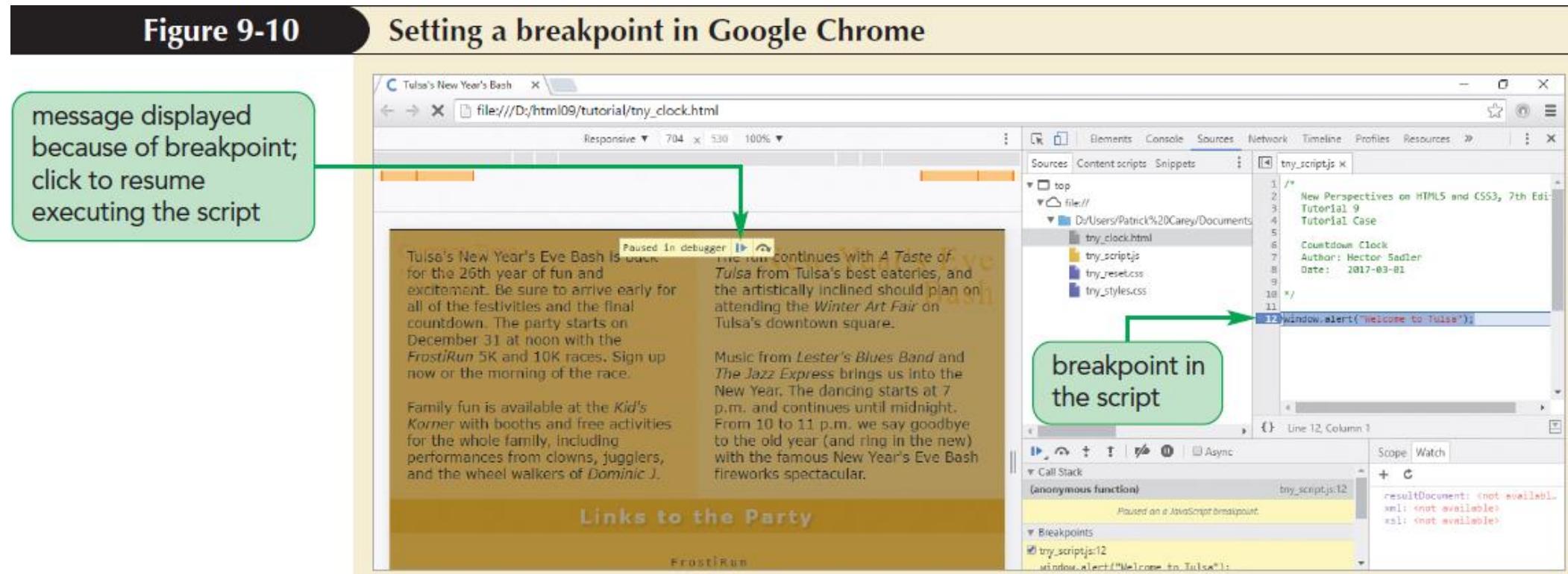
A useful technique to locate the source of an error is to set up **breakpoints**

**Breakpoints** are locations where a browser pauses a program to determine whether an error has occurred at that point during execution

# Inserting a Breakpoint (continued)

Figure 9-10

Setting a breakpoint in Google Chrome



# Applying Strict Usage of JavaScript

---

**Strict mode** enables all lapses in syntax to result in load-time or run-time errors

To run a script in strict mode, add the following statement to the first line of the file:

```
"use strict";
```

# Introducing Objects

---

**Object:** Entity within a browser or web page that has **properties** and **methods**

**Properties:** Define objects

**Methods:** Act upon objects

JavaScript is an **object-based language** that manipulates an object by changing one or more of its properties

# Introducing Objects (continued 1)

---

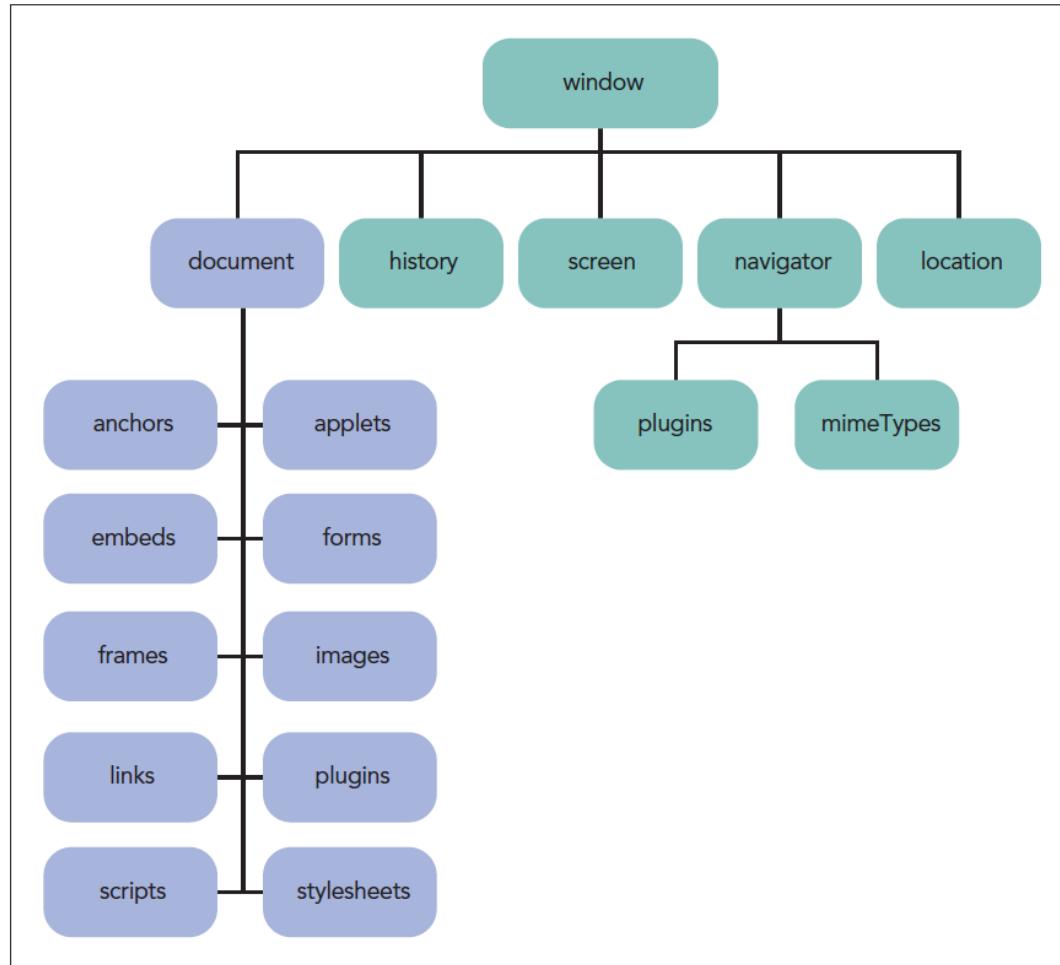
## Types of JavaScript objects

- **Built-in objects** – intrinsic to JavaScript language
- **Browser objects** – part of browser
- **Document objects** – part of web document
- **Customized objects** – created by a programmer to use in an application

**Browser object model (BOM)** and **document object model (DOM)** organize browser and document objects in hierarchical structures, respectively

# Introducing Objects (continued 2)

**Figure 9-12** Object hierarchy



# Object References

---

Objects within the object hierarchy are referenced by their object names such as `window`, `document`, or `navigator`

Objects can be referenced using the notation

*object1.object2.object3 ...*

where *object1* is at the top of the hierarchy, *object2* is a child of *object1*, and so on

# Referencing Object Collections

---

**Object collections:** Objects organized into groups

To reference a specific member of an object collection, use

*collection*[*idref*]

or *collection*.*idref*

where *collection* is a reference to the object collection and *idref* is either an index number or the value of `id` attribute

# Referencing Object Collections (continued)

**Figure 9-13**

**Document object collections**

<b>Object Collection</b>	<b>References</b>
document.anchors	All elements marked with the <a> tag
document.applets	All applet elements
document embeds	All embed elements
document.forms	All web forms
document.frames	All frame elements
document.images	All inline images
document.links	All hypertext links
document.plugins	All plug-ins supported by the browser
document.scripts	All script elements
document.styleSheets	All stylesheet elements

# Referencing an Object by ID and Name

---

An **efficient** approach to reference an element is to use its `id` attribute using the expression

```
document.getElementById(id)
```

where `id` is the value of `id` attribute

# Changing Properties and Applying Methods

---

## Object Properties

- Object property is accessed using

*object.property*

where *object* is a reference to an object and *property* is a property associated with that object

- **Read-only properties** cannot be modified

# Changing Properties and Applying Methods (continued)

---

## Applying a Method

- Objects can be modified using methods
- Methods are applied using the expression

*object.method(values)*

where *object* is a reference to an object, *method* is the name of the method applied to the object, and *values* is a comma-separated list of values associated with the method

# Writing HTML Code

---

HTML code stored within a page element is referenced using

*element.innerHTML*

where *element* is an object reference to an element within a web document

# Writing HTML Code (continued 1)

---

HTML code stored within a page element is referenced using

*element.innerHTML*

where *element* is an object reference to an element within a web document

For example,

```
/* Display the current date and time */
document.getElementById("dateNow").innerHTML =
"m/d/y<br />h:m:s";
```

# Writing HTML Code (continued 2)

**Figure 9-15** Revised date and time content



# Writing HTML Code (continued 3)

---

**Figure 9-16** Properties and methods to insert content

Property or Method	Description
<code>element.innerHTML</code>	Returns the HTML code within <code>element</code>
<code>element.outerHTML</code>	Returns the HTML code within <code>element</code> as well as the HTML code of <code>element</code> itself
<code>element.textContent</code>	Returns the text within <code>element</code> disregarding any HTML tags
<code>element.insertAdjacentHTML(position, text)</code>	Inserts HTML code defined by <code>text</code> into <code>element</code> at <code>position</code> , where <code>position</code> is one of the following: ' <code>beforeBegin</code> ' (before the element's opening tag), ' <code>afterBegin</code> ' (right after the element's opening tag), ' <code>beforeEnd</code> ' (just before the element's closing tag), or ' <code>afterEnd</code> ' (after the element's closing tag)

# Working with Variables

---

**Variable:** Named item in a program that stores a data value

## Declaring a Variable

- Introduced into a script by **declaring** the variable using the `var` keyword

```
var variable = value;
```

where `variable` is the name assigned to the variable and `value` is the variable's initial value

# Working with Variables (continued)

---

Conditions to assign variable names in JavaScript

- First character must be either a letter or an underscore character ( \_ )
- The characters after the first character can be letters, numbers, or underscore characters
- No spaces
- No using names that are part of JavaScript language

# Variables and Data Types

---

**Data type:** Type of information stored in a variable

Supported data types

- Numeric value
- Text string
- Boolean value
- Object
- null value

# Variables and Data Types (continued)

---

- **Numeric value:** Any number
- **Text string:** Group of characters enclosed within either double or single quotation marks
- **Boolean value:** Indicates the truth or falsity of a statement

## Variables and Data Types (continued 1)

---

- Object – Simplifies code by removing the need to rewrite complicated object references
- `null` value – Indicates that no value has yet been assigned to a variable

# Working with Date Objects

**Date object:** Built-in JavaScript object used to store information about dates and times

Figure 9-19

Creating a Date object

```
/*
 * Store the current date and time */
var currentDay = new Date("May 23, 2018 14:35:05");
```

The diagram illustrates the creation of a Date object in JavaScript. It shows the code: `/* Store the current date and time */ var currentDay = new Date("May 23, 2018 14:35:05");`. Three green arrows point from three callout boxes below to specific parts of the code: 1) An arrow points to `currentDay` with the label "declares the currentDay variable". 2) An arrow points to `new Date()` with the label "creates a Date object". 3) An arrow points to the string "May 23, 2018 14:35:05" with the label "date and time stored in the Date object".

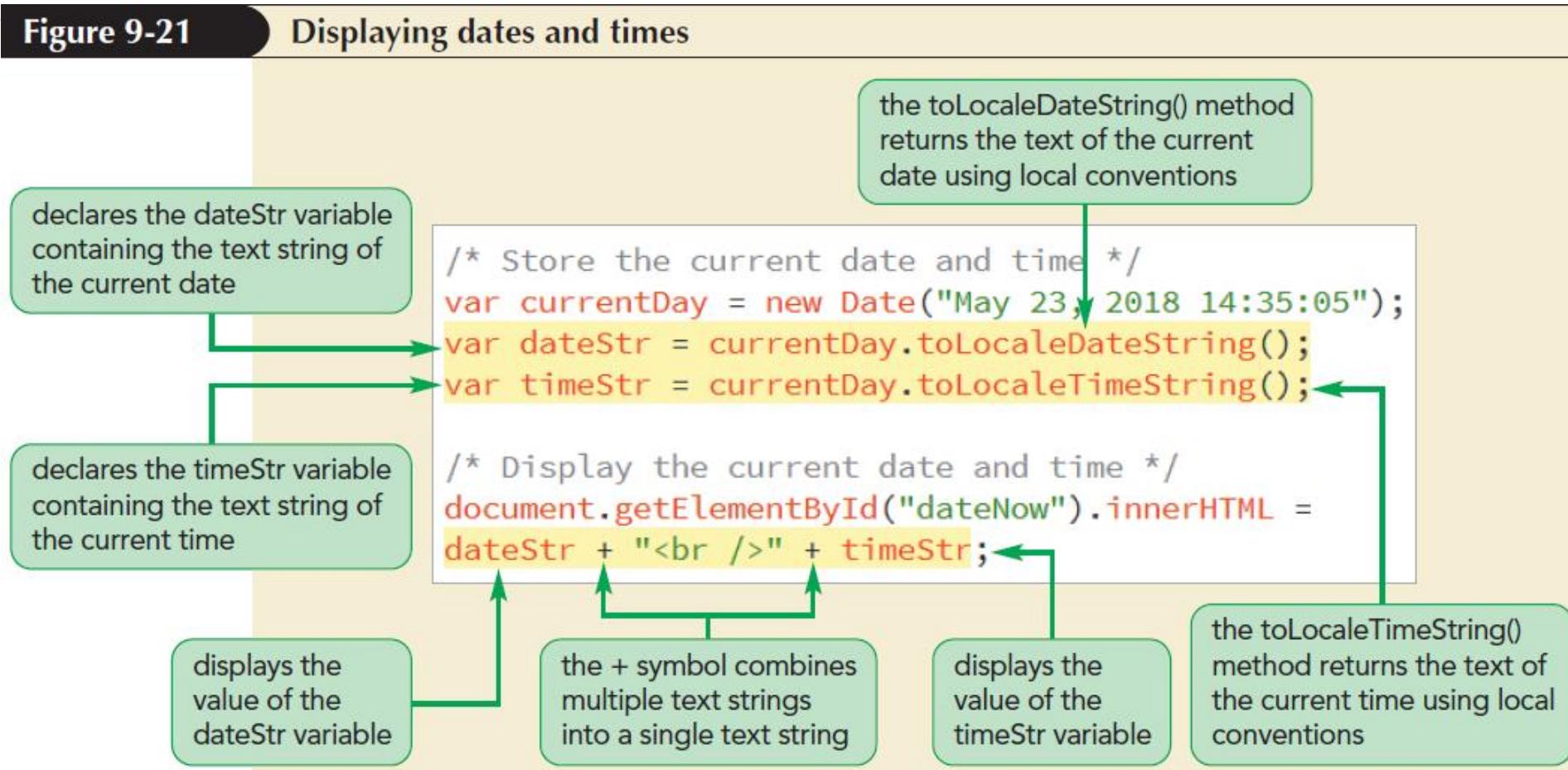
# Working with Date Objects (continued 1)

**Figure 9-20** Methods of the Date object

Date	Method	Description	Result
var thisDay = new Date("May 23, 2018 14:35:05");	thisDay.getSeconds()	seconds	5
	thisDay.getMinutes()	minutes	35
	thisDay.getHours()	hours	14
	thisDay.getDate()	day of the month	23
	thisDay.getMonth()	month number, where January = 0, February = 1, etc.	4
	thisDay.getFullYear()	year	2018
	thisDay.getDay()	day of the week, where Sunday = 0, Monday = 1, etc.	3
	thisDay.toLocaleDateString()	text of the date using local conventions	"5/23/2018"
	thisDay.toLocaleTimeString()	text of the time using local conventions	"2:35:05 PM"

# Working with Date Objects (continued 2)

Figure 9-21    Displaying dates and times



# Setting Date and Time Values

Figure 9-23

JavaScript methods to set values of the Date object

Date Method	Description
<code>date.setDate(value)</code>	Sets the day of the month of <code>date</code> , where <code>value</code> is an integer, ranging from 1 up to 31 (for some months)
<code>date.setFullYear(value)</code>	Sets the four-digit year value of <code>date</code> , where <code>value</code> is an integer
<code>date.setHours(value)</code>	Sets the 24-hour value of <code>date</code> , where <code>value</code> is an integer ranging from 0 to 23
<code>date.setMilliseconds(value)</code>	Sets the millisecond value of <code>date</code> , where <code>value</code> is an integer between 9 and 999
<code>date.setMinutes(value)</code>	Sets the minutes value of <code>date</code> , where <code>value</code> is an integer ranging from 0 to 59
<code>date.setMonth(value)</code>	Sets the month value of <code>date</code> , where <code>value</code> is an integer ranging from 0 (January) to 11 (December)
<code>date.getSeconds(value)</code>	Sets the seconds value of <code>date</code> , where <code>value</code> is an integer ranging from 0 to 59
<code>date.setTime(value)</code>	Sets the time value of <code>date</code> , where <code>value</code> is an integer representing the number of milliseconds since midnight on January 1, 1970

# Working with Operators and Operands

---

**Operator:** Symbol used to act upon an item or a variable within an expression

**Operands:** Variables or expressions that operators act upon

Types of operators

- **Binary operators** – require two operands in an expression

# Working with Operators and Operands (continued)

---

- **Unary operators** – require only one operand
  - **Increment operator (++)** – increases the value of an operand by 1
  - **Decrement operator (--)** – decreases the value of an operand by 1

# Using Assignment Operators

**Assignment operator:** Assigns a value to an item

**Figure 9-25** JavaScript assignment operators

Operator	Example	Equivalent To
=	<code>x = y</code>	<code>x = y</code>
+=	<code>x += y</code>	<code>x = x + y</code>
-=	<code>x -= y</code>	<code>x = x - y</code>
*=	<code>x *= y</code>	<code>x = x * y</code>
/=	<code>x /= y</code>	<code>x = x /y</code>
%=	<code>x %= y</code>	<code>x = x % y</code>

# Working with the Math Object

---

**Math object:** Built-in object used to perform mathematical tasks and store mathematical values

Syntax to apply a Math method is

`Math.method(expression)`

where *method* is the method applied to a mathematical expression

# Working with the Math Object (continued 1)

Figure 9-28

Methods of the Math object

Method	Description	Example	Returns
Math.abs(x)	Returns the absolute value of x	Math.abs(-5)	5
Math.ceil(x)	Rounds x up to the next highest integer	Math.ceil(3.58)	4
Math.exp(x)	Raises e to the power of x	Math.exp(2)	$e^2$ (approximately 7.389)
Math.floor(x)	Rounds x down to the next lowest integer	Math.floor(3.58)	3
Math.log(x)	Returns the natural logarithm of x	Math.log(2)	0.693
Math.max(x, y)	Returns the larger of x and y	Math.max(3, 5)	5
Math.min(x, y)	Returns the smaller of x and y	Math.min(3, 5)	3
Math.pow(x, y)	Returns x raised to the power of y	Math.pow(2,3)	$2^3$ (or 8)
Math.rand()	Returns a random number between 0 and 1	Math.rand()	Random number between 0 and 1
Math.round(x)	Rounds x to the nearest integer	Math.round(3.58)	4
Math.sqrt(x)	Returns the square root of x	Math.sqrt(2)	approximately 1.414

# Working with the Math Object (continued 2)

Figure 9-30

Calculating the hours left in the current day

```
var daysLeft = (newYear - currentDay)/(1000*60*60*24);  
  
/* Calculate the hours left in the current day */  
var hrsLeft = (daysLeft - Math.floor(daysLeft))*24;  
  
/* Display the time left until New Year's Eve */  
document.getElementById("days").textContent = Math.floor(daysLeft);  
document.getElementById("hrs").textContent = Math.floor(hrsLeft);  
document.getElementById("mins").textContent = "mm";  
document.getElementById("secs").textContent = "ss";
```

calculates the fractional part of the current day in terms of hours

in the current day

number of days left in the current year

displays the integer part of hours left



© jbdphotography/Shutterstock.com; Source: www.1001fonts.com

# Using Math Constants

---

Math functions refer to built-in constants stored in JavaScript Math object

Syntax to access mathematical constants is

`Math . CONSTANT`

where *CONSTANT* is the name of one of the mathematical constants supported by Math object

# Using Math Constants (continued)

**Figure 9-34**  **Math constants**

Constant	Description
Math.E	The base of the natural logarithms (2.71828...)
Math.LN10	The natural logarithm of 10 (2.3026...)
Math.LN2	The natural logarithm of 2 (0.6931...)
Math.LOG10E	The base 10 logarithm of e (0.4343...)
Math.LOG2E	The base 2 logarithm of e (1.4427...)
Math.PI	The value of $\pi$ (3.14159...)
Math.SQRT1_2	The value of 1 divided by the square root of 2 (0.7071...)
Math.SQRT2	The square root of 2 (1.4142 ...)

# Working with JavaScript Functions

---

**Function:** Collection of commands that performs an action or returns a value

A function name identifies a function and a set of commands that are run when the function is called

**Parameters:** Variables associated with the function

# Working with JavaScript Functions (continued)

---

General syntax of a JavaScript function is

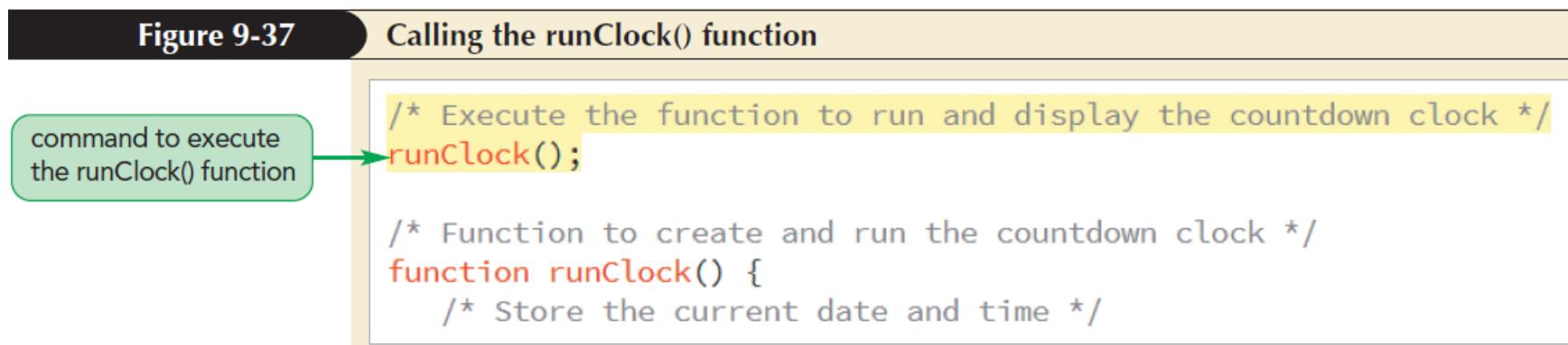
```
function function_name(parameters) {  
    commands  
}
```

where,

- *function\_name* is the name of the function
- *parameters* is a comma-separated list of variables used in the function
- *commands* is the set of statements run by the function

# Calling a Function

---



# Creating a Function to Return a Value

---

Functions return values using `return` statement

```
function function_name(parameters) {  
    commands  
    return value;  
}
```

where *value* is the calculated value that is returned by the function

# Running Timed Commands

---

Methods to update the current and the remaining time constantly

- Time-delayed commands
- Timed-interval commands
- **Working with Time-Delayed Commands**
  - **Time-delayed commands:** JavaScript commands run after a specified amount of time has passed

# Running Timed Commands (continued 1)

---

- Time delay is defined using

```
setTimeout ("command", delay);
```

where *command* is a JavaScript command and *delay* is the delay time in milliseconds before a browser runs the command

- **Running Commands at Specified Intervals**

- The timed-interval command instructs browsers to run a command repeatedly at a specified interval

## Running Timed Commands (continued 2)

---

- Timed-interval commands are applied using `setInterval()` method

```
setInterval("command", interval);
```

where *interval* is the interval in milliseconds before the command is run again

# Running Timed Commands (continued 3)

**Figure 9-38** Repeating the `runClock()` function

```
/* Execute the function to run and display the countdown clock */  
runClock();  
setInterval("runClock()", 1000);
```

repeats the `runClock()`  
function every second

# Controlling How JavaScript Works with Numeric Values

---

## Handling Illegal Operations

- Mathematical operations can return results that are not numeric values
- JavaScript returns NaN if an operation does not involve only numeric values

# Controlling How JavaScript Works with Numeric Values (continued)

---

- `isNaN()` function returns a Boolean value of `true` if the value is not numeric and `false` if otherwise
- `Infinity` value is generated for an operation whose result is less than the smallest numeric value and greater than the largest numeric value supported by JavaScript

# Defining a Number Format

---

JavaScript stores a numeric `value` to 16 decimal places of accuracy

The number of digits displayed by browsers is controlled using `toFixed()` method

`value.toFixed(n)`

where `value` is the value or variable and `n` is the number of decimal places displayed in the output

# Defining a Number Format (continued)

---

`toFixed()` limits the number of decimals displayed by a value and converts the value into a text string

`toFixed()` rounds the last digit in an expression rather than truncating it

# Converting Between Numbers and Text

---

+ operator adds a text string to a number

For example,

```
testNumber = 123; // numeric value  
testString = testNumber + ""; // text string
```

**where** + operator concatenates a numeric value with an empty text string resulting in a text string

# Converting Between Numbers and Text (continued 1)

---

`parseInt()` function extracts the leading integer value from a text string

It returns the integer value from the text string by discarding any non-integer characters

Example,

```
parseInt("120.88 lbs"); // returns 120
```

```
parseInt("weight equals 120 lbs"); // returns  
NaN
```

# Converting Between Numbers and Text (continued 2)

**Figure 9-39** Numerical functions and methods

Numerical Function	Description
<code>isFinite(value)</code>	Indicates whether <code>value</code> is finite and a real number
<code>isNaN(value)</code>	Indicates whether <code>value</code> is a number
<code>parseFloat(string)</code>	Extracts the first numeric value from the text <code>string</code>
<code>parseInt(string)</code>	Extracts the first integer value from the text <code>string</code>
Numerical Method	Description
<code>value.toExponential(n)</code>	Returns a text string displaying <code>value</code> in exponential notation with <code>n</code> digits to the right of the decimal point
<code>value.toFixed(n)</code>	Returns a text string displaying <code>value</code> to <code>n</code> decimal places
<code>value.toPrecision(n)</code>	Returns a text string displaying <code>value</code> to <code>n</code> significant digits either to the left or to the right of the decimal point



# JavaScript: Control Statements, Part 1

JavaScript –introduction



## OBJECTIVES

In this chapter you will:

- Learn basic problem-solving techniques.
- Develop algorithms through the process of top-down, stepwise refinement.
- Use the `if` and `if...else` selection statements to choose among alternative actions.
- Use the `while` repetition statement to execute statements in a script repeatedly.
- Implement counter-controlled repetition and sentinel-controlled repetition.
- Use the increment, decrement and assignment operators.



- 7.1** Introduction
- 7.2** Algorithms
- 7.3** Pseudocode
- 7.4** Control Statements
- 7.5** if Selection Statement
- 7.6** if...else Selection Statement
- 7.7** while Repetition Statement
- 7.8** Formulating Algorithms: Counter-Controlled Repetition
- 7.9** Formulating Algorithms: Sentinel-Controlled Repetition
- 7.10** Formulating Algorithms: Nested Control Statements
- 7.11** Assignment Operators
- 7.12** Increment and Decrement Operators
- 7.13** Web Resources



## 7.2 Algorithms

- ▶ Any computable problem can be solved by executing a series of actions in a specific order
- ▶ A **procedure** for solving a problem in terms of
  - the **actions** to be executed, and
  - the **order** in which the actions are to be executedis called an **algorithm**



# 7.3 Pseudocode

## ▶ Pseudocode

- An informal language that helps you develop algorithms
- Pseudocode is similar to everyday English; it's convenient and user friendly, although it's not an actual computer programming language



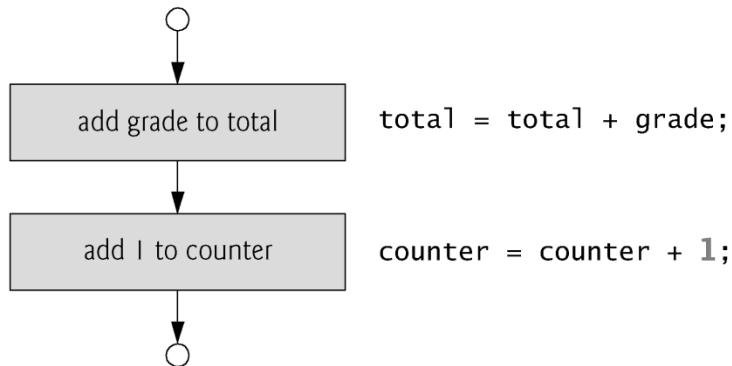
## Software Engineering Observation 7.1

Pseudocode is often used to “think out” a script during the script-design process. Carefully prepared pseudocode can easily be converted to JavaScript.



# 7.4 Control Statements

- ▶ Sequential execution
  - Execute statements in the order they appear in the code
- ▶ Transfer of control
  - Changing the order in which statements execute
- ▶ All scripts can be written in terms of only three control statements
  - sequence
  - selection
  - repetition



---

**Fig. 7.1 |** Flowcharting JavaScript's sequence structure.



# 7.4 Control Statements (Cont.)

## ▶ Flowchart

- A graphical representation of an algorithm or of a portion of an algorithm
- Drawn using certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles
- Symbols are connected by arrows called **flowlines**, which indicate the order in which the actions of the algorithm execute



## 7.4 Control Statements (Cont.)

- ▶ In a flowchart that represents a *complete* algorithm, **oval symbols** containing the words “Begin” and “End” represent the start and the end of the algorithm.
- ▶ In a flowchart that shows only a portion of an algorithm, the oval symbols are omitted in favor of using **small circle symbols**, also called **connector symbols**.
- ▶ Perhaps the most important flowcharting symbol is the **diamond symbol**, also called the **decision symbol**, which indicates that a decision is to be made.



# 7.4 Control Statements (Cont.)

- ▶ JavaScript provides three selection structures.
  - The `if` statement either performs (selects) an action if a condition is true or skips the action if the condition is false.
    - Called a single-selection statement because it selects or ignores a single action or group of actions.
  - The `if...else` statement performs an action if a condition is true and performs a different action if the condition is false.
    - Double-selection statement because it selects between two different actions or group of actions.
  - The `switch` statement performs one of many different actions, depending on the value of an expression.
    - Multiple-selection statement because it selects among many different actions or groups of actions.



## 7.4 Control Statements (Cont.)

- ▶ JavaScript provides four repetition statements, namely, while, do...while, for and for...in.
- ▶ In addition to keywords, JavaScript has other words that are reserved for use by the language, such as the values null, true and false, and words that are reserved for possible future use.



## Common Programming Error 7.1

Using a keyword as an identifier (e.g., for variable names) is a syntax error.

3/3



## JavaScript reserved keywords

break	case	catch	continue	default
delete	do	else	false	finally
for	function	if	in	instanceof
new	null	return	switch	this
throw	true	try	typeof	var
void	while	with		

*Keywords that are reserved but not used by JavaScript*

class	const	enum	export	extends
implements	import	interface	let	package
private	protected	public	static	super
yield				

**Fig. 7.2 |** JavaScript reserved keywords.



## 7.4 Control Statements (Cont.)

- ▶ **Single-entry/single-exit control statements** make it easy to build scripts.
- ▶ Control statements are attached to one another by connecting the exit point of one control statement to the entry point of the next.
  - **Control-statement stacking.**
- ▶ There is only one other way control statements may be connected
  - **Control-statement nesting**



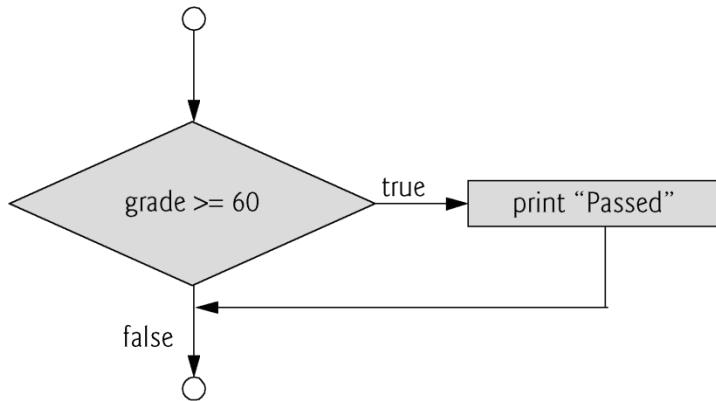
# 7.5 if Selection Statement

- ▶ The JavaScript interpreter ignores *white-space characters*
  - blanks, tabs and newlines used for indentation and vertical spacing
- ▶ A decision can be made on any expression that evaluates to a value of JavaScript's boolean type (i.e., any expression that evaluates to true or false).
- ▶ The indentation convention you choose should be carefully applied throughout your scripts
  - It is difficult to read scripts that do not use uniform spacing conventions



## Good Programming Practice 7.1

Consistently applying reasonable indentation conventions improves script readability. We use three spaces per indent.



---

**Fig. 7.3 |** Flowcharting the single-selection `if` statement.



## Software Engineering Observation 7.2

In JavaScript, any nonzero numeric value in a condition evaluates to `true`, and 0 evaluates to `false`. For strings, any string containing one or more characters evaluates to `true`, and the empty string (the string containing no characters, represented as "") evaluates to `false`. Also, a variable that's been declared with `var` but has not been assigned a value evaluates to `false`.

3.2



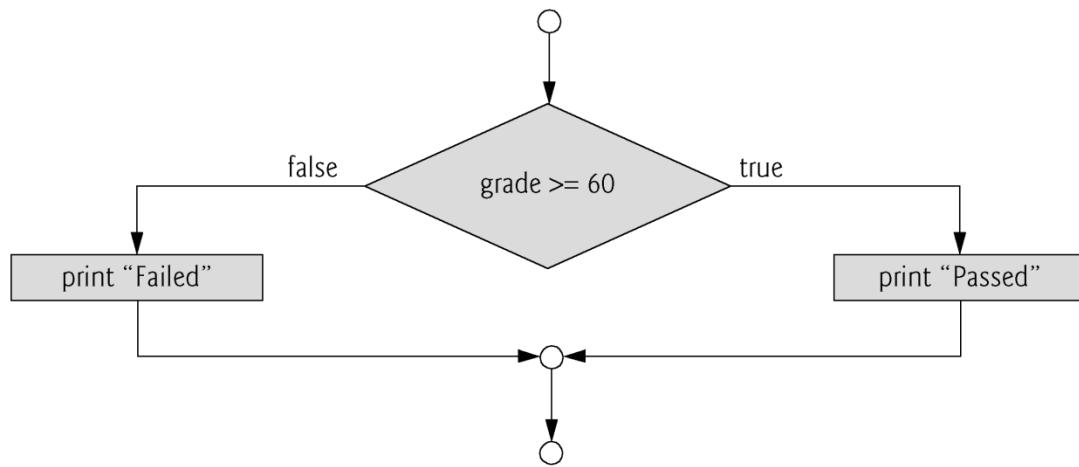
# 7.6 if...else Selection Statement

- ▶ Allows you to specify that different actions should be performed when the condition is true and when the condition is false.



## Good Programming Practice 7.2

Indent both body statements of an `if...else` statement.



**Fig. 7.4 |** Flowcharting the double-selection `if...else` statement.



## 7.6 if...else Selection Statement (Cont.)

- ▶ Conditional operator (?:)
  - Closely related to the if...else statement
  - JavaScript's only ternary operator—it takes three operands
  - The operands together with the ?: operator form a conditional expression
  - The first operand is a boolean expression
  - The second is the value for the conditional expression if the boolean expression evaluates to true
  - Third is the value for the conditional expression if the boolean expression evaluates to false



# 7.6 if...else Selection Statement (Cont.)

- ▶ Nested if...else statements
  - Test for multiple cases by placing if...else statements inside other if...else statements
- ▶ The JavaScript interpreter always associates an else with the previous if, unless told to do otherwise by the placement of braces ({}
  - To include several statements, enclose the statements in braces ({ and })
  - A set of statements contained within a pair of braces is called a block
- ▶ The if selection statement expects only one statement in its body
  - To include several statements, enclose the statements in braces ({ and })
  - A set of statements contained within a pair of braces is called a block



## Good Programming Practice 7.3

If there are several levels of indentation, each level should be indented the same additional amount of space.



## Software Engineering Observation 7.3

A block can be placed anywhere in a script that a single statement can be placed.

10



## Software Engineering Observation 7.4

Unlike individual statements, a block does not end with a semicolon. However, each statement within the braces of a block should end with a semicolon.



## 7.6 if...else Selection Statement (Cont.)

- ▶ A **logic error** has its effect at execution time.
- ▶ A **fatal logic error** causes a script to fail and terminate prematurely.
- ▶ A **nonfatal logic error** allows a script to continue executing, but the script produces incorrect results.



## Software Engineering Observation 7.5

Just as a block can be placed anywhere a single statement can be placed, it's also possible to have no statement at all (the empty statement) in such places. We represent the empty statement by placing a semicolon (;) where a statement would normally be.

10



# 7.7 while Repetition Statement

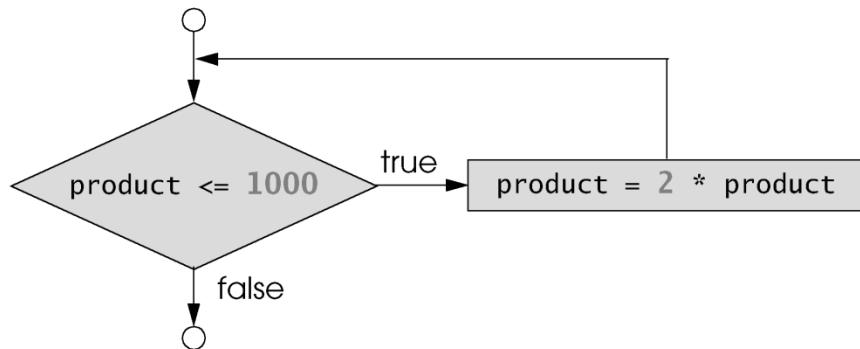
## ▶ while

- Allows you to specify that an action is to be repeated while some condition remains true
- The body of a loop may be a single statement or a block
- Eventually, the condition becomes false and repetition terminates



## Common Programming Error 7.2

If the body of a `while` statement never causes the `while` statement's condition to become true, a logic error occurs. Normally, such a repetition structure will never terminate—an error called an . Many browsers show a dialog allowing the user to terminate a script that contains an infinite loop.



---

**Fig. 7.5 |** Flowcharting the `while` repetition statement.



## 7.8 Formulating Algorithms: Counter-Controlled Repetition

- ▶ Counter-controlled repetition
  - Often called definite repetition, because the number of repetitions is known before the loop begins executing
- ▶ A *total* is a variable in which a script accumulates the sum of a series of values
  - Variables that store totals should normally be initialized to zero before they are used in a script
- ▶ A *counter* is a variable a script uses to count—typically in a repetition statement



- 1** Set total to zero
- 2** Set grade counter to one
- 3**
- 4** While grade counter is less than or equal to ten
  - 5** Input the next grade
  - 6** Add the grade into the total
  - 7** Add one to the grade counter
- 8**
- 9** Set the class average to the total divided by ten
- 10** Print the class average

**Fig. 7.6** | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.



```
1  <!DOCTYPE html>
2
3  <!-- Fig. 7.7: average.html -->
4  <!-- Counter-controlled repetition to calculate a class average. -->
5  <html>
6      <head>
7          <meta charset = "utf-8">
8          <title>Class Average Program</title>
9          <script>
10
11      var total; // sum of grades
12      var gradeCounter; // number of grades entered
13      var grade; // grade typed by user (as a string)
14      var gradeValue; // grade value (converted to integer)
15      var average; // average of all grades
16
17      // initialization phase
18      total = 0; // clear total
19      gradeCounter = 1; // prepare to loop
20
```

**Fig. 7.7 |** Counter-controlled repetition to calculate a class average.  
(Part 1 of 4.)



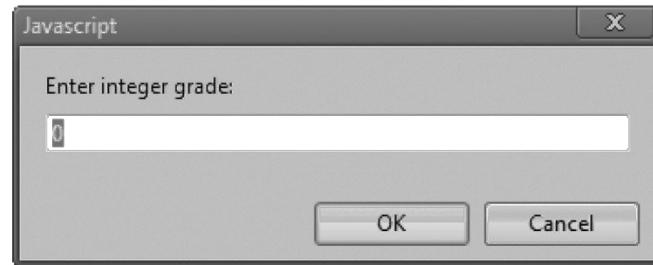
```
21 // processing phase
22 while ( gradeCounter <= 10 ) // loop 10 times
23 {
24
25     // prompt for input and read grade from user
26     grade = window.prompt( "Enter integer grade:", "0" );
27
28     // convert grade from a string to an integer
29     gradeValue = parseInt( grade );
30
31     // add gradeValue to total
32     total = total + gradeValue;
33
34     // add 1 to gradeCounter
35     gradeCounter = gradeCounter + 1;
36 } // end while
37
```

**Fig. 7.7 | Counter-controlled repetition to calculate a class average.  
(Part 2 of 4.)**

```
38     // termination phase
39     average = total / 10;    // calculate the average
40
41     // display average of exam grades
42     document.writeln(
43         "<h1>Class average is " + average + "</h1>" );
44
45     </script>
46 </head><body></body>
47 </html>
```

**Fig. 7.7** | Counter-controlled repetition to calculate a class average.  
(Part 3 of 4.)

a) This dialog is displayed 10 times. User input is 100, 88, 93, 55, 68, 77, 83, 95, 73 and 62. User enters each grade and presses **OK**.



b) The class average is displayed in a web page



**Fig. 7.7** | Counter-controlled repetition to calculate a class average.  
(Part 4 of 4.)



## Common Programming Error 7.3

Not initializing a variable that will be used in a calculation results in a logic error that produces the value NaN (“Not a Number”).



## 7.8 Formulating Algorithms: Counter-Controlled Repetition

- ▶ JavaScript represents all numbers as floating-point numbers in memory
- ▶ Floating-point numbers often develop through division
- ▶ The computer allocates only a fixed amount of space to hold such a value, so the stored floating-point value can only be an approximation



## Software Engineering Observation 7.6

If the string passed to `parseInt` contains a floating-point numeric value, `parseInt` simply truncates the floating-point part. For example, the string "27.95" results in the integer 27, and the string "-123.45" results in the integer -123. If the string passed to `parseInt` does begin with a numeric value, `parseInt` returns `Nan` (not a number). If you need to know whether `parseInt` returned `Nan`, JavaScript provides the function `isNaN`, which determines whether its argument has the value `Nan` and, if so, returns `true`; otherwise, it returns `false`.



## 7.9 Formulating Algorithms: Sentinel-Controlled Repetition

- ▶ Sentinel-controlled repetition
  - Special value called a sentinel value (also called a signal value, a dummy value or a flag value) indicates the end of data entry
  - Often is called indefinite repetition, because the number of repetitions is not known in advance
- ▶ Choose a sentinel value that cannot be confused with an acceptable input value



## 7.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- ▶ Top-down, stepwise refinement
  - A technique that is essential to the development of well-structured algorithms
  - Approach begins with pseudocode of the top, the statement that conveys the script's overall purpose
  - Divide the top into a series of smaller tasks and list them in the order in which they need to be performed—the first refinement
  - Second refinement commits to specific variables



## Software Engineering Observation 7.7

Each refinement, as well as the top itself, is a <sup>complete</sup> specification of the algorithm; only the level of detail varies.



## Error-Prevention Tip 7.1

When performing division by an expression whose value could be zero, explicitly test for this case, and handle it appropriately in your script (e.g., by displaying an error message) rather than allowing the division by zero to occur.



## Software Engineering Observation 7.8

Many algorithms can be divided logically into three phases: an initialization phase that initializes the script variables, a processing phase that inputs data values and adjusts variables accordingly, and a termination phase that calculates and prints the results.

10



- 1 Initialize total to zero
- 2 Initialize gradeCounter to zero
- 3
- 4 Input the first grade (possibly the sentinel)
- 5
- 6 While the user has not as yet entered the sentinel
  - 7 Add this grade into the running total
  - 8 Add one to the grade counter
  - 9 Input the next grade (possibly the sentinel)
- 10
- 11 If the counter is not equal to zero
  - 12 Set the average to the total divided by the counter
  - 13 Print the average
- 14 Else
  - 15 Print "No grades were entered"

**Fig. 7.8 |** Sentinel-controlled repetition to solve the class-average problem.

---



## Software Engineering Observation 7.9

You terminate the top-down, stepwise refinement process after specifying the pseudocode algorithm in sufficient detail for you to convert the pseudocode to JavaScript. Then, implementing the JavaScript is normally straightforward.

7.9



## Software Engineering Observation 7.10

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution.

1/10



## Software Engineering Observation 7.11

Many experienced programmers write scripts without ever using script-development tools like pseudocode. As they see it, their ultimate goal is to solve the problem on a computer, and writing pseudocode merely delays the production of final outputs. Although this approach may work for simple and familiar problems, it can lead to serious errors in large, complex projects.

3.11



## 7.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- ▶ Control statements may be stacked on top of one another in sequence



```
1  <!DOCTYPE html>
2
3  <!-- Fig. 7.9: average2.html -->
4  <!-- Sentinel-controlled repetition to calculate a class average. -->
5  <html>
6      <head>
7          <meta charset = "utf-8">
8          <title>Class Average Program: Sentinel-controlled Repetition</title>
9          <script>
10
11      var total; // sum of grades
12      var gradeCounter; // number of grades entered
13      var grade; // grade typed by user (as a string)
14      var gradeValue; // grade value (converted to integer)
15      var average; // average of all grades
16
17      // initialization phase
18      total = 0; // clear total
19      gradeCounter = 0; // prepare to loop
20
```

**Fig. 7.9 |** Sentinel-controlled repetition to calculate a class average.  
(Part 1 of 4.)



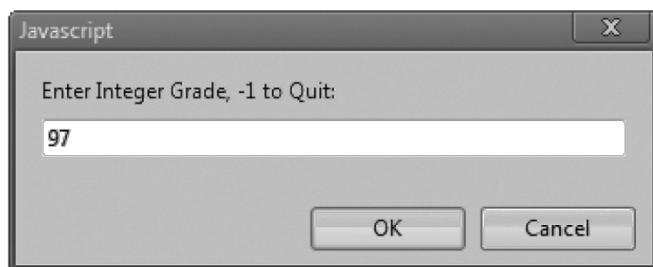
```
21 // processing phase
22 // prompt for input and read grade from user
23 grade = window.prompt(
24     "Enter Integer Grade, -1 to Quit:", "0" );
25
26 // convert grade from a string to an integer
27 gradeValue = parseInt( grade );
28
29 while ( gradeValue != -1 )
30 {
31     // add gradeValue to total
32     total = total + gradeValue;
33
34     // add 1 to gradeCounter
35     gradeCounter = gradeCounter + 1;
36
37     // prompt for input and read grade from user
38     grade = window.prompt(
39         "Enter Integer Grade, -1 to Quit:", "0" );
40
41     // convert grade from a string to an integer
42     gradeValue = parseInt( grade );
43 } // end while
```

**Fig. 7.9 |** Sentinel-controlled repetition to calculate a class average.  
(Part 2 of 4.)

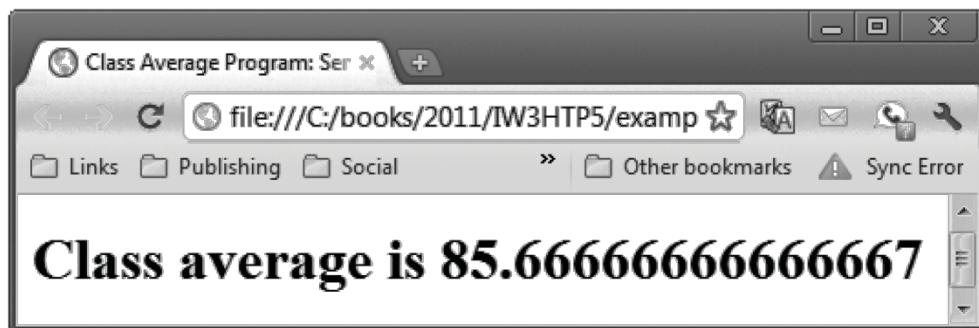


```
44
45      // termination phase
46      if ( gradeCounter != 0 )
47      {
48          average = total / gradeCounter;
49
50          // display average of exam grades
51          document.writeln(
52              "<h1>Class average is " + average + "</h1>" );
53      } // end if
54      else
55          document.writeln( "<p>No grades were entered</p>" );
56
57      </script>
58  </head><body></body>
59 </html>
```

**Fig. 7.9** | Sentinel-controlled repetition to calculate a class average.  
(Part 3 of 4.)



This dialog is displayed four times.  
User input is 97, 88, 72 and -1.



**Fig. 7.9** | Sentinel-controlled repetition to calculate a class average.  
(Part 4 of 4.)



## 7.10 Formulating Algorithms: Nested Control Statements

- ▶ Control structures may be nested inside of one another



```
1 Initialize passes to zero
2 Initialize failures to zero
3 Initialize student to one
4
5 While student counter is less than or equal to ten
6   Input the next exam result
7
8   If the student passed
9     Add one to passes
10  Else
11    Add one to failures
12  Add one to student counter
13
14 Print the number of passes
15 Print the number of failures
16
17 If more than eight students passed
18   Print "Bonus to Instructor!"
```

**Fig. 7.10 |** Examination-results problem pseudocode.



```
1  <!DOCTYPE html>
2
3  <!-- Fig. 7.11: analysis.html -->
4  <!-- Examination-results calculation. -->
5  <html>
6      <head>
7          <meta charset = "utf-8">
8          <title>Analysis of Examination Results</title>
9          <script>
10
11             // initializing variables in declarations
12             var passes = 0; // number of passes
13             var failures = 0; // number of failures
14             var student = 1; // student counter
15             var result; // an exam result
16
```

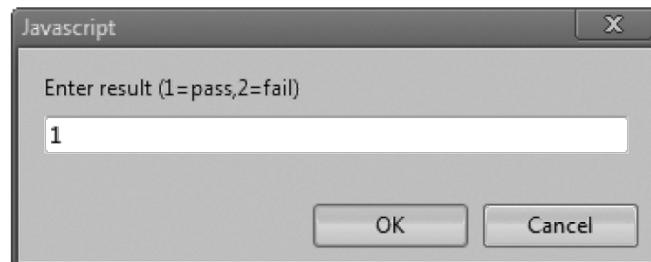
**Fig. 7.11 |** Examination-results calculation. (Part I of 4.)



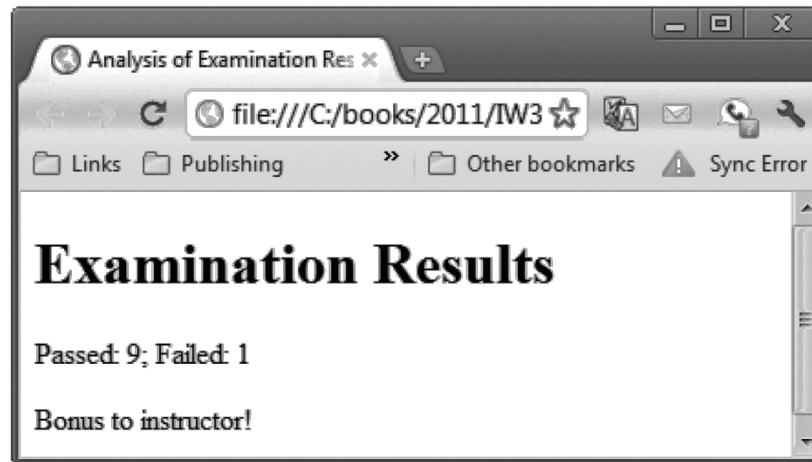
```
17 // process 10 students; counter-controlled loop
18 while ( student <= 10 )
19 {
20     result = window.prompt( "Enter result (1=pass,2=fail)", "0" );
21
22     if ( result == "1" )
23         passes = passes + 1;
24     else
25         failures = failures + 1;
26
27     student = student + 1;
28 } // end while
29
30 // termination phase
31 document.writeln( "<h1>Examination Results</h1>" );
32 document.writeln( "<p>Passed: " + passes +
33                 "; Failed: " + failures + "</p>" );
34
35 if ( passes > 8 )
36     document.writeln( "<p>Bonus to instructor!</p>" );
37
38 </script>
39 </head><body></body>
40 </html>
```

**Fig. 7.11** | Examination-results calculation. (Part 2 of 4.)

a) This dialog is displayed 10 times. User input is 1, 2, 1, 1, 1, 1, 1, 1 and 1.

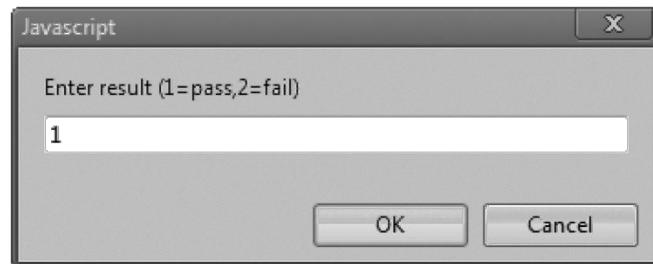


b) Nine students passed and one failed, therefore "Bonus to instructor!" is printed.

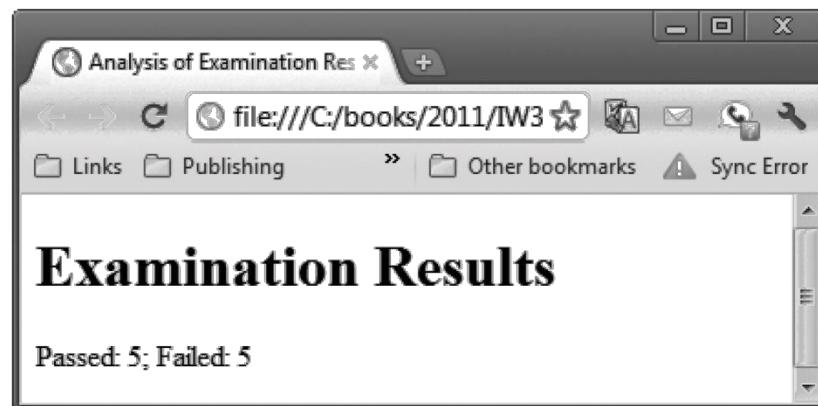


**Fig. 7.11** | Examination-results calculation. (Part 3 of 4.)

- c) This dialog is displayed 10 times. User input is 1, 2, 1, 2, 2, 1, 2, 2, 1 and 1.



- d) Five students passed and five failed, so no bonus is paid to the instructor.



**Fig. 7.11** | Examination-results calculation. (Part 4 of 4.)



## Good Programming Practice 7.4

When inputting values from the user, validate the input to ensure that it's correct. If an input value is incorrect, prompt the user to input the value again. The HTML5 self-validating controls can help you check the formatting of your data, but you may need additional tests to check that properly formatted values make sense in the context of your application.



## 7.11 Assignment Operators

- ▶ JavaScript provides the arithmetic assignment operators `+=`, `-=`, `*=`, `/=` and `%=`, which abbreviate certain common types of expressions.



Assignment operator	Initial value of variable	Sample expression	Explanation	Assigns
<code>+=</code>	<code>c = 3</code>	<code>c += 7</code>	<code>c = c + 7</code>	10 to c
<code>-=</code>	<code>d = 5</code>	<code>d -= 4</code>	<code>d = d - 4</code>	1 to d
<code>*=</code>	<code>e = 4</code>	<code>e *= 5</code>	<code>e = e * 5</code>	20 to e
<code>/=</code>	<code>f = 6</code>	<code>f /= 3</code>	<code>f = f / 3</code>	2 to f
<code>%=</code>	<code>g = 12</code>	<code>g %= 9</code>	<code>g = g % 9</code>	3 to g

**Fig. 7.12 |** Arithmetic assignment operators.



## 7.12 Increment and Decrement Operators

- ▶ The increment operator, `++`, and the decrement operator, `--`, increment or decrement a variable by 1, respectively.
- ▶ If the operator is prefixed to the variable, the variable is incremented or decremented by 1, then used in its expression.
- ▶ If the operator is postfix to the variable, the variable is used in its expression, then incremented or decremented by 1.



Operator	Example	Called	Explanation
<code>++</code>	<code>++a</code>	preincrement	Increment <code>a</code> by 1, then use the new value of <code>a</code> in the expression in which <code>a</code> resides.
<code>++</code>	<code>a++</code>	postincrement	Use the current value of <code>a</code> in the expression in which <code>a</code> resides, then increment <code>a</code> by 1.
<code>--</code>	<code>--b</code>	predecrement	Decrement <code>b</code> by 1, then use the new value of <code>b</code> in the expression in which <code>b</code> resides.
<code>--</code>	<code>b--</code>	postdecrement	Use the current value of <code>b</code> in the expression in which <code>b</code> resides, then decrement <code>b</code> by 1.

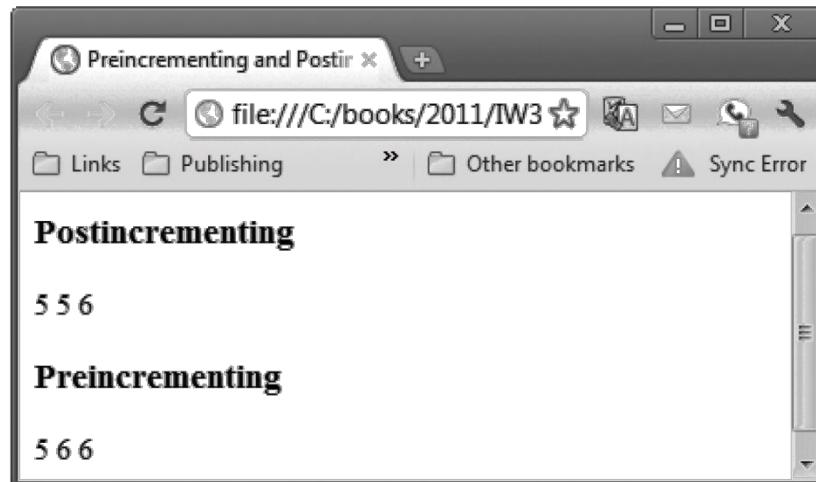
**Fig. 7.13 | Increment and decrement operators.**



```
1 <!DOCTYPE html>
2
3 <!-- Fig. 7.14: increment.html -->
4 <!-- Preincrementing and Postincrementing. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Preincrementing and Postincrementing</title>
9     <script>
10
11       var c;
12
13       c = 5;
14       document.writeln( "<h3>Postincrementing</h3>" );
15       document.writeln( "<p>" + c ); // prints 5
16       // prints 5 then increments
17       document.writeln( " " + c++ );
18       document.writeln( " " + c + "</p>" ); // prints 6
19
```

**Fig. 7.14 | Preincrementing and postincrementing. (Part I of 2.)**

```
20    c = 5;
21    document.writeln( "<h3>Preincrementing</h3>" );
22    document.writeln( "<p>" + c ); // prints 5
23    // increments then prints 6
24    document.writeln( " " + ++c );
25    document.writeln( " " + c + "</p>" ); // prints 6
26
27    </script>
28  </head><body></body>
29 </html>
```



**Fig. 7.14** | Preincrementing and postincrementing. (Part 2 of 2.)



## **Good Programming Practice 7.5**

For readability, unary operators should be placed next to their operands, with no intervening spaces.



## 7.12 Increment and Decrement Operators (Cont.)

- ▶ When incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the same effect, and the predecrement and postdecrement forms have the same effect
- ▶ When a variable appears in the context of a larger expression, preincrementing the variable and postincrementing the variable have different effects. Predecrementing and postdecrementing behave similarly.



## Common Programming Error 7.4

Attempting to use the increment or decrement operator on an expression other than a *left-hand-side expression*—commonly called an *lvalue*—is a syntax error. A left-hand-side expression is a variable or expression that can appear on the left side of an assignment operation. For example, writing `++(x + 1)` is a syntax error, because `(x + 1)` is not a left-hand-side expression.

Operator	Associativity	Type
<code>++ --</code>	right to left	unary
<code>* / %</code>	left to right	multiplicative
<code>+ -</code>	left to right	additive
<code>&lt; &lt;= &gt; &gt;=</code>	left to right	relational
<code>== != === !==</code>	left to right	equality
<code>?:</code>	right to left	conditional
<code>= += -= *= /= %=</code>	right to left	assignment

**Fig. 7.15** | Precedence and associativity of the operators discussed so far.



# JavaScript: Control Statements, Part 2

JavaScript – Introduction



## OBJECTIVES

In this chapter you'll:

- Learn the essentials of counter-controlled repetition
- Use the `for` and `do...while` repetition statements to execute statements in a program repeatedly.
- Perform multiple selection using the `switch` selection statement.
- Use the `break` and `continue` program-control statements
- Use the logical operators to make decisions.



**8.1** Introduction

**8.2** Essentials of Counter-Controlled Repetition

**8.3** `for` Repetition Statement

**8.4** Examples Using the `for` Statement

**8.5** `switch` Multiple-Selection Statement

**8.6** `do...while` Repetition Statement

**8.7** `break` and `continue` Statements

**8.8** Logical Operators

**8.9** Web Resources



## 8.2 Essentials of Counter-Controlled Repetition

- ▶ Counter-controlled repetition requires
  - name of a control variable
  - initial value of the control variable
  - the increment (or decrement) by which the control variable is modified each time through the loop
  - the condition that tests for the final value of the control variable to determine whether looping should continue



## 8.2 Essentials of Counter-Controlled Repetition (Cont.)

- ▶ The double-quote character delimits the beginning and end of a string literal in JavaScript
  - it cannot be used in a string unless it is preceded by a \ to create the escape sequence \"



## 8.2 Essentials of Counter-Controlled Repetition (Cont.)

- ▶ HTML5 allows either single quotes ('') or double quotes ("") to be placed around the value specified for an attribute
- ▶ JavaScript allows single quotes to be placed in a string literal



```
1  <!DOCTYPE html>
2
3  <!-- Fig. 8.1: WhileCounter.html -->
4  <!-- Counter-controlled repetition. -->
5  <html>
6      <head>
7          <meta charset = "utf-8">
8          <title>Counter-Controlled Repetition</title>
9          <script>
10
11             var counter = 1; // initialization
12
13         while ( counter <= 7 ) // repetition condition
14         {
15             document.writeln( "<p style = 'font-size: " +
16                               counter + "ex'>HTML5 font size " + counter + "ex</p>" );
17             ++counter; // increment
18         } //end while
19
20         </script>
21     </head><body></body>
22 </html>
```

**Fig. 8.1** | Counter-controlled repetition. (Part I of 2.)



**Fig. 8.1** | Counter-controlled repetition. (Part 2 of 2.)



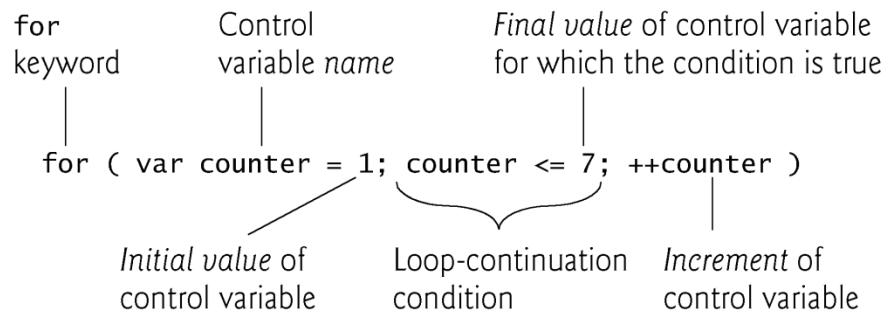
# 8.3 for Repetition Statement

- ▶ **for statement**
  - Specifies each of the items needed for counter-controlled repetition with a control variable
  - Can use a block to put multiple statements into the body
- ▶ If the loop's condition uses a < or > instead of a <= or >=, or vice-versa, it can result in an off-by-one error
- ▶ **for statement header contains three expressions**
  - Initialization
  - Condition
  - Increment Expression
- ▶ The increment expression in the for statement acts like a stand-alone statement at the end of the body of the for statement
- ▶ Place only expressions involving the control variable in the initialization and increment sections of a for statement



```
1 <!DOCTYPE html>
2
3 <!-- Fig. 8.2: ForCounter.html -->
4 <!-- Counter-controlled repetition with the for statement. -->
5 <html>
6   <head>
7     <meta charset="utf-8">
8     <title>Counter-Controlled Repetition</title>
9     <script>
10
11       // Initialization, repetition condition and
12       // incrementing are all included in the for
13       // statement header.
14       for ( var counter = 1; counter <= 7; ++counter )
15         document.writeln( "<p style = 'font-size: " +
16                           counter + "ex'>HTML5 font size " + counter + "ex</p>" );
17
18     </script>
19   </head><body></body>
20 </html>
```

**Fig. 8.2** | Counter-controlled repetition with the for statement.



**Fig. 8.3 |** `for` statement header components.



## 8.3 for Repetition Statement (Cont.)

- ▶ The three expressions in the for statement are optional
- ▶ The two semicolons in the for statement are required
- ▶ The initialization, loop-continuation condition and increment portions of a for statement can contain arithmetic expressions



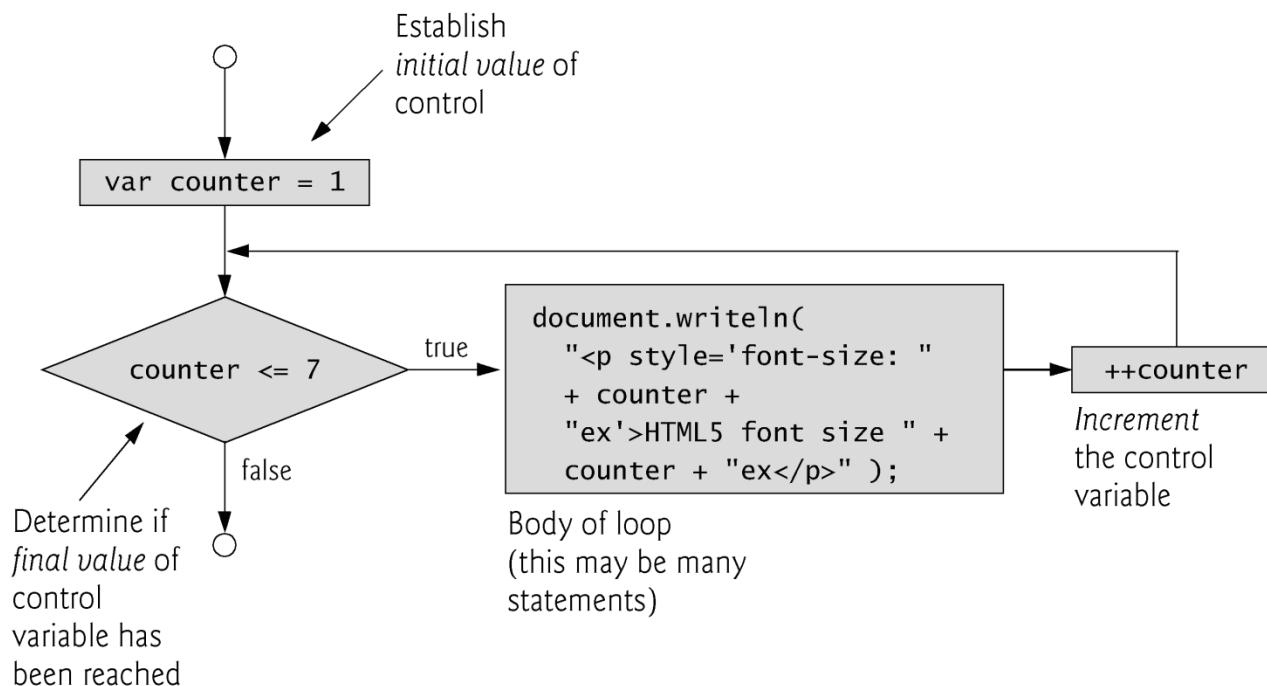
## 8.3 for Repetition Statement (Cont.)

- ▶ The part of a script in which a variable name can be used is known as the variable's scope
- ▶ The “increment” of a for statement may be negative, in which case it is called a decrement and the loop actually counts downward
- ▶ If the loop-continuation condition initially is false, the body of the for statement is not performed
  - Execution proceeds with the statement following the for statement



## Error-Prevention Tip 8.1

Although the value of the control variable can be changed in the body of a `for` statement, avoid changing it, because doing so can lead to subtle errors.



**Fig. 8.4 |** for repetition statement flowchart.



## 8.4 Examples Using the for Statement

- ▶ Figure 8.5 uses the for statement to sum the even integers from 2 to 100.



## Common Programming Error 8.1

Not using the proper relational operator in the loop-continuation condition of a loop that counts downward (e.g., using `i <= 1` instead of `i >= 1` in a loop that counts down to 1) is a logic error.



```
1 <!DOCTYPE html>
2
3 <!-- Fig. 8.5: Sum.html -->
4 <!-- Summation with the for repetition structure. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Sum the Even Integers from 2 to 100</title>
9     <script>
10
11       var sum = 0;
12
13       for ( var number = 2; number <= 100; number += 2 )
14         sum += number;
15
16       document.writeln( "The sum of the even integers " +
17                     "from 2 to 100 is " + sum );
18
19     </script>
20   </head><body></body>
21 </html>
```

**Fig. 8.5** | Summation with the for repetition structure. (Part I of 2.)



**Fig. 8.5** | Summation with the for repetition structure. (Part 2 of 2.)



## Good Programming Practice 8.1

Although statements preceding a `for` statement and in the body of a `for` statement can often be merged into the `for` header, avoid doing so, because it makes the program more difficult to read.



```
1 <!DOCTYPE html>
2
3 <!-- Fig. 8.6: Interest.html -->
4 <!-- Compound interest calculation with a for loop. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Calculating Compound Interest</title>
9     <style type = "text/css">
10    table { width: 300px;
11          border-collapse: collapse;
12          background-color: lightblue; }
13    table, td, th { border: 1px solid black;
14                  padding: 4px; }
15    th { text-align: left;
16          color: white;
17          background-color: darkblue; }
18    tr.oddrow { background-color: white; }
19  </style>
```

**Fig. 8.6** | Compound interest calculation with a for loop. (Part I of 4.)



```
20 <script>
21
22     var amount; // current amount of money
23     var principal = 1000.00; // principal amount
24     var rate = 0.05; // interest rate
25
26     document.writeln("<table>"); // begin the table
27     document.writeln(
28         "<caption>Calculating Compound Interest</caption>");
29     document.writeln(
30         "<thead><tr><th>Year</th>" ); // year column heading
31     document.writeln(
32         "<th>Amount on deposit</th>" ); // amount column heading
33     document.writeln( "</tr></thead><tbody>" );
34
```

**Fig. 8.6** | Compound interest calculation with a for loop. (Part 2 of 4.)



```
35 // output a table row for each year
36 for ( var year = 1; year <= 10; ++year )
37 {
38     amount = principal * Math.pow( 1.0 + rate, year );
39
40     if ( year % 2 !== 0 )
41         document.writeln( "<tr class='oddrow'><td>" + year +
42                         "</td><td>" + amount.toFixed(2) + "</td></tr>" );
43     else
44         document.writeln( "<tr><td>" + year +
45                         "</td><td>" + amount.toFixed(2) + "</td></tr>" );
46 } //end for
47
48 document.writeln( "</tbody></table>" );
49
50 </script>
51 </head><body></body>
52 </html>
```

**Fig. 8.6** | Compound interest calculation with a for loop. (Part 3 of 4.)

The screenshot shows a web browser window with the title "Calculating Compound Interest". The browser interface includes standard buttons for back, forward, and search, along with links and bookmarks. The main content is a table with two columns: "Year" and "Amount on deposit". The data shows the growth of an initial amount over 10 years at a compound interest rate.

Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.63
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

**Fig. 8.6** | Compound interest calculation with a for loop. (Part 4 of 4.)



## 8.4 Examples Using the for Statement (cont.)

- ▶ JavaScript does not include an exponentiation operator
  - Math object's pow method for this purpose.  
`Math.pow(x, y)` calculates the value of  $x$  raised to the  $y$ th power.



# 8.5 switch Multiple-Selection Statement

- ▶ **switch multiple-selection statement**
  - Tests a variable or expression separately for each of the values it may assume
  - Different actions are taken for each value
- ▶ **CSS property list-style-type**
  - Allows you to set the numbering system for a list
  - Possible values include
    - decimal (numbers—the default)
    - lower-roman (lowercase roman numerals)
    - upper-roman (uppercase roman numerals)
    - lower-alpha (lowercase letters)
    - upper-alpha (uppercase letters)
    - others



# 8.5 switch Multiple-Selection Statement (Cont.)

- ▶ **switch statement**
  - Consists of a series of `case` labels and an optional `default` case
  - When control reaches a `switch` statement
    - The script evaluates the controlling expression in the parentheses
    - Compares this value with the value in each of the `case` labels
    - If the comparison evaluates to `true`, the statements after the `case` label are executed in order until a `break` statement is reached
- ▶ The `break` statement is used as the last statement in each case to exit the `switch` statement immediately
- ▶ The `default` case allows you to specify a set of statements to execute if no other case is satisfied
  - Usually the last case in the `switch` statement



## 8.5 switch Multiple-Selection Statement (Cont.)

- ▶ Each case can have multiple actions (statements)
- ▶ Braces are not required around multiple actions in a case of a switch
- ▶ The break statement is not required for the last case because program control automatically continues with the next statement after the switch
- ▶ Having several case labels listed together (e.g., case 1: case 2: with no statements between the cases) executes the same set of actions for each case



```
1 <!DOCTYPE html>
2
3 <!-- Fig. 8.7: SwitchTest.html -->
4 <!-- Using the switch multiple-selection statement. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Switching between HTML5 List Formats</title>
9     <script>
10
11       var choice; // user's choice
12       var startTag; // starting list item tag
13       var endTag; // ending list item tag
14       var validInput = true; // true if input valid else false
15       var listType; // type of list as a string
16
17       choice = window.prompt( "Select a list style:\n" +
18         "1 (numbered), 2 (lettered), 3 (roman numbered)", "1" );
19
```

**Fig. 8.7 |** Using the switch multiple-selection statement. (Part I of 6.)



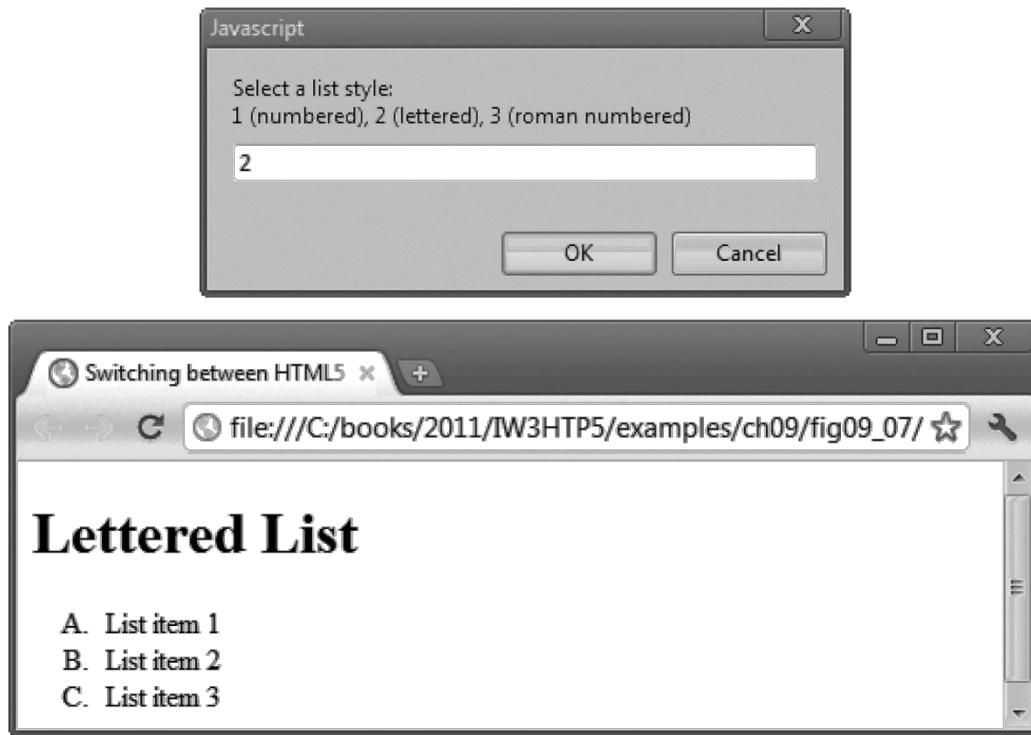
```
20     switch ( choice )
21     {
22         case "1":
23             startTag = "<ol>";
24             endTag = "</ol>";
25             listType = "<h1>Numbered List</h1>";
26             break;
27         case "2":
28             startTag = "<ol style = 'list-style-type: upper-alpha'>";
29             endTag = "</ol>";
30             listType = "<h1>Lettered List</h1>";
31             break;
32         case "3":
33             startTag = "<ol style = 'list-style-type: upper-roman'>";
34             endTag = "</ol>";
35             listType = "<h1>Roman Numbered List</h1>";
36             break;
37         default:
38             validInput = false;
39             break;
40     } //end switch
41
```

**Fig. 8.7 |** Using the `switch` multiple-selection statement. (Part 2 of 6.)

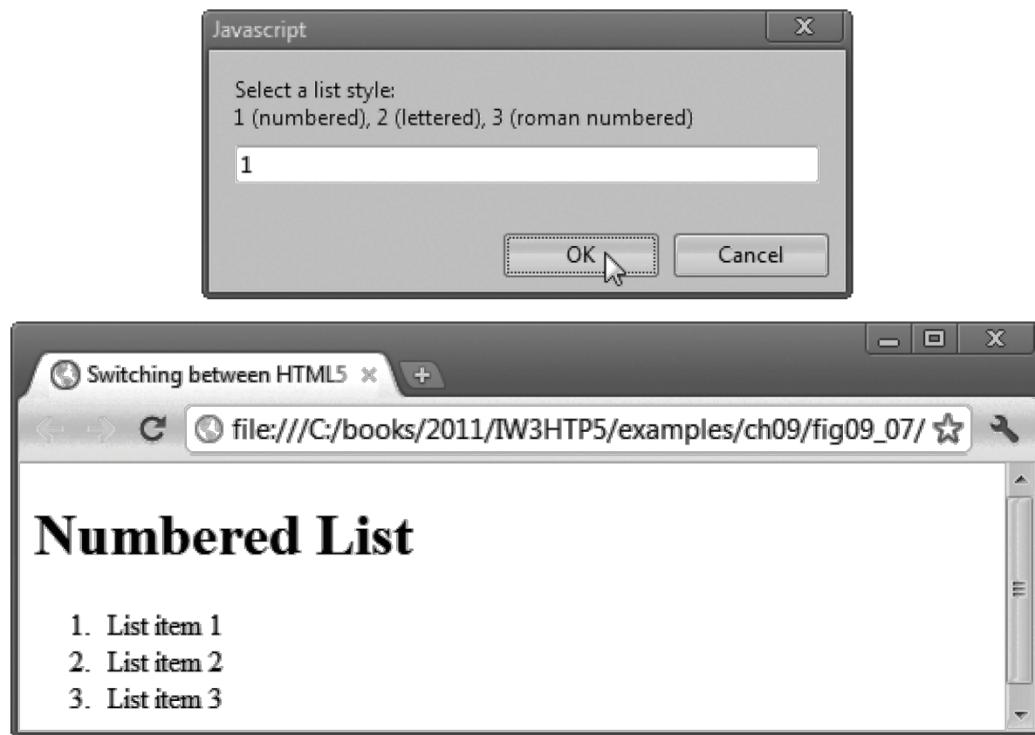


```
42     if ( validInput === true )
43     {
44         document.writeln( listType + startTag );
45
46         for ( var i = 1; i <= 3; ++i )
47             document.writeln( "<li>List item " + i + "</li>" );
48
49         document.writeln( endTag );
50     } //end if
51 else
52     document.writeln( "Invalid choice: " + choice );
53
54     </script>
55     </head><body></body>
56 </html>
```

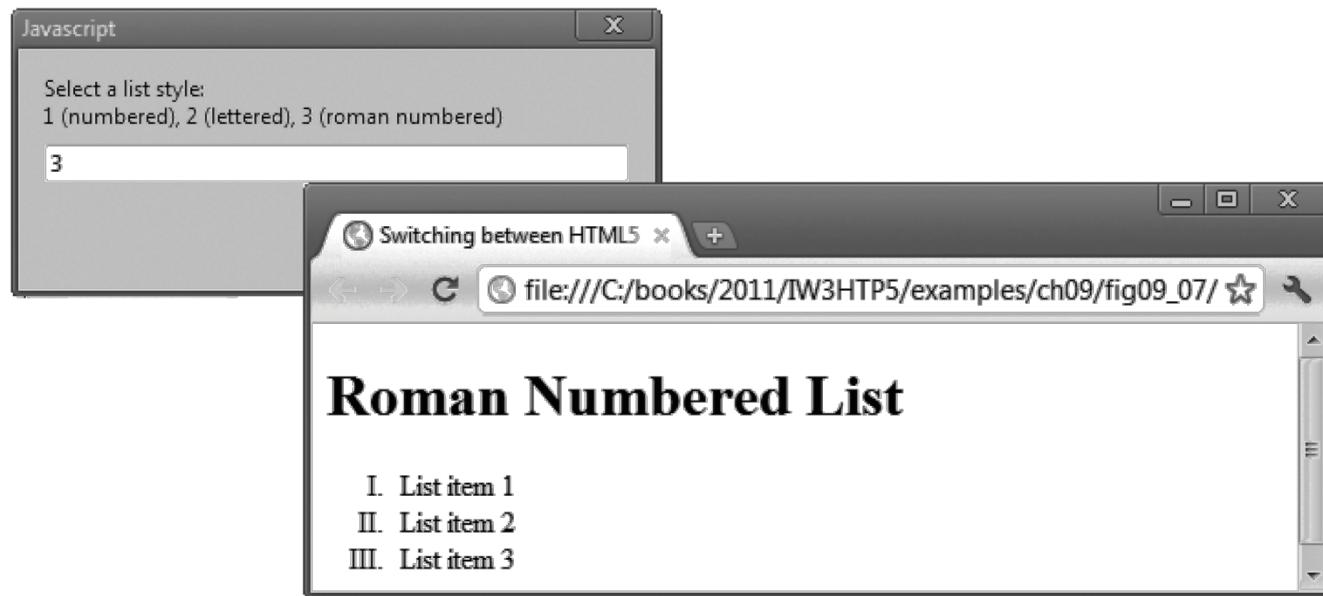
**Fig. 8.7** | Using the switch multiple-selection statement. (Part 3 of 6.)



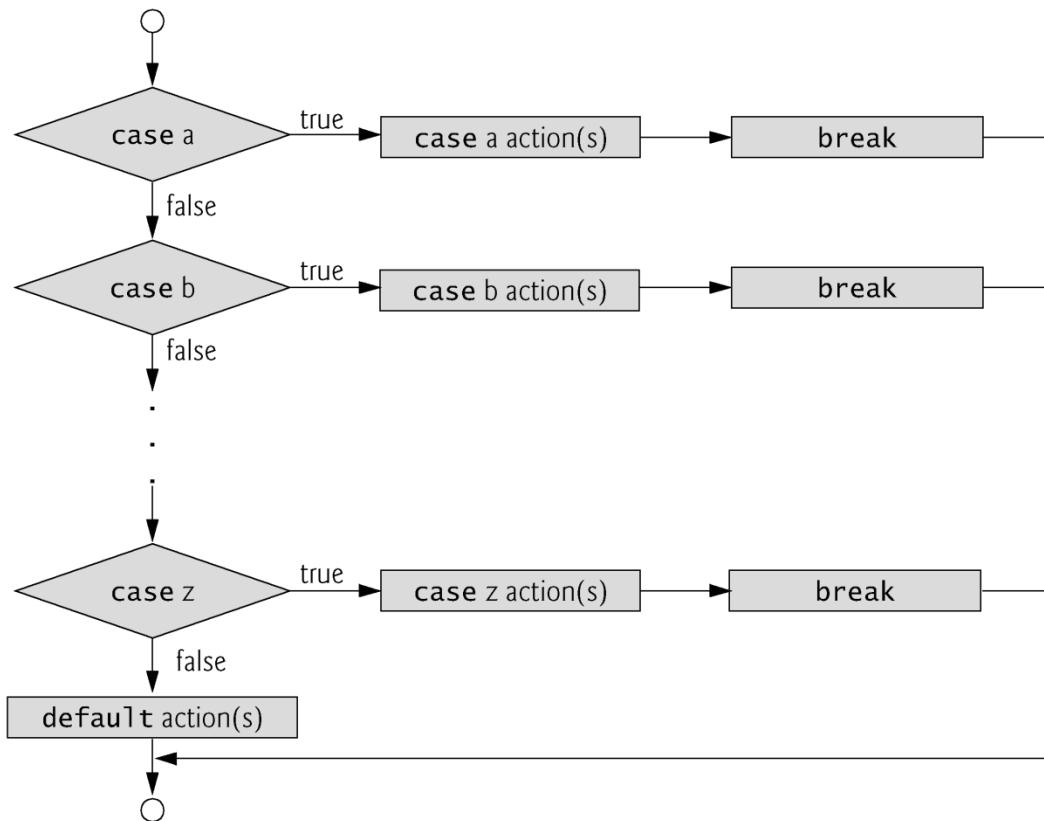
**Fig. 8.7** | Using the `switch` multiple-selection statement. (Part 4 of 6.)



**Fig. 8.7** | Using the switch multiple-selection statement. (Part 5 of 6.)



**Fig. 8.7** | Using the switch multiple-selection statement. (Part 6 of 6.)



**Fig. 8.8 |** switch multiple-selection statement.



# 8.6 do...while Repetition Statement

- ▶ do...while statement
  - tests the loop-continuation condition *after* the loop body executes
  - *The loop body always executes at least once*

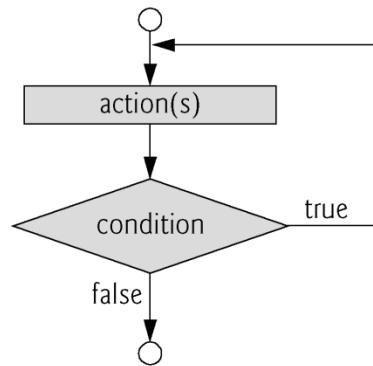


```
1  <!DOCTYPE html>
2
3  <!-- Fig. 8.9: DoWhileTest.html -->
4  <!-- Using the do...while repetition statement. -->
5  <html>
6      <head>
7          <meta charset = "utf-8">
8          <title>Using the do...while Repetition Statement</title>
9          <script>
10
11             var counter = 1;
12
13         do {
14             document.writeln( "<h" + counter + ">This is " +
15                 "an h" + counter + " level head" + "</h" +
16                     counter + ">" );
17             ++counter;
18         } while ( counter <= 6 );
19
20     </script>
21
22     </head><body></body>
23 </html>
```

**Fig. 8.9** | Using the do...while repetition statement. (Part 1 of 2.)



**Fig. 8.9** | Using the do...while repetition statement. (Part 2 of 2.)



**Fig. 8.10** | `do...while` repetition statement flowchart.



## Common Programming Error 8.2

Infinite loops are caused when the loop-continuation condition never becomes `false` in a `while`, `for` or `do...while` statement. To prevent this, make sure that there's not a semicolon immediately after the header of a `while` or `for` statement. In a counter-controlled loop, make sure that the control variable is incremented (or decremented) in the body of the loop. In a sentinel-controlled loop, the sentinel value should eventually be input.



# 8.7 break and continue Statements

- ▶ break statement in a while, for, do...while or switch statement
  - Causes *immediate exit* from the statement
  - Execution continues with the next statement in sequence
- ▶ break statement common uses
  - Escape early from a loop
  - Skip the remainder of a switch statement



## 8.7 break and continue Statements (Cont.)

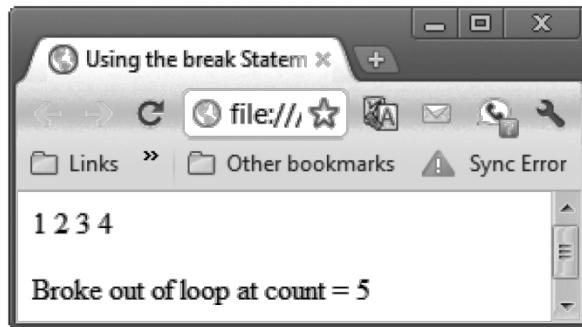
- ▶ **continue statement in a while, for or do...while**
  - skips the remaining statements in the body of the statement and proceeds with the next iteration of the loop
  - In while and do...while statements, the loop-continuation test evaluates immediately after the continue statement executes
  - In for statements, the increment expression executes, then the loop-continuation test evaluates



```
1 <!DOCTYPE html>
2
3 <!-- Fig. 8.11: BreakTest.html -->
4 <!-- Using the break statement in a for statement. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>
9       Using the break Statement in a for Statement
10    </title>
11    <script>
12
13      for ( var count = 1; count <= 10; ++count )
14      {
15        if ( count == 5 )
16          break; // break loop only if count == 5
17
18        document.writeln( count + " " );
19      } //end for
20
```

**Fig. 8.11 |** Using the break statement in a for statement. (Part 1 of 2.)

```
21     document.writeln(  
22         "<p>Broke out of loop at count = " + count + "</p>" );  
23  
24     </script>  
25 </head><body></body>  
26 </html>
```



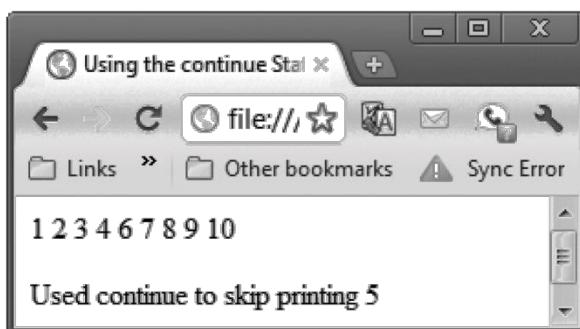
**Fig. 8.11** | Using the break statement in a for statement. (Part 2 of 2.)



```
1 <!DOCTYPE html>
2
3 <!-- Fig. 8.12: ContinueTest.html -->
4 <!-- Using the continue statement in a for statement. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>
9       Using the continue Statement in a for Statement
10    </title>
11
12   <script>
13
14     for ( var count = 1; count <= 10; ++count )
15     {
16       if ( count == 5 )
17         continue; // skip remaining loop code only if count == 5
18
19       document.writeln( count + " " );
20     } //end for
21
```

**Fig. 8.12 |** Using the continue statement in a for statement. (Part 1 of 2.)

```
22     document.writeln( "<p>Used continue to skip printing 5</p>" );
23
24 </script>
25
26 </head><body></body>
27 </html>
```



**Fig. 8.12 |** Using the `continue` statement in a `for` statement. (Part 2 of 2.)

# 8.8 Logical Operators

- ▶ Logical operators can be used to form complex conditions by combining simple conditions
  - `&&` (logical AND)
  - `||` (logical OR)
  - `!` (logical NOT, also called logical negation)
- ▶ The `&&` operator is used to ensure that two conditions are both true before choosing a certain path of execution
- ▶ JavaScript evaluates to `false` or `true` all expressions that include relational operators, equality operators and/or logical operators

<b>expression1</b>	<b>expression2</b>	<b>expression1 &amp;&amp; expression2</b>
false	false	false
false	true	false
true	false	false
true	true	true

**Fig. 8.13** | Truth table for the `&&` (logical AND) operator.

## 8.8 Logical Operators (Cont.)

- ▶ The `||` (logical OR) operator is used to ensure that either or both of two conditions are true before choosing choose a certain path of execution

<b>expression1</b>	<b>expression2</b>	<b>expression1    expression2</b>
false	false	false
false	true	true
true	false	true
true	true	true

**Fig. 8.14** | Truth table for the `||` (logical OR) operator.

## 8.8 Logical Operators (Cont.)

- ▶ The `&&` operator has a higher precedence than the `||` operator
- ▶ Both operators associate from left to right.
- ▶ An expression containing `&&` or `||` operators is evaluated only until truth or falsity is known
  - This is called short-circuit evaluation

## 8.8 Logical Operators (Cont.)

- ▶ ! (logical negation) operator
  - reverses the meaning of a condition (i.e., a true value becomes false, and a false value becomes true)
  - Has only a single condition as an operand (i.e., it is a unary operator)
  - Placed before a condition to evaluate to true if the original condition (without the logical negation operator) is false

expression	! expression
false	true
true	false

**Fig. 8.15 |** Truth table for operator ! (logical negation).



## 8.9 Logical Operators (Cont.)

- ▶ Most nonboolean values can be converted to a boolean true or false value
- ▶ Nonzero numeric values are considered to be true
- ▶ The numeric value zero is considered to be false
- ▶ Any string that contains characters is considered to be true
- ▶ The empty string is considered to be false
- ▶ The value null and variables that have been declared but not initialized are considered to be false
- ▶ All objects are considered to be true

Operator	Associativity	Type
<code>++</code> <code>--</code> <code>!</code>	right to left	unary
<code>*</code> <code>/</code> <code>%</code>	left to right	multiplicative
<code>+</code> <code>-</code>	left to right	additive
<code>&lt;</code> <code>&lt;=</code> <code>&gt;</code> <code>&gt;=</code>	left to right	relational
<code>==</code> <code>!=</code> <code>====</code> <code>!==</code>	left to right	equality
<code>&amp;&amp;</code>	left to right	logical AND
<code>  </code>	left to right	logical OR
<code>?:</code>	right to left	conditional
<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code>	right to left	assignment

**Fig. 8.16** | Precedence and associativity of the operators discussed so far.

# JAVASCRIPT DATA TYPES

*Working with  
Functions,  
Data Types,  
and Operators*

# Objectives



When you complete this chapter, you will be able to:



Use functions to organize your JavaScript code



Use expressions and operators



Identify the order of operator precedence in an expression

# Working with Functions

- Methods
  - Procedures associated with an object
- Functions
  - Related group of JavaScript statements
  - Executed as a single unit
  - Virtually identical to methods
    - Not associated with an object
  - Must be contained within a `script element`



# Defining Functions

- Named function
  - Related statements assigned a name
  - Call, or reference, named function to execute it
- Anonymous function
  - Related statements with no name assigned
  - Work only where they are located in code
- Use named function when you want to reuse code
- Use anonymous function for code that runs only once

# Defining Functions (cont'd.)

- Function definition
  - Lines making up a function
- Named function syntax

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

- Anonymous function syntax

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

# Defining Functions (cont'd.)

- Parameter
  - Variable used within a function
  - Placed within parentheses following a function name
  - Multiple parameters allowed

`calculateVolume(length, width, height)`

# Defining Functions (cont'd.)

- Function statements
  - Do the actual work
  - Contained within function braces
- Put functions in an external .js file
  - Reference at bottom of body section

```
function calculateVolume(length, width, height) {  
    var volume = length * width * height;  
    document.write(volume);  
}
```

# Calling Functions

- To execute a named function:
  - Must invoke, or call, it
- Function call
  - Code calling a function
  - Consists of function name followed by parentheses
    - Contains any variables or values assigned to the function parameters
- Arguments (actual parameters)
  - Variables (values) placed in the function call statement parentheses

# Calling Functions (cont'd.)

- Passing arguments
  - Sending arguments to parameters of a called function
    - Argument value assigned to the corresponding parameter value in the function definition

# Calling Functions (cont'd.)

- Handling events
  - Three options
    - Specify function as value for HTML attribute

```
<input type="submit" onclick="showMessage()" />
```
    - Specify function as property value for object

```
document.getElementById("submitButton").onclick = showMessage;
```
    - Use addEventListener() method

```
var submit = document.getElementById("submitButton");  
submit.addEventListener("click", showMessage, false);
```

# Calling Functions (cont'd.)

- Adding an event listener is most flexible
  - Separates HTML and JavaScript code
  - Can specify several event handlers for a single event
- IE8 requires use of the `attachEvent()` method instead of `addEventListener()` (see Chapter 3)

# Locating Errors with the Browser Console

- Unintentional coding mistakes keep code from working
  - Browsers generate error messages in response
  - Messages displayed in browser console pane
  - Hidden by default to avoid alarming users
- Developers display browser console to see errors

BROWSER	KEYBOARD SHORTCUT	MENU STEPS
Internet Explorer	<b>F12</b> , then <b>Ctrl + 2</b>	Click the <b>Tools</b> button, click <b>F12 Developer Tools</b> on the menu, and then in the window that opens, click the <b>Console</b> button.
Firefox	<b>Ctrl + Shift + K</b> (Win) <b>option + command + K</b> (Mac)	Click the <b>Firefox</b> button (Win) or <b>Tools</b> (Mac or Win), point to <b>Web Developer</b> , and then click <b>Web Console</b> .
Chrome	<b>Ctrl + Shift + J</b> (Win) <b>option + command + J</b> (Mac)	Click the <b>Customize and control Google Chrome</b> button, point to <b>Tools</b> , and then click <b>JavaScript console</b> .

# Locating Errors with the Browser Console (cont'd.)

- Consoles specify a line number with each error

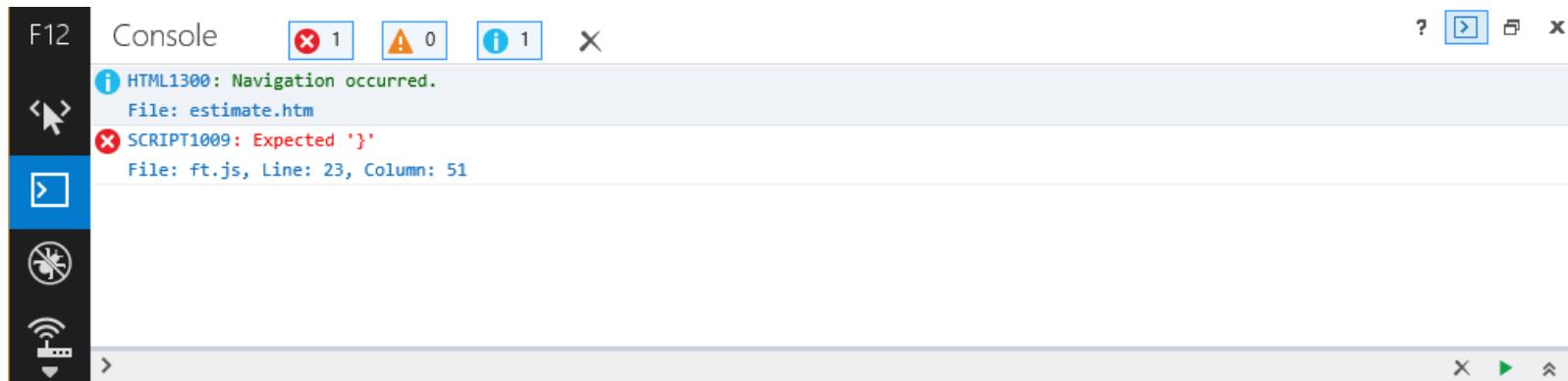


Figure 2-3: Internet Explorer browser console



Figure 2-4: Chrome browser console

# Using Return Statements

- Can return function value to a calling statement
- Return statement
  - Returns a value to the statement calling the function
  - Use the `return` keyword with the variable or value to send to the calling statement
- Example:

```
function averageNumbers(a, b, c) {  
    var sum_of_numbers = a + b + c;  
    var result = sum_of_numbers / 3;  
    return result;  
}
```

# Understanding Variable Scope

- Variable scope
  - Where in code a declared variable can be used
- Global variable
  - Declared outside a function
    - Available to all parts of code
- Local variable
  - Declared inside a function
    - Only available within the function in which it is declared
  - Cease to exist when the function ends
  - Keyword `var` required

# Understanding Variable Scope (cont'd.)

- Good programming technique
  - Always use the `var` keyword when declaring variables
    - Clarifies where and when variable used
- Poor programming technique
  - Declaring a global variable inside of a function by not using the `var` keyword
    - Harder to identify global variables in your scripts

# Understanding Variable Scope (cont'd.)

- If variable declared within a function and does not include the `var` keyword
  - Variable automatically becomes a global variable
- Program may contain global and local variables with the same name
  - Local variable takes precedence
  - Value assigned to local variable of the same name
    - Not assigned to global variable of the same name

# Understanding Variable Scope (cont'd.)

```
var color = "green";
function duplicateVariableNames() {
    let color = "purple";
    document.write(color);
    // value printed is purple
}
duplicateVariableNames();
document.write(color);
// value printed is green
```

# Using Built-in JavaScript Functions

- Called the same way a custom function is called

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

**Table 2-2** Built-in JavaScript functions

# Working with Data Types

- Data type
  - Specific information category a variable contains
- Primitive types
  - Data types assigned 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 <code>true</code> or <code>false</code>
string	Text such as "Hello World"
<code>undefined</code>	An unassigned, undeclared, or nonexistent value
<code>null</code>	An empty value

**Table 2-3** Primitive JavaScript data types

# Working with Data Types (cont'd.)

- The `null` value: data type and a value
  - Can be assigned to a variable
  - Indicates no usable value
  - Use: ensure a variable does not contain any data
- Undefined variable
  - Never had a value assigned to it, has not been declared, or does not exist
  - Indicates variable never assigned a value: not even `null`
  - Use: determine if a value being used by another part of a script

# Working with Data Types (cont'd.)

```
var stateTax;  
document.write(stateTax);  
stateTax = 40;  
document.write(stateTax);  
stateTax = null;  
document.write(stateTax);
```

undefined  
40  
null

**Figure 2-7** Variable assigned values of undefined and null

# Working with Data Types (cont'd.)

- Strongly typed programming languages
  - Require declaration of the data types of variables
  - Strong typing also known as static typing
    - Data types do not change after declared
- Loosely typed programming languages
  - Do not require declaration of the data types of variables
  - Loose typing also known as dynamic typing
    - Data types can change after declared

# Working with Data Types (cont'd.)

- JavaScript interpreter automatically determines data type stored in a variable
- Examples:

```
diffTypes = "Hello World"; // String
diffTypes = 8;           // Integer number
diffTypes = 5.367;        // Floating-point number
diffTypes = true;         // Boolean
diffTypes = null;         // Null
```

# Understanding Numeric Data Types

- JavaScript supports two numeric data types
  - Integers and floating-point numbers
- Integer
  - Positive or negative number with no decimal places
- Floating-point number
  - Number containing decimal places or written in exponential notation
  - Exponential notation (scientific notation)
    - Shortened format for writing very large numbers or numbers with many decimal places

# Using Boolean Values

- Logical value of true or false
  - Used for decision making
    - Which parts of a program should execute
  - Used for comparing data
- JavaScript programming Boolean values
  - The words `true` and `false`
    - JavaScript converts `true` and `false` values to the integers 1 and 0 when necessary

# Using Boolean Values (cont'd.)

```
1 var newCustomer = true;  
2 var contractorRates = false;  
3 document.write("<p>New customer: " + newCustomer + "</p>");  
4 document.write("<p>Contractor rates: " + contractorRates +  
5 "</p>");
```

New customer: true  
Contractor rates: false

**Figure 2-9** Boolean values

# Working with Strings

- Text string
  - Contains zero or more characters
    - Surrounded by double or single quotation marks
  - Can be used as literal values or assigned to a variable
- Empty string
  - Zero-length string value
  - Valid for literal strings
    - Not considered to be null or undefined

# Working with Strings (cont'd.)

- To include a quoted string within a literal string surrounded by double quotation marks
  - Surround the quoted string with single quotation marks
- To include a quoted string within a literal string surrounded by single quotation marks
  - Surround the quoted string with double quotation marks
- String must begin and end with the same type of quotation marks

# Working with Strings (cont'd.)

```
document.write("<h1>Speech at the Berlin Wall<br>
(excerpt)</h1>");
document.write("<p>Two thousand years ago, the proudest boast<br>
was 'civis Romanus sum.'<br />");
document.write('Today, in the world of freedom, the proudest<br>
boast is "Ich bin ein Berliner."</p>');
var speaker = "<p>John F. Kennedy</p>";
var date = 'June 26, 1963';
document.write(speaker);
document.write(date);
```

## Speech at the Berlin Wall (excerpt)

Two thousand years ago, the proudest boast was 'civis Romanus sum.'

Today, in the world of freedom, the proudest boast is "Ich bin ein Berliner."

John F. Kennedy

June 26, 1963

**Figure 2-10** String examples in a browser

# Working with Strings (cont'd.)

- String operators
  - Concatenation operator (+): combines two strings

```
var destination = "Honolulu";
var location = "Hawaii";
destination = destination + " is in " + location;
```
- Compound assignment operator (+=): combines two strings

```
var destination = "Honolulu";
destination += " is in Hawaii";
```
- Plus sign
  - Concatenation operator and addition operator

# Working with Strings (cont'd.)

- Escape characters and sequences
  - Escape character
    - Tells the compiler or interpreter that the character that follows has a special purpose
    - In JavaScript, escape character is backslash (\)
  - Escape sequence
    - Escape character combined with other characters
    - Most escape sequences carry out special functions

# Working with Strings (cont'd.)

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
\xXX	Latin-1 character specified by the XX characters, which represent two hexadecimal digits
\uXXXX	Unicode character specified by the XXXX characters, which represent four hexadecimal digits

**Table 2-4** JavaScript escape sequences

# Using Operators to Build Expressions

OPERATOR TYPE	OPERATORS	DESCRIPTION
Arithmetic	addition (+) subtraction (-) multiplication (*) division (/) modulus (%) increment (++) decrement (--) negation (-)	Perform mathematical calculations
Assignment	assignment (=) compound addition assignment (+=) compound subtraction assignment (-=) compound multiplication assignment (*=) compound division assignment (/=) compound modulus assignment (%=)	Assign values to variables
Comparison	equal (==) strict equal (===) not equal (!=) strict not equal (!==) greater than (>) less than (<) greater than or equal (>=) less than or equal (<=)	Compare operands and return a Boolean value

**Table 2-5** JavaScript operator types (*continues*)

# Using Operators to Build Expressions (cont'd.)

OPERATOR TYPE	OPERATORS	DESCRIPTION
Logical	And ( <code>&amp;&amp;</code> ) Or ( <code>  </code> ) Not ( <code>!</code> )	Perform Boolean operations on Boolean operands
String	concatenation ( <code>+</code> ) compound concatenation assignment ( <code>+=</code> )	Perform operations on strings
Special	property access ( <code>.</code> ) array index ( <code>[]</code> ) function call ( <code>()</code> ) comma ( <code>,</code> ) conditional expression ( <code>? :</code> ) <code>delete</code> ( <code>delete</code> ) property exists ( <code>in</code> ) object type ( <code>instanceof</code> ) new object ( <code>new</code> ) data type ( <code>typeof</code> ) <code>void</code> ( <code>void</code> )	Various purposes; do not fit within other operator categories

Table 2-5 JavaScript operator types (cont'd.)

# Using Operators to Build Expressions (cont'd.)

- Binary operator
  - Requires an operand before and after the operator
- Unary operator
  - Requires a single operand either before or after the operator

# Arithmetic Operators

- Perform mathematical calculations
  - Addition, subtraction, multiplication, division
  - Returns the modulus of a calculation
- Arithmetic binary operators

NAME	OPERATOR	DESCRIPTION
Addition	+	Adds two operands
Subtraction	-	Subtracts one operand from another operand
Multiplication	*	Multiplies one operand by another operand
Division	/	Divides one operand by another operand
Modulus	%	Divides one operand by another operand and returns the remainder

**Table 2-6** Arithmetic binary operators

# Arithmetic Operators (cont'd.)

- Arithmetic binary operators (cont'd.)
  - Value of operation on right side of the assignment operator assigned to variable on the left side
  - Example: `arithmeticValue = x + y;`
    - Result assigned to the `arithmeticValue` variable
  - Division operator (/)
    - Standard mathematical division operation
  - Modulus operator (%)
    - Returns the remainder resulting from the division of two integers

# Arithmetic Operators (cont'd.)

```
var divisionResult = 15 / 6;  
var modulusResult = 15 % 6;  
document.write("<p>15 divided by 6 is "<br>  
+ divisionResult + ".</p>"); // prints '2.5'  
document.write("<p>The whole number 6 goes into 15 twice,<br>  
with a remainder of "<math>+ modulusResult + "</p>"); // prints '3'
```

15 divided by 6 is 2.5.

The whole number 6 goes into 15 twice, with a remainder of 3.

**Figure 2-13** Division and modulus expressions

# Arithmetic Operators (cont'd.)

- Arithmetic binary operators (cont'd.)
  - Assignment statement
    - Can include combination of variables and literal values on the right side
    - Cannot include a literal value as the left operand
  - JavaScript interpreter
    - Attempts to convert the string values to numbers
    - Does not convert strings to numbers when using the addition operator

# Arithmetic Operators (cont'd.)

- Prefix operator
  - Placed before a variable
- Postfix operator
  - Placed after a variable

NAME	OPERATOR	DESCRIPTION
Increment	<code>++</code>	Increases an operand by a value of one
Decrement	<code>--</code>	Decreases an operand by a value of one
Negation	<code>-</code>	Returns the opposite value (negative or positive) of an operand

**Table 2-7** Arithmetic unary operators

# Arithmetic Operators (cont'd.)

- Arithmetic unary operators
  - Performed on a single variable using unary operators
  - Increment (++) unary operator: used as prefix operators
    - Prefix operator placed before a variable
  - Decrement (--) unary operator: used as postfix operator
    - Postfix operator placed after a variable
  - Example: `++count;` and `count++;`
    - Both increase the count variable by one, but return different values

# Arithmetic Operators (cont'd.)

```
1  var studentID = 100;  
2  var curStudentID;  
3  curStudentID = ++studentID; // assigns '101'  
4  document.write("<p>The first student ID is " +↔  
5      curStudentID + "</p>");  
6  curStudentID = ++studentID; // assigns '102'  
7  document.write("<p>The second student ID is " +↔  
8      curStudentID + "</p>");  
9  curStudentID = ++studentID; // assigns '103'  
10 document.write("<p>The third student ID is " +↔  
11      curStudentID + "</p>");
```

The first student ID is 101

The second student ID is 102

The third student ID is 103

**Figure 2-14** Output of the prefix version of the student ID script

# Arithmetic Operators (cont'd.)

```
1  var studentID = 100;  
2  var curStudentID;  
3  curStudentID = studentID++; // assigns '100'  
4  document.write("<p>The first student ID is " + curStudentID + "</p>");  
5  
6  curStudentID = studentID++; // assigns '101'  
7  document.write("<p>The second student ID is " + curStudentID + "</p>");  
8  
9  curStudentID = studentID++; // assigns '102'  
10 document.write("<p>The third student ID is " + curStudentID + "</p>");  
11
```

The first student ID is 100

The second student ID is 101

The third student ID is 102

**Figure 2-15** Output of the postfix version of the student ID script

# Assignment Operators

- Used for assigning a value to a variable
- Equal sign (=)
  - Assigns initial value to a new variable
  - Assigns new value to an existing variable
- Compound assignment operators
  - Perform mathematical calculations on variables and literal values in an expression
    - Then assign a new value to the left operand

# Assignment Operators (cont'd.)

NAME	OPERATOR	DESCRIPTION
Assignment	=	Assigns the value of the right operand to the left operand
Compound addition assignment	+=	Combines the value of the right operand with the value of the left operand (if the operands are strings), or adds the value of the right operand to the value of the left operand (if the operands are numbers), and assigns the new value to the left operand
Compound subtraction assignment	-=	Subtracts the value of the right operand from the value of the left operand, and assigns the new value to the left operand
Compound multiplication assignment	*=	Multiplies the value of the right operand by the value of the left operand, and assigns the new value to the left operand
Compound division assignment	/=	Divides the value of the left operand by the value of the right operand, and assigns the new value to the left operand
Compound modulus assignment	%=	Divides the value of the left operand by the value of the right operand, and assigns the remainder (the modulus) to the left operand

**Table 2-8 Assignment operators**

# Assignment Operators (cont'd.)

- `+=` compound addition assignment operator
  - Used to combine two strings and to add numbers
- Examples:

```
1  var x, y;
2  // += operator with string values
3  x = "Hello ";
4  x += "World"; // x changes to "Hello World"
5  // += operator with numeric values
6  x = 100;
7  y = 200;
8  x += y; // x changes to 300
9  // -= operator
10 x = 10;
11 y = 7;
12 x -= y; // x changes to 3
13 // *= operator
14 x = 2;
15 y = 6;
16 x *= y; // x changes to 12
```

# Assignment Operators (cont'd.)

- Examples: (cont'd.)

```
17 // /= operator
18 x = 24;
19 y = 3;
20 x /= y; // x changes to 8
21 // %= operator
22 x = 3;
23 y = 2;
24 x %= y; // x changes to 1
25 // *= operator with a number and a convertible string
26 x = "100";
27 y = 5;
28 x *= y; // x changes to 500
29 // *= operator with a number and a nonconvertible string
30 x = "one hundred";
31 y = 5;
32 x *= y; // x changes to NaN
```

# Comparison and Conditional Operators

- Comparison operators
  - Compare two operands
    - Determine if one numeric value is greater than another
  - Boolean value of true or false returned after compare
- Operands of comparison operators
  - Two numeric values: compared numerically
  - Two nonnumeric values: compared in alphabetical order
  - Number and a string: convert string value to a number
    - If conversion fails: value of false returned

# Comparison and Conditional Operators (cont'd.)

NAME	OPERATOR	DESCRIPTION
Equal	<code>==</code>	Returns true if the operands are equal
Strict equal	<code>===</code>	Returns true if the operands are equal and of the same type
Not equal	<code>!=</code>	Returns true if the operands are not equal
Strict not equal	<code>!==</code>	Returns true if the operands are not equal or not of the same type
Greater than	<code>&gt;</code>	Returns true if the left operand is greater than the right operand
Less than	<code>&lt;</code>	Returns true if the left operand is less than the right operand
Greater than or equal	<code>&gt;=</code>	Returns true if the left operand is greater than or equal to the right operand
Less than or equal	<code>&lt;=</code>	Returns true if the left operand is less than or equal to the right operand

**Table 2-9** Comparison operators

# Comparison and Conditional Operators (cont'd.)

- Conditional operator
  - Executes one of two expressions based on conditional expression results
  - Syntax  
*conditional expression ? expression1 : expression2;*
  - If conditional expression evaluates to true:
    - Then `expression1` executes
  - If the conditional expression evaluates to false:
    - Then `expression2` executes

# Comparison and Conditional Operators (cont'd.)

- Example of conditional operator:

```
var intVariable = 150;  
var result;  
intVariable > 100 ?  
    result = "intVariable is greater than 100" :  
    result = "intVariable is less than or equal to 100";  
  
document.write(result);
```

# Falsy and Truthy Values

- Six falsy values treated like Boolean `false`:
  - `""`
  - `-0`
  - `0`
  - `NaN`
  - `null`
  - `undefined`
- All other values are truthy, treated like Boolean `true`

# Logical Operators

- Compare two Boolean operands for equality

NAME	OPERATOR	DESCRIPTION
And	<code>&amp;&amp;</code>	Returns <code>true</code> if both the left operand and right operand return a value of <code>true</code> ; otherwise, it returns a value of <code>false</code>
Or	<code>  </code>	Returns <code>true</code> if either the left operand or right operand returns a value of <code>true</code> ; if neither operand returns a value of <code>true</code> , then the expression containing the <code>Or   </code> operator returns a value of <code>false</code>
Not	<code>!</code>	Returns <code>true</code> if an expression is <code>false</code> , and returns <code>false</code> if an expression is <code>true</code>

**Table 2-10** Logical operators

# Special Operators

NAME	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	,	Allows you to include multiple expressions in the same statement
Conditional expression	? :	Executes one of two expressions based on the results of a conditional expression
Delete	<code>delete</code>	Deletes array elements, variables created without the <code>var</code> keyword, and properties of custom objects
Property exists	<code>in</code>	Returns a value of <code>true</code> if a specified property is contained within an object
Object type	<code>instanceof</code>	Returns <code>true</code> if an object is of a specified object type
New object	<code>new</code>	Creates a new instance of a user-defined object type or a predefined JavaScript object type
Data type	<code>typeof</code>	Determines the data type of a variable
Void	<code>void</code>	Evaluates an expression without returning a result

**Table 2-11** Special operators

# Special Operators (cont'd.)

RETURN VALUE	RETURNED FOR
number	Integers and floating-point numbers
string	Text strings
boolean	True or false
object	Objects, arrays, and null variables
function	Functions
undefined	Undefined variables

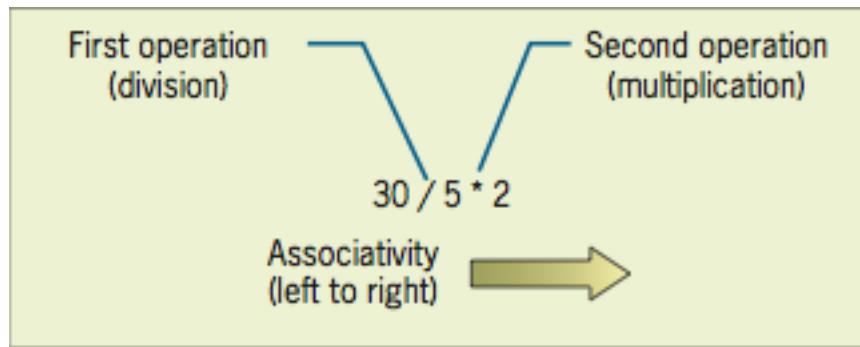
**Table 2-12** Values returned by `typeof` operator

# Understanding Operator Precedence

- Operator precedence
  - Order in which operations in an expression evaluate
- Associativity
  - Order in which operators of equal precedence execute
  - Left to right associativity
  - Right to left associativity

# Understanding Operator Precedence (cont'd.)

- Evaluating associativity
  - Example: multiplication and division operators
    - Associativity of left to right



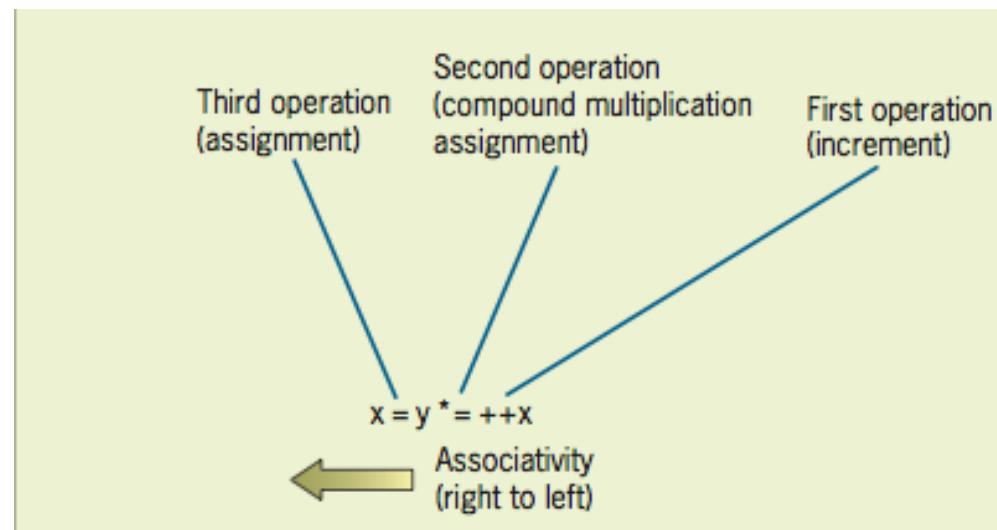
**Figure 2-16** Conceptual illustration  
of left to right associativity

# Understanding Operator Precedence (cont'd.)

- Evaluating associativity (cont'd.)
  - Example: Assignment operator and compound assignment operators
    - Associativity of right to left

`x = y *= ++x`

```
var x = 3;  
var y = 2;  
x = y *= ++x;
```



**Figure 2-17** Conceptual illustration of right-to-left associativity

# Summary

- Functions
  - Similar to methods associated with an object
  - Pass parameters
  - To execute, must be called
- Variable scope
  - Where a declared variable can be used
  - Global and local variables
- Data type
  - Specific category of information a variable contains
  - Static typing and dynamic typing

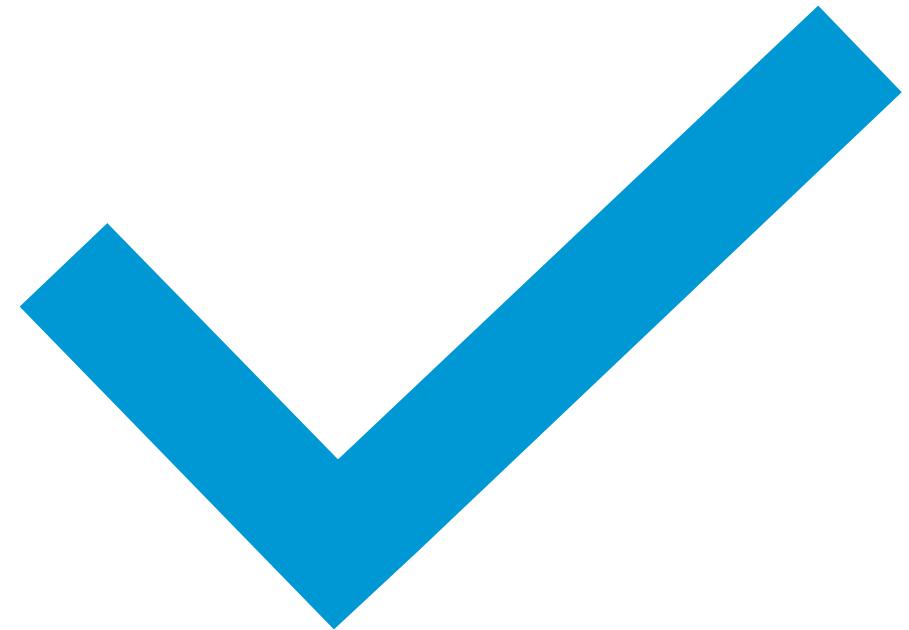
# Summary (cont'd.)

- Numeric data types: integer and floating point
- Boolean values: true and false
- Strings: one or more character surrounded by double or single quotes
  - String operators
  - Escape character
- Operators build expressions
- Operator precedence
  - Order in which operations in an expression are evaluated

JavaScript

# Enhancing and Validating Forms

JavaScript



# Objectives

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

1. Use JavaScript to reference forms and form elements.
2. Retrieve values from selection lists.
3. Retrieve values from option buttons.
4. Format numeric values and currency values based on local standards.
5. Write scripts that respond to form events.
6. Store values in hidden fields.

# Objectives

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

1. Understand how web forms are submitted for validation.
2. Validate web form fields using customized tools.
3. Test a field value against a regular expression.
4. Create a customized validation check for credit card data.
5. Manage the form validation process.

# Exploring Forms and Form Elements (1 of 5)

The screenshot shows a product page for the DigiPot® Multi-Functional Pressure Cooker. The page has a header with the brand name 'COCTURA' and a subtitle 'traditional cooking for the modern age'. Below the header, there's a product summary and a detailed description. A 'Product Order' section is highlighted with several fields annotated with labels:

- model field**: Points to the 'Model' dropdown menu set to '6-Quart (\$129.95) ▾'.
- qty field**: Points to the 'Qty' dropdown menu set to '1 ▾'.
- modelCost field**: Points to the 'modelCost' input field below the quantity dropdown.
- plan field**: Points to the 'plan' input field next to the 'Protection Plan' section, which contains four radio button options:
  - No protection plan (\$0.00)
  - 1-year protection plan (\$11.95)
  - 2-year protection plan (\$15.95)
  - 3-year protection plan (\$19.95)
- planCost field**: Points to the 'planCost' input field below the protection plan options.
- subtotal field**: Points to the 'subtotal' input field.
- salesTax field**: Points to the 'salesTax' input field.
- totalCost field**: Points to the 'totalCost' input field at the bottom.

At the bottom right of the form area is a blue 'add to cart' button.

Figure 6-1 Product order form

# Exploring Forms and Form Elements (2 of 5)

- The forms collection
  - JavaScript organizes forms into the HTML collection: `document.forms`
  - Syntax for referencing, where `n` is the index or `fname` is the value of the form's `name` attribute:  
`document.forms[n]`  
`document.forms[fname]`  
`document.forms.fname`
- Working with form elements
  - Organized into the HTML collection: `form.elements`
  - Syntax for referencing fields, where `ename` is the value of the `name` attribute:  
`form.elements[ename]`  
`form.elements.ename`
  - Reference the associated control using the `document.getElementById()` method

# Exploring Forms and Form Elements (3 of 5)

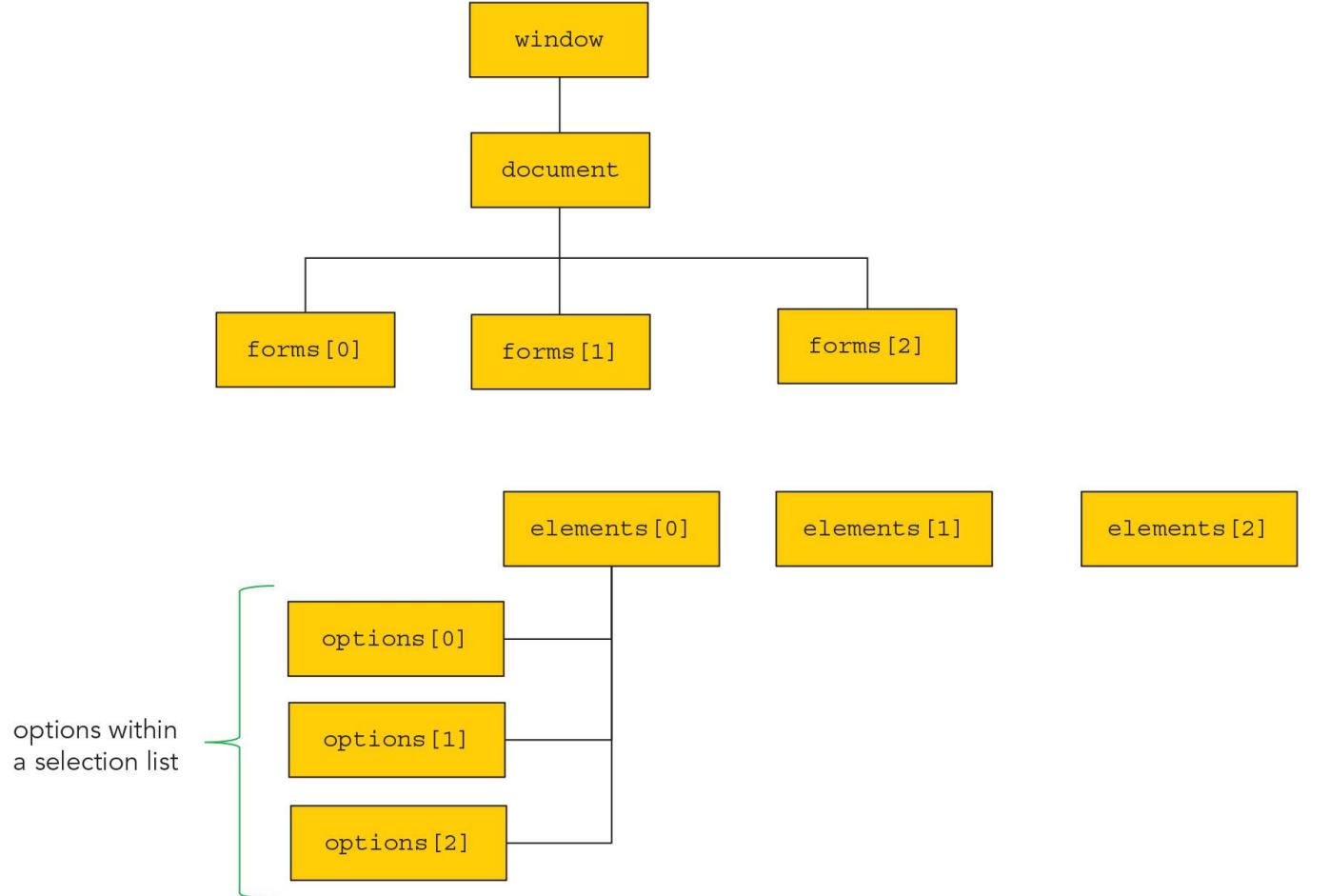


Figure 6-3 Web form hierarchy

# Exploring Forms and Form Elements (4 of 5)

- Properties and methods of `input` elements
  - Every attribute associated with the HTML `<input>` tag is mirrored by a JavaScript property
  - Sample statement to set the value of the `username` field within the `orderForm` web form:  
`document.orderForm.username.value = "John Smith";`
- Navigating between input controls
  - A form control receives the browser's **focus** when it becomes active
  - The `element.focus()` method gives focus to a form element
  - The `element.blur()` method removes focus from an element without reassigning it

# Exploring Forms and Form Elements (5 of 5)

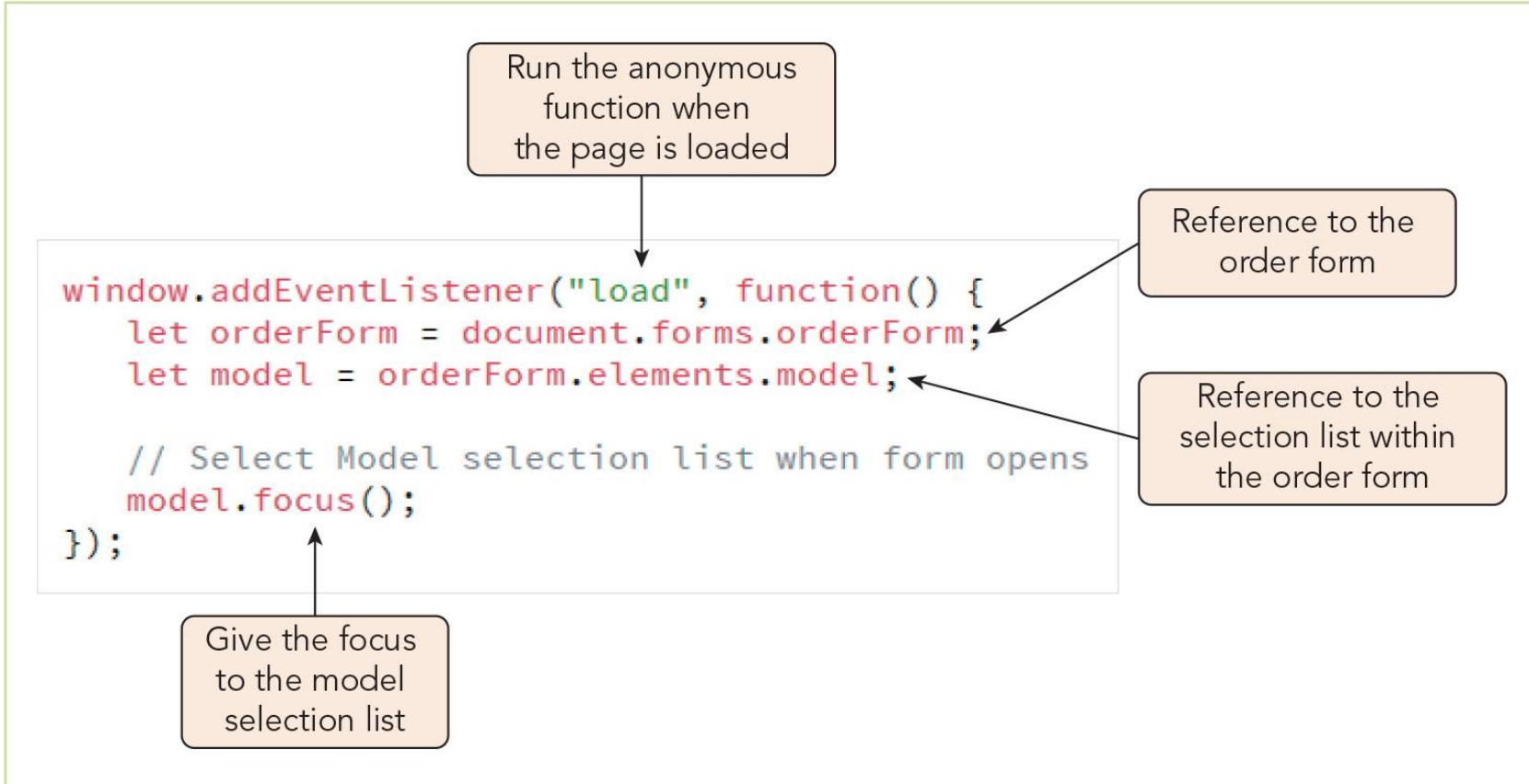


Figure 6-5 Giving the focus to the Model selection list

# Working with Selection Lists (1 of 4)

Property or Method	Description
<code>select.length</code>	The number of options in the selection list, <code>select</code>
<code>select.multiple</code>	Returns true if more than one option can be selected from the list
<code>select.name</code>	The selection list field name
<code>select.options</code>	The object collection of the selection list <code>options</code>
<code>select.selectedIndex</code>	The index number of the currently selected option
<code>select.size</code>	The number of options displayed in the selection list
<code>select.add(option)</code>	Adds <code>option</code> to the selection list
<code>select.remove(index)</code>	Removes the option with the index number, <code>index</code> , from the selection list

# Working with Selection Lists (2 of 4)

Property or Method	Description
<code>option.defaultSelected</code>	Returns true if <code>option</code> is selected by default
<code>option.index</code>	The index number of <code>option</code> within the options collection
<code>option.selected</code>	Returns true if the option has been selected by the user
<code>option.text</code>	The text associated with <code>option</code>
<code>option.value</code>	The field value of <code>option</code>

# Working with Selection Lists (3 of 4)

- To return the value from a selection list field:
  - Determine which option was selected using the `selectedIndex` property
    - Returns -1 if no option is selected
  - Reference the `value` property of the selected option
- Sample code for returning the cost of the product chosen from the model selection list:

```
let mIndex = model.selectedIndex;
let mValue = model.options[mIndex].value;
```
- Selection list with multiple values
  - Create a `for` loop that runs through the options in the list, checking to determine whether the `selected` property is `true`

# Working with Selection Lists (4 of 4)

```
// Select Model selection list when form opens  
model.focus();  
  
Calculate the cost of  
the customer order → // Calculate the cost of the order  
calcOrder();  
  
Retrieve the value of  
the selected model → function calcOrder() {  
    // Determine the selected model  
    let mIndex = model.selectedIndex;  
    let mValue = model.options[mIndex].value;  
  
    // Determine the selected quantity  
    let qIndex = orderForm.elements.qty.selectedIndex;  
    let quantity = orderForm.elements.qty[qIndex].value;  
  
    // Model cost = model cost times quantity  
    let modelCost = mValue*quantity;  
    orderForm.elements.modelCost.value = modelCost;  
}  
  
});
```

Figure 6-8 Calculating  
the cost of models  
ordered

# Working with Option Buttons (1 of 6)

- Option (radio) buttons are grouped into a `form.elements.options` HTML collection when they share the same `name` attribute
- Sample reference to the first option button for the `plan` field:  
`document.forms.orderForm.elements.plan[0]`

Property	Description
<code>option.checked</code>	Boolean value indicating whether the option button, <code>option</code> , is currently checked by the user
<code>option.defaultChecked</code>	Boolean value indicating whether <code>option</code> is checked by default
<code>option.disabled</code>	Boolean value indicating whether <code>option</code> is disabled or not
<code>option.name</code>	The field name associated with <code>option</code>
<code>option.value</code>	The field value association with <code>option</code>

# Working with Option Buttons (2 of 6)

- Locating the checked option

- Sample for loop that stores the value of the selected option in the planValue variable:

```
let orderForm = document.forms.orderForm;
let plan = orderForm.elements.plan;
for (let i = 0; i < plan.length; i++) {
    if (plan[i].checked) {
        planValue = plan[i].value;
        break;
    }
}
```

- Sample code using the querySelector() method to retrieve the checked option's value:

```
let planValue =
document.querySelector('input[name="plan"] :checked').value;
```

# Working with Option Buttons (3 of 6)

```
// Model cost = model cost times quantity  
let modelCost = mValue*quantity;  
orderForm.elements.modelCost.value = modelCost;  
  
// Retrieve the cost of the protection plan  
let planValue = document.querySelector('input[name="plan"]:checked').value;  
  
// Charge the plan to each item ordered  
let planCost = planValue * quantity;  
orderForm.elements.planCost.value = planCost;
```

Value of the plan selected by the customer

Calculate the cost of applying the plan to every item ordered

Figure 6-11 Calculating the cost of the protection plan

# Working with Option Buttons (4 of 6)

```
// Charge the plan to each item ordered
let planCost = planValue * quantity;
orderForm.elements.planCost.value = planCost;

// Calculate the order subtotal
let subtotal = modelCost + planCost;
orderForm.elements.subtotal.value = subtotal;

// Calculate the 5% sales tax
let salesTax = subtotal * 0.05;
orderForm.elements.salesTax.value = salesTax;

// Calculate the total cost of the order
let totalCost = subtotal + salesTax;
orderForm.elements.totalCost.value = totalCost;
```

Calculate the sum of the cost of the model and the cost of the protection plan

Calculate the sales tax applied to the subtotal

Calculate the total cost of the order

Figure 6-12 Calculating the subtotal, sales tax, and total order cost

# Working with Option Buttons (5 of 6)

**Product Order**

Model  Qty  129.95

Protection Plan

No protection plan (\$0.00)  
 1-year protection plan (\$11.95)  
 2-year protection plan (\$15.95)  
 3-year protection plan (\$19.95)

Subtotal 129.95  
Tax (5%) 6.4975  
TOTAL 136.4475

The diagram illustrates the calculation process for the total order cost. It starts with the 'Model' price of 129.95. This value is used to calculate the 'Sum of the model and protection plan costs'. From there, it adds the 'Sales tax on the order' (calculated as 5% of the subtotal) to reach the final 'TOTAL' of 136.4475.

- Cost of the protection plan
- Sum of the model and protection plan costs
- Sales tax on the order
- Total cost of the order

Figure 6-13 Initial order calculations

# Working with Option Buttons (6 of 6)

- Accessing the option label
  - Text associated with the button is part of the `<label>` tag, not part of the `<input>` tag
  - Label is associated with the input control through its `for` attribute, which matches the input control's `id` attribute
  - JavaScript's `labels` node list for any `input` element: `input.labels`
  - **Sample code to retrieve the label text for the No Protection Plan option button:**  
`let noProtection = document.getElementById("plan_0");  
let planLabel = noProtection.labels[0].textContent;`
  - **Sample code to retrieve the label text for the selected option button:**  
`let plan = document.querySelector('input[name="plan"] :checked');  
let planLabel = plan.labels[0].textContent;`
  - HTML code in a label is retrieved using the `innerHTML` property

# Activity 6.1: Knowledge Check

1. Compare the process of retrieving the value of an input box in a web form to that of retrieving the chosen value from a selection list.
2. How can you retrieve the value and the label for the option selected by the user from a group of radio buttons?

# Activity 6.1: Knowledge Check Answers (1 of 2)

1. Compare the process of retrieving the value of an input box in a web form to that of retrieving the chosen value from a selection list.

To retrieve the value of an input box, you can simply reference its `value` property. To retrieve the chosen value for a selection list, on the other hand, you must first identify the selection option used the field's `selectedIndex` property, and then reference the `value` property for the selected option. If the selection list allows multiple selections, then you must iterate through the options to store all the selected options in an array, then iterate over the array to extract the values.

## Activity 6.1: Knowledge Check Answers (2 of 2)

- How can you retrieve the value and the label for the option selected by the user from a group of radio buttons?

One method for retrieving the selected option's value is to iterate over the options to find the checked option, assign its `value` property to a variable, and then exit the loop. A second route is to call the `querySelector()` method on the document, passing in the CSS selector `input [name="name"] :checked` as the argument, and assign the `value` property of the returned reference to a variable.

To retrieve the selected option's label text, you reference its index value in the `labels` node list for that `input` element (which will be 0 if each radio button has just one label). You can assign a reference to the checked `input` element [obtained using the `querySelector()` method] to a variable, then assign the value of `thatVariable.labels[0].textContent` to your label variable.

# Formatting Data Values in a Form (1 of 4)

- The `toFixed()` method
  - Syntax to return a text string of a number rounded to  $n$  decimal places:  
`value.toFixed( $n$ )`
- Formatting values using a locale string
  - Syntax used to format numbers (or dates) with various options:  
`value.toLocaleString(locale, {options})`
  - Sample use of this method to apply local formatting from the user's computer:  
`let total = 14281.478;  
total.toLocaleString(); // returns "14,281.478"`
  - Sample code to format a number as U.S. currency:  
`let total = 14281.5;  
total.toLocaleString("en-US", {style: "currency", currency: "USD"}) // returns "$14,281.50"`

# Formatting Data Values in a Form (2 of 4)

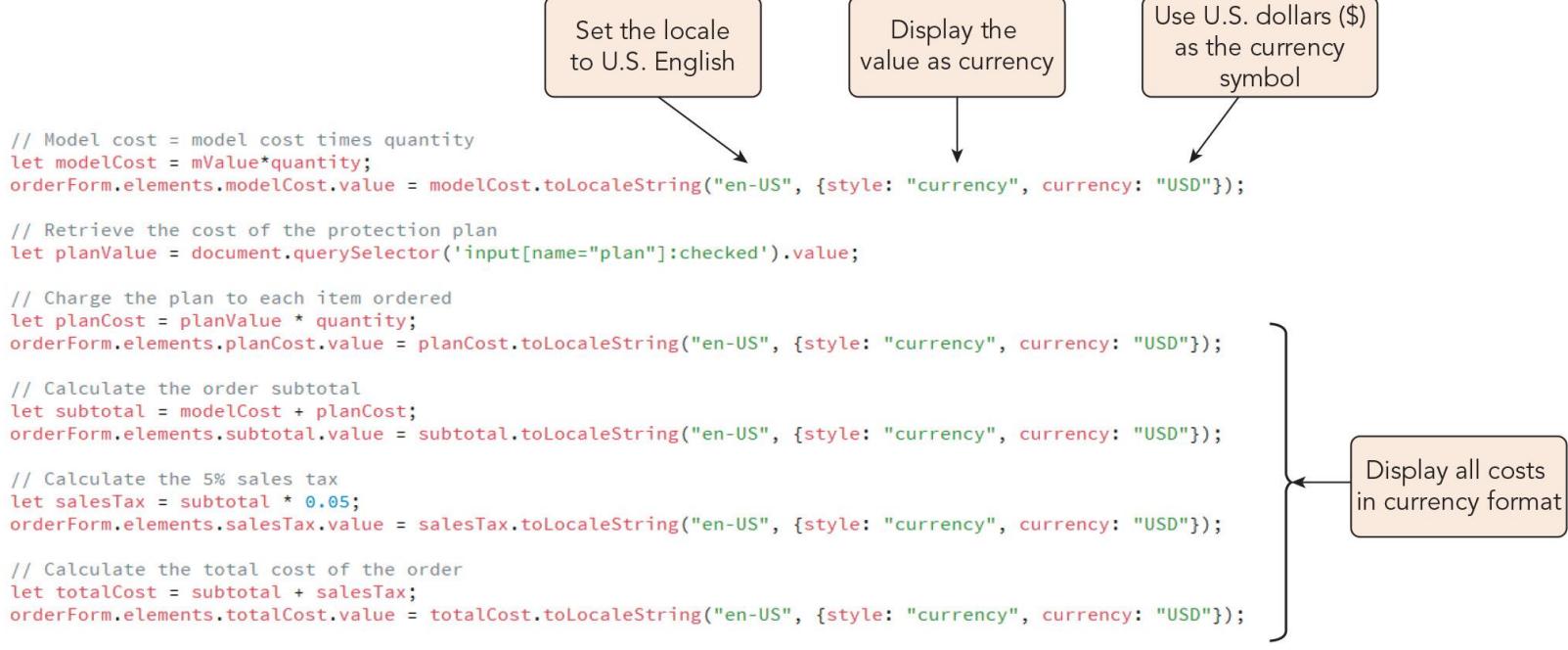


Figure 6-15 Applying the `toLocaleString()` method

# Formatting Data Values in a Form (3 of 4)

The screenshot shows a shopping cart interface with the following details:

- Model:** 6-Quart (\$129.95)
- Qty:** 1
- Subtotal:** \$129.95
- Tax (5%):** \$6.50
- TOTAL:** \$136.45

A callout box labeled "Costs displayed as U.S. currency" points to the Subtotal, Tax, and TOTAL fields.

Figure 6-16 Displaying numeric values as currency

# Formatting Data Values in a Form (4 of 4)

- Making a website international friendly
  - Support international conventions
  - Avoid images that contain text strings
  - Make your layout flexible
  - Optimize your site for international searches
  - Provide currency conversion features
  - Account for cultural differences

# Responding to Form Events (1 of 3)

Event Handler	Description
<code>element.onblur</code>	The form <code>element</code> has lost the focus
<code>element.onchange</code>	The value of <code>element</code> has changed, and <code>element</code> has lost the focus
<code>element.onfocus</code>	The <code>element</code> has received the focus
<code>element.oninput</code>	The <code>element</code> has received user input
<code>element.oninvalid</code>	The <code>element</code> value is invalid
<code>form.onreset</code>	The <code>form</code> has been reset
<code>element.onsearch</code>	The user has entered something into a search field
<code>element.onselect</code>	Text has been selected within the <code>element</code>
<code>form.onsubmit</code>	The <code>form</code> has been submitted

# Responding to Form Events (2 of 3)

```
window.addEventListener("load", function() {
  let orderForm = document.forms.orderForm;
  let model = orderForm.elements.model;

  // Select Model selection list when form opens
  model.focus();

  // Add an event listener for every form element
  for (let i = 0; i < orderForm.elements.length; i++) {
    orderForm.elements[i].addEventListener("change", calcOrder);
  }
}
```

Run the calcOrder()  
function when any  
order form element  
changes its value

Figure 6-18 Adding event listeners to a form

# Responding to Form Events (3 of 3)

**Product Order**

Model  Qty  \$1,279.60

---

Protection Plan

No protection plan (\$0.00)  
 1-year protection plan (\$11.95)  
 2-year protection plan (\$15.95)  
 3-year protection plan (\$19.95) \$159.60

---

Subtotal \$1,439.20  
Tax (5%) \$71.96

---

TOTAL \$1,511.16

Figure 6-19 Order costs recalculated for different customer choices

# Working with Hidden Fields (1 of 2)

- Hidden fields are often used to make data available to the server processing a web form while hiding it from the user
- Sample HTML code for a hidden field:  
`<input type="hidden" id="modelName" name="modelName" />`
- Sample code to store data in a hidden field (`mIndex` is the index of the chosen option):  
`orderForm.elements.modelName.value =  
orderForm.elements.model.options[mIndex].text;`
- **Query string:** a long text string containing field names and values from a web form

# Working with Hidden Fields (2 of 2)

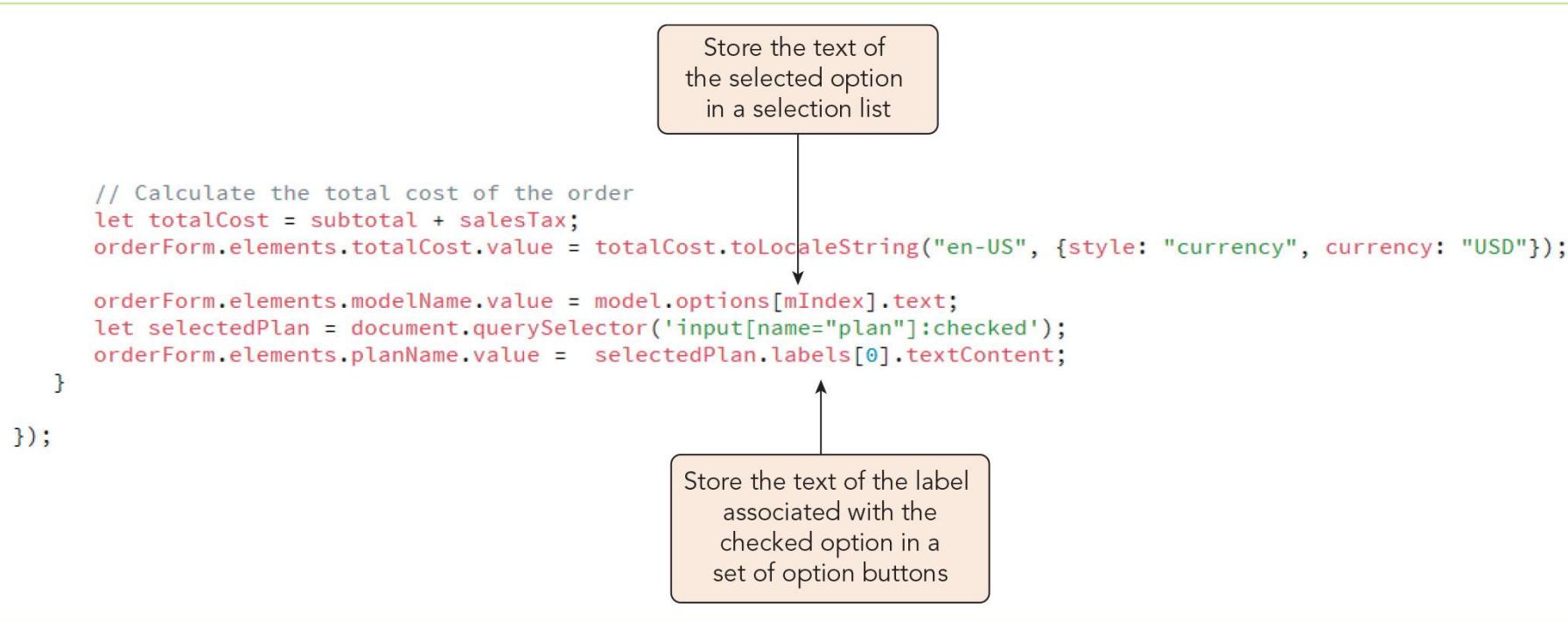


Figure 6-20 Setting the value of hidden fields

# Exploring Form Submission

- Actions within the browser when a submit button (which has the attribute `type="submit"`) is activated:
  - Field values are checked for invalid data
  - If no invalid data is found, a `submit` event is fired
  - If no errors occur in form submission, a request is sent to the server or other resource handling the form data
- Using the `submit` event
  - Use the `form.submit()` method to submit a form without validating data or firing the `submit` event of the `form` object
- Resetting a form
  - Forms are reset when a user clicks a reset button
  - Can also be reset using the `form.reset()` method

# Validating Form Data with JavaScript (1 of 11)

- **Validation**: the process of checking the form for invalid data
  - **Client-side validation** is performed by the user's computer
  - **Server-side validation** is handled by the web server
- **Browser-based validation (native validation)**: validation derived from HTML attributes to restrict what data the user can enter and CSS style rules that highlight data entry errors
- Sample HTML code to mark a required field:  
`<input name="cardName" id="cardName" required type="text" />`
- Sample CSS style rule to highlight invalid data:  
`input:invalid {  
 background-color: rgba(221,147,148,0.2);  
}`

# Validating Form Data with JavaScript (2 of 11)

The screenshot shows a payment form for COCTURA. On the left, there's a summary of the order:

Your Order		
8-Quart (\$159.95)	Qty: 6	\$959.70
1-year protection plan (\$11.95)		\$71.70
Subtotal	\$1,031.40	
Tax (5%)	\$51.57	
TOTAL	\$1,082.97	

On the right is the "Payment Form". It includes fields for Name, Credit Card selection, Credit Card Number, Expiration Date, CVC, and a "Submit Payment" button. A note at the bottom states "\* - Required Item". Arrows point from labels to specific form fields:

- "cardName field" points to the "Name\*" input field.
- "cardNumber field" points to the "Credit Card Number\*" input field.
- "cvc field" points to the "CVC\*" input field.
- "expMonth and expYear fields" points to the dropdown menus for "Expiration Date\* mm" and "yy".
- "credit field" points to a bracket grouping the "Credit Card\*" label and the four radio buttons for American Express, Discover, MasterCard, and Visa.

Figure 6-22 Payment form

# Validating Form Data with JavaScript (3 of 11)

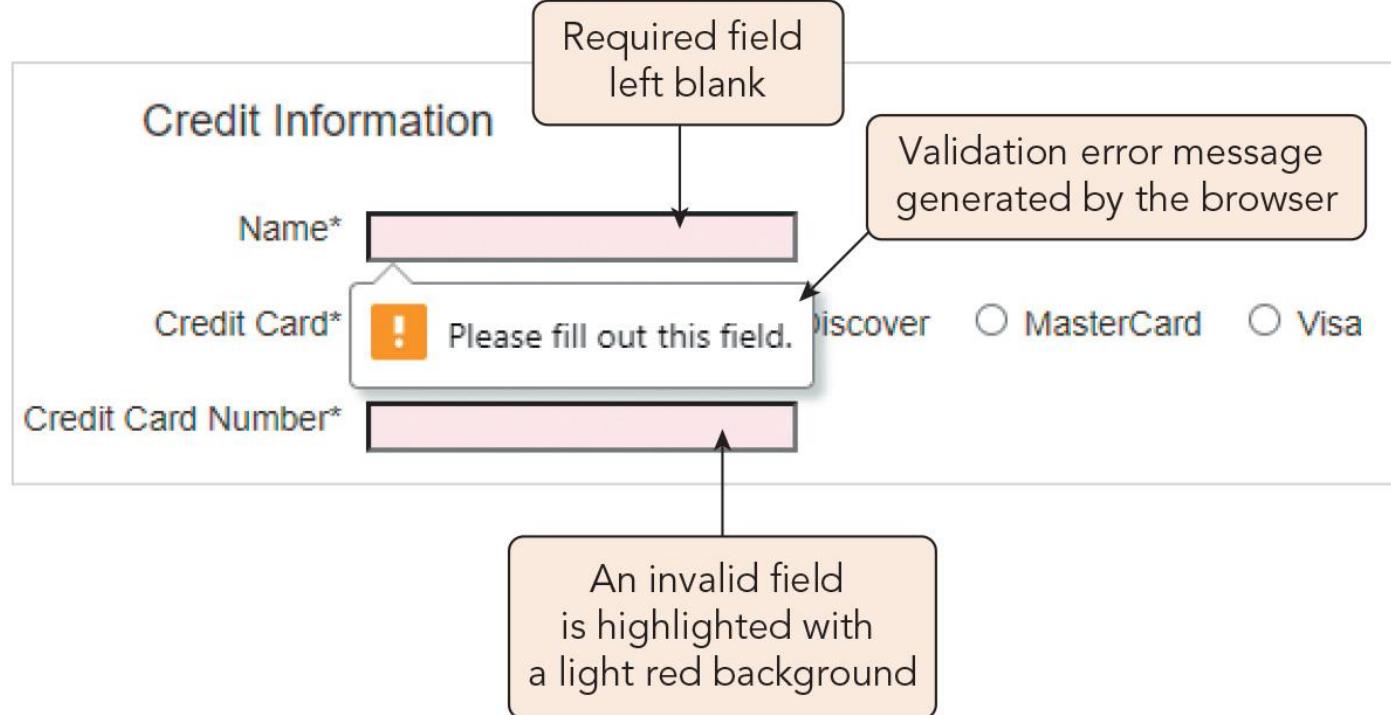


Figure 6-23 Browser validation message and highlighting

# Validating Form Data with JavaScript (4 of 11)

- Limitations of browser-based validation:
  - Validation error message is generic
  - Validation tests are based on a single field value
  - Tests are limited to data field entries; cannot be used for calculated items or functions
- **Constraint Validation API**: form validation properties and methods built into JavaScript
  - Used to supplement native browser tools
- Working with the Constraint Validation API
  - The `valid` property for a field returns `true` if it contains valid data
  - The `checkValidity()` method returns `true` for valid data or returns `false` and fires an `invalid` event for invalid data
    - **Invalid event**: an event that occurs whenever the browser encounters a field whose value does not match the rules specified for its content

# Validating Form Data with JavaScript (5 of 11)

- Exploring the `ValidityState` object
  - The **ValidityState object** stores information about the cause of invalid data and is referenced with the expression `element.validity`
  - Syntax to check whether a field is invalid due to a specific validity state (e.g., `tooLong` or `valueMissing`):  
`element.validity.ValidityState`
- Creating a custom validation message
  - Syntax: `element.setCustomValidity(msg)`
  - Can be used with an `if else` block that displays the custom message for a given invalid validity state or an empty string for a valid state

# Validating Form Data with JavaScript (6 of 11)

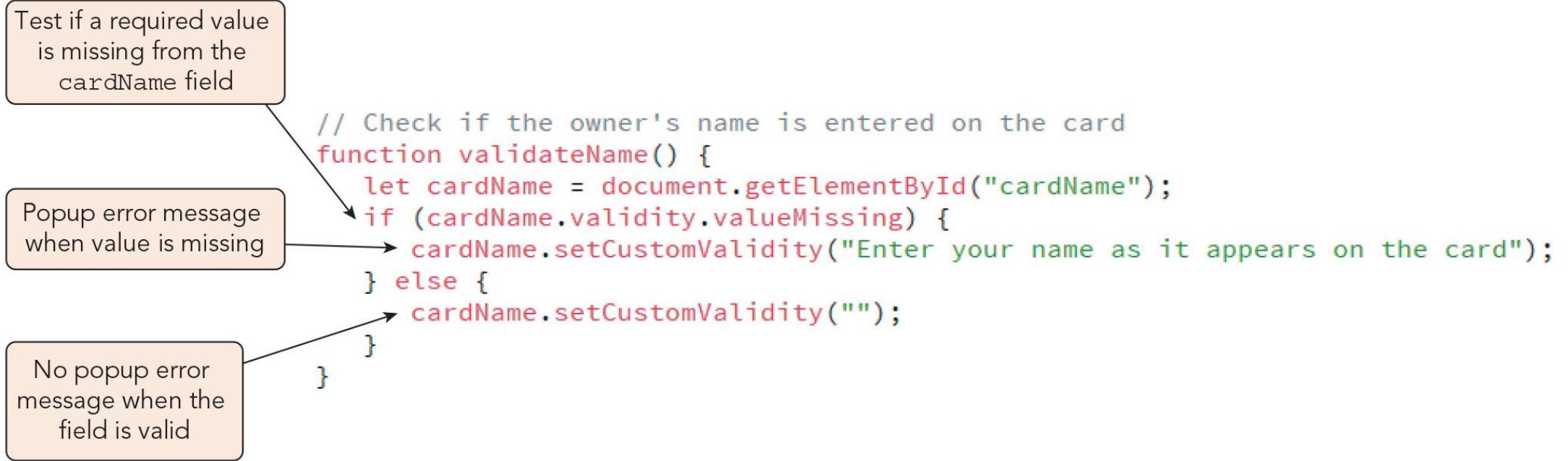


Figure 6-27 Creating the `validateName()` function

# Validating Form Data with JavaScript (7 of 11)

- Responding to invalid data
  - Use event handlers or event listeners:
    - For the `change` event of an input control to check the validity of data upon entry
    - For the `click` event of the form's submit button to check the validity of data on form submission
- Validating data with pattern matching
  - Text string content can be validated against a **regular expression**, concise code describing the general pattern and content of the characters within a text string
  - Useful for numerical IDs with specific patterns, such as credit card numbers
- Validating a selection list
  - If the user must change from a dummy option to a valid selection, you must instruct the browser to declare the field invalid if the dummy option's index is left selected

# Validating Form Data with JavaScript (8 of 11)

```
* /  
  
let subButton = document.getElementById("subButton");  
  
// Validate the payment when the submit button is clicked  
subButton.addEventListener("click", validateName);
```

Run the validateName () function when the submit button is clicked

Figure 6-28 Calling the validateName () function

# Validating Form Data with JavaScript (9 of 11)

```
With option buttons you  
only have to check the  
first option in the list }  
  
// Check if a credit card has been selected  
function validateCard() {  
    let card = document.forms.payment.elements.credit[0];  
    if (card.validity.valueMissing) {  
        → card.setCustomValidity("Select your credit card");  
    } else {  
        → card.setCustomValidity("");  
    }  
}  
  
Popup error message if  
no options are selected  
  
No popup error  
message when the  
field is valid
```

Figure 6-30 Creating the validateCard() function

# Validating Form Data with JavaScript (10 of 11)

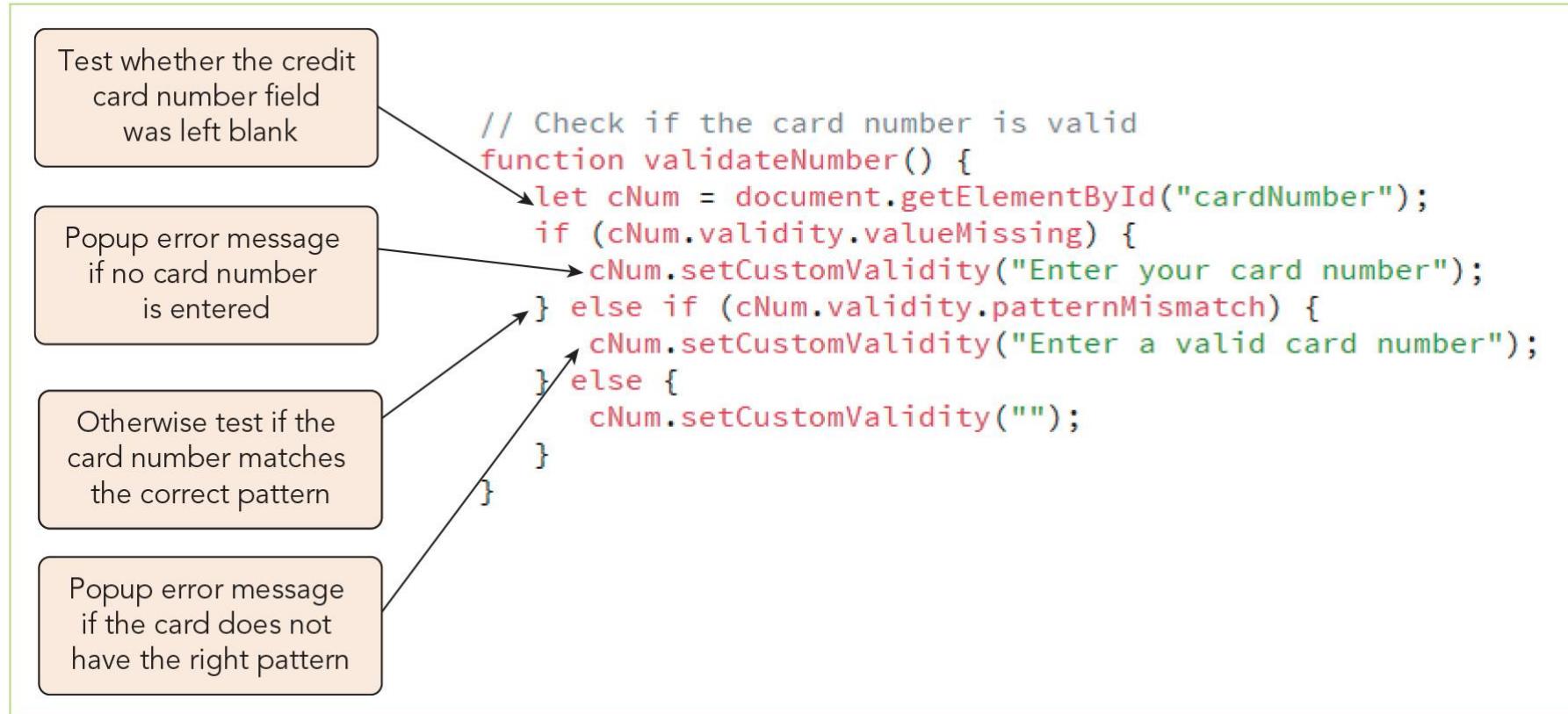


Figure 6-31  
Creating the `validateNumber()` function

# Validating Form Data with JavaScript (11 of 11)

```
// Check that a month is selected for the expiration date
function validateMonth() {
    let month = document.getElementById("expMonth");
    if (month.selectedIndex === 0) {
        month.setCustomValidity("Select the expiration month");
    } else {
        month.setCustomValidity("");
    }
}

// Check that a year is selected for the expiration date
function validateYear() {
    let year = document.getElementById("expYear");
    if (year.selectedIndex === 0) {
        year.setCustomValidity("Select the expiration year");
    } else {
        year.setCustomValidity("");
    }
}
```

If the first option in the selection list is selected, the month is invalid

If the first option in the selection list is selected, the year is invalid

Figure 6-33 Creating the validateMonth() and validateYear() functions

## Activity 6.2: Think, Pair, and Share

1. Form pairs/groups of two to four class members.
2. As a group, choose a purpose/objective for which you might design a short web form—for example, scheduling an appointment, selecting a shirt to purchase, or leaving feedback about services received. Make a list of the fields that the form would require, including the type of input control that you would use to collect the information from the user.
3. Now that you have decided on the fields for your web form, plan how you would ensure that it would submit valid data for processing at the back end. For each field, write notes about the form(s) of validation that you would use and how you would handle invalid data entry by the user.

# Testing a Form Field Against a Regular Expression (1 of 2)

- Syntax to check whether a text string conforms to a regular expression pattern:  
`regExp.test(text)`
- Sample use of the `test()` method to check whether `cardCVC`'s value is a 4-digit number:  
`let cvc = document.getElementById("cvc");  
let isValid = /\d{4}$.test(cvc.value)`

# Testing a Form Field Against a Regular Expression (2 of 2)

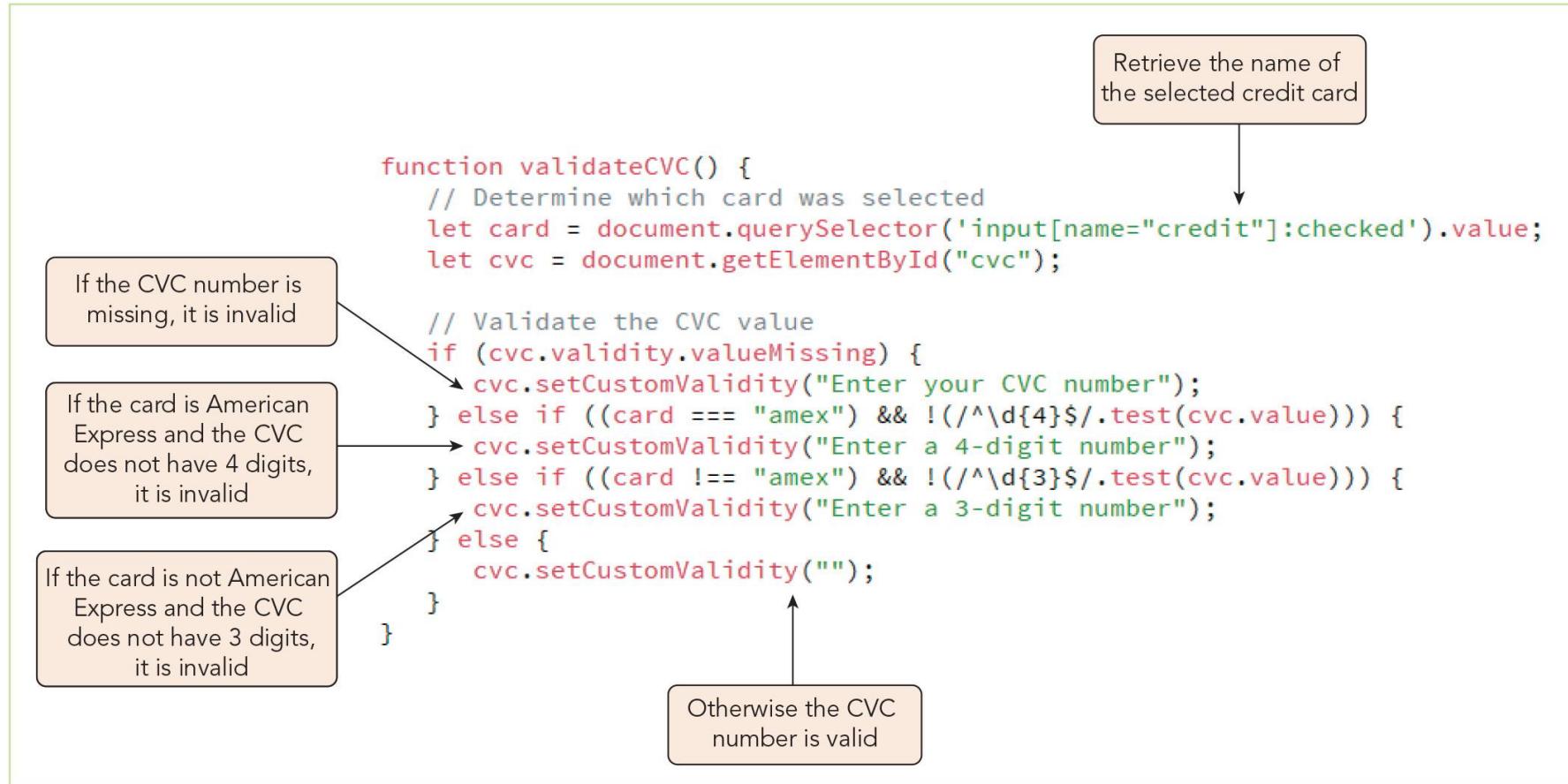


Figure 6-34  
Creating the  
validateCVC ()  
function

# Creating a Custom Validity Check (1 of 2)

- **Checksum algorithm**: an algorithm that sums the digits of a number in a particular way and checks whether the sum satisfies specific mathematical conditions
- **Luhn algorithm (mod 10 algorithm)**: a checksum algorithm that calculates the sum of the digits in a credit card number after doubling every other digit going backwards from the end of the number; for a legitimate card number, the sum is a multiple of 10
- A function that performs the calculations for a checksum algorithm can be used as the basis of a custom validity check

# Creating a Custom Validity Check (2 of 2)

```
// Check if the card number is valid
function validateNumber() {
    let cNum = document.getElementById("cardNumber");
    if (cNum.validity.valueMissing) {
        cNum.setCustomValidity("Enter your card number");
    } else if (cNum.validity.patternMismatch) {
        cNum.setCustomValidity("Enter a valid card number");
    } else if (luhn(cNum.value) === false) {
        cNum.setCustomValidity("Enter a legitimate card number");
    } else {
        cNum.setCustomValidity("");
    }
}
```

If the card number fails the Luhn algorithm, display a validation error

Figure 6-36 Validating the credit card number

# Managing Form Validation (1 of 3)

- You can disable the native browser validation tools and supply a custom validation framework (e.g., to support older browsers that do not provide validation or to avoid the browser's error bubbles)
- Syntax to disable built-in browser validation: `form.noValidate = true;`
- Alternatively, you can add an event listener to run an anonymous function that calls the `preventDefault()` method on the associated event object when it hears an `invalid` event
- **Event object:** the object associated with an event captured by the script

# Managing Form Validation (2 of 3)

- You can also disable the built-in validation tools and write a set of procedures to run in response to the submit event for the web form, e.g.:

```
form.onsubmit = myValidation;  
function myValidation(e) {  
    e.preventDefault();  
    commands to determine if form passes validation  
    if (form is valid) {  
        commands run when form passes validation  
        return true;  
    } else {  
        commands run when form doesn't pass validation  
        return false;  
    }  
}
```

# Managing Form Validation (3 of 3)

- Designing an e-commerce website
  - Provide guests with easy access to your catalog
  - Provide robust search tools
  - Make the purchase process easy to navigate
  - Show discount options and membership deals up front
  - Use validation tests and security measures
  - Incorporate social media

## Activity 6.3: Discussion Questions

1. Describe the properties and methods of the `form` object.
2. Why is the `input` element the most commonly used form element?

# Self-Assessment

1. Do you enjoy working with web forms using HTML, CSS, and/or JavaScript? Why or why not? How likely is it that you will work with forms regularly in your future career?
2. Which, if any, of the validation approaches described in Chapter 6 have you used before? Which do you expect to be in most frequent use in the category or area of web development that most interests you? Why?
3. As a website user, which forms of user notifications related to invalid entries do you find most and least helpful to you when completing and submitting a web form?

# Summary

- Now that the lesson has ended, you should have learned to:
  - Use JavaScript to reference forms and form elements.
  - Retrieve values from selection lists.
  - Retrieve values from option buttons.
  - Format numeric values and currency values based on local standards.
  - Write scripts that respond to form events.
  - Store values in hidden fields.
  - Understand how web forms are submitted for validation.
  - Validate web form fields using customized tools.
  - Test a field value against a regular expression.
  - Create a customized validation check for credit card data.
  - Manage the form validation process.



# JAVASCRIPT ARRAYS

JavaScript Fundamentals



# Objectives

---

---

- Create an array
- Work with array properties and methods
- Store data in arrays
- Create a program loop
- Work with the for loop
- Write comparison and logical operators
- Create a conditional statement
- Use the if statement

# Introducing Arrays

---

---

- Array: Collection of values organized under a single name
- Index: The number that each individual value is associated with and that distinguishes it from other values in the array
- Array values are referenced using the expression  
`array[index]`
- where `array` is the name of the array and `i` is the index of a specific value in the array

# Introducing Arrays

---

---

- Index values start with 0 so that the initial item in an array has an index value of 0
- Second item in the array will have an index value of 1, and so on
- Example

The expression `monthName[4]` references the fifth (not the fourth) item in the `monthName` array

# Creating and Populating an Array

---

---

- To create an array, apply the object constructor
- `var array = new Array(length);`
- where `array` is the name of the array and `length` is the number of items in the array
- The `length` value is optional; if the `length` parameter is omitted then the array expands automatically as more items are added to it

## Introducing Arrays (continued 2)

---

---

- Index values start with 0 so that the initial item in an array has an index value of 0
- Second item in the array will have an index value of 1, and so on
- Example

The expression `monthName[4]` references the fifth (not the fourth) item in the `monthName` array

# Creating and Populating an Array

---

---

- JavaScript represents arrays with the `Array` object
  - Contains a special constructor named `Array()`
- Constructor
  - Special function type used as the basis for creating reference variables
- Syntax

```
var newsSections = new Array(6);
```

- To create an array, apply the object constructor

```
var array = new Array(length);
```

where `array` is the name of the array and `length` is the number of items in the array

- The `length` value is optional; if the `length` parameter is omitted then the array expands automatically as more items are added to it

# Creating and Populating an Array (continued 1)

---

---

- An array can be populated with values by specifying both the array name and the index number of the array item
- Command to set the value of a specific item in an array

```
array[i] = value;
```

where *value* is the value assigned to the array item with the index value *i*

# Referencing Default Collections of Elements

---

---

- `getElementsByName()` method
  - Can reference web page element by looking up all elements of a certain type in document and referencing one element in that collection
  - Resulting collection uses syntax similar to arrays
- Example:
  - `document.getElementsByName("li")[2]`

# Creating and Populating an Array (continued 2)

---

---

- Populate the entire array in a single statement using the following command:

```
var array = new Array(values);
```

where *values* is a comma-separated list of the values in the array

- Example

```
var monthName = new Array("January", "February",
"March", "April", "May", "June", "July",
"August", "September", "October", "November",
"December");
```

# Creating and Populating an Array (continued 3)

---

---

- **Array literal:** Creates an array in which the array values are a comma-separated list within a set of square brackets

```
var array = [values];
```

where *values* are the values of the array

- Example

```
var monthName = ["January", "February", "March",
"April", "May", "June", "July", "August",
"September", "October", "November", "December"];
```

# Creating and Populating an Array (continued 4)

---

---

- Array values need not be of the same data type
- Mix of numeric values, text strings, and other data types within a single array is allowed
- Example

```
var x = ["April", 3.14, true, null];
```

# Working with Array Length

---

---

- JavaScript array automatically expands in length as more items are added
- Apply the following `length` property to determine the array's current size:

`array.length`

where `array` is the name of the array

- Value returned by the `length` property is equal to one more than the highest index number in the array

# Working with Array Length (continued)

---

---

- **Sparse arrays:** Created in JavaScript in which some of the array values are undefined
- The `length` value is not always the same as the number of array values
- Occurs frequently in database applications
- Value of the `length` property cannot be reduced without removing items from the end of the array

# Reversing an Array

---

---

- Items are placed in an array either in the order in which they are defined or explicitly by index number, by default
- JavaScript supports two methods for changing the order of the array items
  - `reverse()`
  - `sort()`
- `reverse()`: Reverses the order of items in an array, making the last items first and the first items last

# Sorting an Array

---

---

- `sort()`: Rearranges array items in alphabetical order
- The `sort()` method when applied to numeric values will sort the values in order by their leading digits, rather than by their numerical values

# *7. Working with the Document Object Model (DOM)*

JAVASCRIPT  
FUNDAMENTALS

# Objectives

---

Access elements by id, tag name, class, name, or selector

Access element content, CSS properties, and attributes

Add and remove document nodes

Create and close new browser tabs and windows with an app

# Objectives (cont'd.)

---

Use the `setTimeout()` and `setInterval()` methods to specify a delay or a duration

Use the History, Location, Navigation, and Screen objects to manipulate the browser window

# Understanding the Browser Object Model and the Document Object Model

---

JavaScript treats web page content as set of related components

- objects

Every element on web page is an object

You can also create objects

- a function is an object

# Understanding the Browser Object Model

---

Browser object model (BOM) or client-side object model

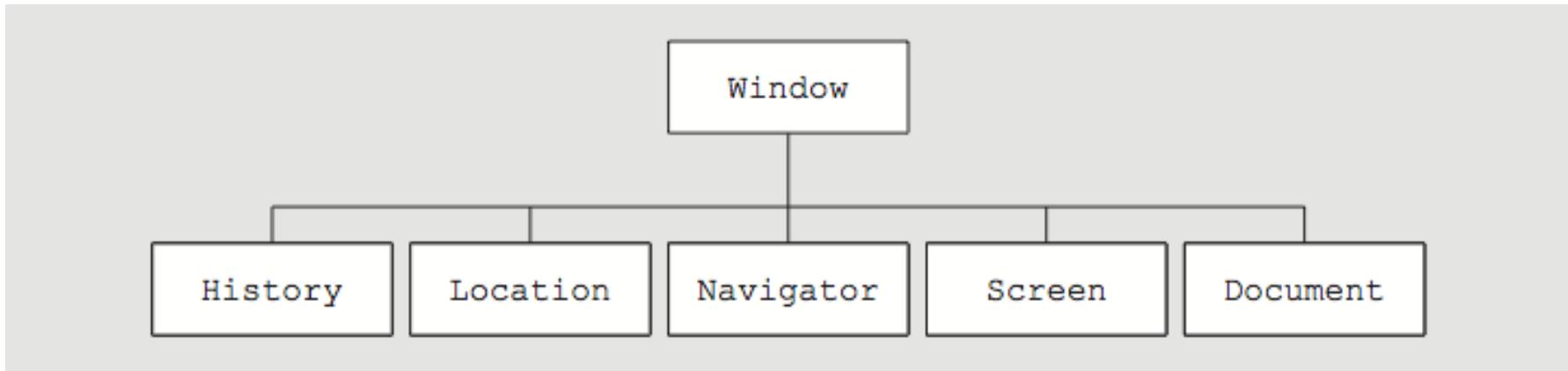
- Hierarchy of objects
- Each provides programmatic access
  - To a different aspect of the web browser window or the web page

Window object

- Represents a Web browser window
- Called the global object
  - Because all other BOM objects contained within it

# Understanding the Browser Object Model (cont'd.)

---



**Figure 5-3** Browser object model

# The Document Object Model

---

## Document object

- Represents the Web page displayed in a browser
- Contains all Web page elements
- JavaScript represents each element by its own object

# The DOM and DHTML

---

## Dynamic HTML (DHTML)

- Interaction can change content of web page without reloading
- Can also change presentation of content
- Combination of HTML, CSS, and JavaScript

## DOM

- example of an application programming interface (API)
- structure of objects with set of properties and methods

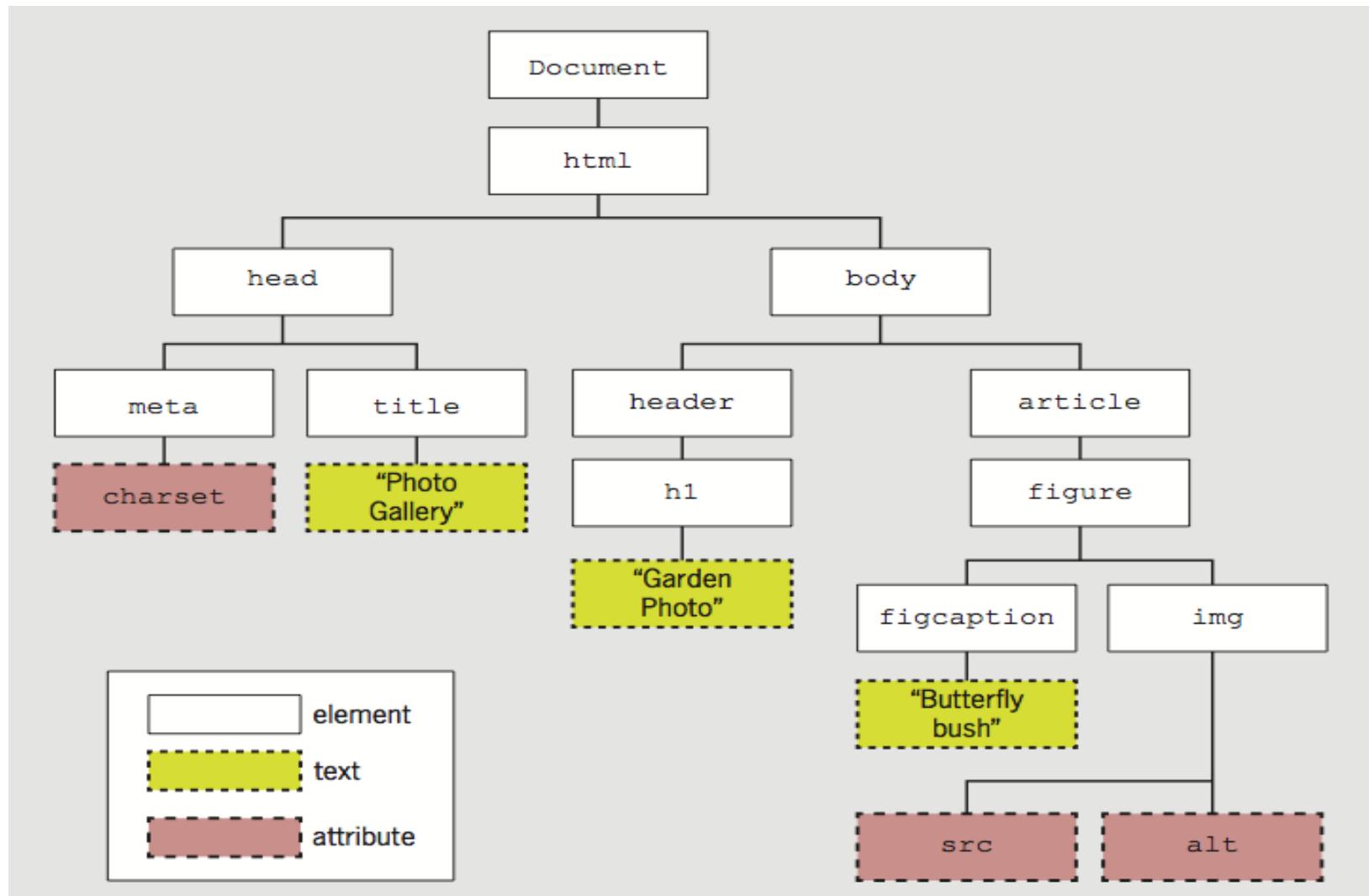
# The DOM tree

The DOM hierarchy depends on a document's contents

---

```
1  <html lang="en">
2      <head>
3          <meta charset="utf-8" />
4          <title>Photo Gallery</title>
5      </head>
6      <body>
7          <header>
8              <h1>Garden Photo</h1>
9          </header>
10         <article>
11             <figure>
12                 <figcaption>Butterfly bush</figcaption>
13                 
14             </figure>
15         </article>
16     </body>
17 </html>
```

# The DOM tree (cont'd.)



**Figure 5-4** Example DOM tree

# The DOM tree

---

Each item in the DOM tree is a node

Element, attribute, and text content nodes are most commonly used

# DOM Document Object Methods

METHOD	DESCRIPTION
<code>getElementById(<i>ID</i>)</code>	Returns the element with the <code>id</code> value <i>ID</i>
<code>getElementsByClassName(<i>class1</i> [<i>class2</i> ...])</code>	If one class name, <code>class1</code> , is specified, returns the collection of elements that belong to <code>class1</code> ; if two or more space-separated class names are specified, the returned collection consists of those elements that belong to all specified class names
<code>getElementsByName(<i>name</i>)</code>	Returns the collection of elements with the name <i>name</i>
<code>getElementsByTagName(<i>tag</i>)</code>	Returns the collection of elements with the tag (element) name <i>tag</i>
<code>querySelectorAll(<i>selector</i>)</code>	Returns the collection of elements that match the CSS selector specified by <i>selector</i>
<code>write(<i>text</i>)</code>	Writes <i>text</i> to the document

**Table 5-1** HTML DOM Document object methods

# DOM Document Object Properties

PROPERTY	DESCRIPTION
body	Document's <code>body</code> element
cookie	Current document's cookie string, which contains small pieces of information about a user that are stored by a web server in text files on the user's computer
domain	Domain name of the server where the current document is located
lastModified	Date the document was last modified
location	Location of the current document, including its URL
referrer	URL of the document that provided a link to the current document
title	Title of the document as specified by the <code>title</code> element in the document head section
URL	URL of the current document

**Table 5-2** Selected DOM Document object properties

# Accessing Document Elements, Content, Properties, and Attributes

---

Methods such as `getElementById()`  
and `getElementsByName()` are  
methods of the Document object

Several methods available for JavaScript to  
reference web page elements

# Accessing Elements by id value

---

Set the `id` value in HTML

`getElementById()` method

- Returns the first element in a document with a matching `id` attribute

Example:

## **HTML element with id value**

```
<input type="number" id="zip" />
```

## **JavaScript to reference HTML element**

```
var zipField = document.getElementById("zip");
```

# Accessing Elements by Tag Name

---

## `getElementsByTagName()` method

- Returns array of elements matching a specified tag name
- Tag name is name of element

Method returns a set of elements

- Node list is indexed collection of nodes
- HTML collection is indexed collection of HTML elements
- Either set uses array syntax

# Accessing Elements by Tag Name (cont'd.)

---

## Example:

- Index numbers start at 0, so second element uses index number 1
- To work with the *second* h1 element on a page:

```
var secondH1 =  
    document.getElementsByTagName("h1")[1];
```

# Accessing Elements by Class Name

---

## `getElementsByClassName()` method

- Returns node list or HTML collection of elements with a class attribute matching a specified value

### Example

- All elements with class value side:

```
var sideElements =  
    document.getElementsByClassName("side");
```

# Accessing Elements by Class Name (cont'd.)

---

class attribute takes multiple values, so  
**getElementsByClassName()** method takes multiple arguments

Arguments enclosed in single set of quotes, with class names separated by spaces

## Example

- All elements with class values side and green:

```
var sideGreenElements =  
    document.getElementsByClassName("side green");
```

# Accessing Elements by Name

---

## `getElementsByName()` method

- Returns node list or HTML collection of elements with a name attribute matching a specified value

Not as useful as preceding options

- But creates more concise code when accessing set of option buttons or check boxes in a form:

```
var colorButtons =  
    document.getElementsByName("color");
```

# Accessing Elements with CSS Selectors

## querySelector () method

---

- References elements using CSS syntax
- Returns first occurrence of element matching a CSS selector

Example:

### **HTML:**

```
<header>
    <h1></h1>
</header>
```

### **JavaScript to reference img element**

```
querySelector("header h1 img")
```

# Accessing Elements with CSS Selectors (cont'd)

## IE8 supports only simple selectors

- Can use different approach in creating CSS selector
- Previous example could be rewritten as

```
querySelector("img.logo")
```

# Accessing Elements with CSS Selectors (cont'd)

## **querySelectorAll()** method

---

- Returns collection of elements matching selector
- Different from `querySelector()` method, which returns only first occurrence
- Example: **HTML:**

```
<nav>
  <ul>
    <li>About Us</li>
    <li>Order</li>
    <li>Support</li>
  </ul>
</nav>
```

**JavaScript to reference all three `li` elements:**

```
querySelectorAll("nav ul li")
```

# Accessing an Element's Content

## `textContent` property

- Accesses and changes text that an element contains
- Unlike `innerHTML`, `textContent` strips out HTML tags

### Example: **HTML:**

```
<ul>
  <li class="topnav"><a href="aboutus.htm">About Us</a></li>
  <li class="topnav"><a href="order.htm">Order</a></li>
  <li class="topnav"><a href="support.htm">Support</a></li>
</ul>
```

### JavaScript to reference and access first `li` element:

```
var button1 = querySelectorAll("li.topNav") [0];
var allContent = button1.innerHTML;
// <a href="aboutus.htm">About Us</a>
var justText = button1.textContent;
// About Us
```

# Accessing an Element's Content (cont'd)

---

**textContent** property is more secure

- Not supported by IE8 or earlier
- Some developers use `if/else` construction to implement `textContent` only on supported browsers

# Accessing an Element's CSS Properties

---

Can access CSS properties through DOM

- Use dot notation
- Reference element's style property followed by name of CSS property
- Example: change value of CSS display property to none for element with id value logo:

```
document.getElementById("logo").style.display =  
    "none";
```

# Accessing an Element's CSS Properties (cont'd.)

---

When CSS property includes hyphen (-), remove hyphen and capitalize letter following hyphen

- Use dot notation
- font-family becomes fontFamily
- Example:

```
var font = document.getElementById("logo").style.fontFamily;
```

CSS value specified using DOM reference is an inline style

- Higher priority than embedded or external styles

# Accessing an Element's CSS Properties (cont'd.)

---

To remove a style you previously added with a DOM reference

- set its value to an empty string
- Example:

```
document.getElementById("navbar").style.color = "";
```

# Accessing Element Attributes

---

Access element attribute with period and name of attribute after element reference

- Reference element with `id` value `homeLink`:

```
document.getElementById("homeLink")
```

- Reference `href` attribute of same element:

```
document.getElementById("homeLink").href
```

Can use to look up attribute value and assign to variable, or to assign new value to attribute

# Accessing Element Attributes (cont'd)

---

One exception for accessing element attributes

- Must use property name `className` to refer to `class` attribute values
- Single class value returned like standard attribute value
- Multiple class values returned in single string, separated by spaces

# Adding and Removing Document Nodes

---

DOM includes methods to change DOM tree

- Can create brand new elements
- Can add/remove elements from DOM tree

# Creating Nodes

---

`createElement()` method

- Creates a new element
- Syntax:

```
document.createElement("element")
```

- *element* is an element name
- Example:
  - To create a new `div` element:

```
document.createElement("div");
```

# Attaching Nodes

---

Newly created node is independent of DOM tree

`appendChild()` method

- Attaches node to DOM tree
- Syntax:

*parentNode.appendChild(childNode)*

- *childNode* is node to be attached
- *parentNode* is node to attach child node to

# Attaching Nodes (cont'd.)

---

## Example:

- Create new li element and attach to element with id value navList:

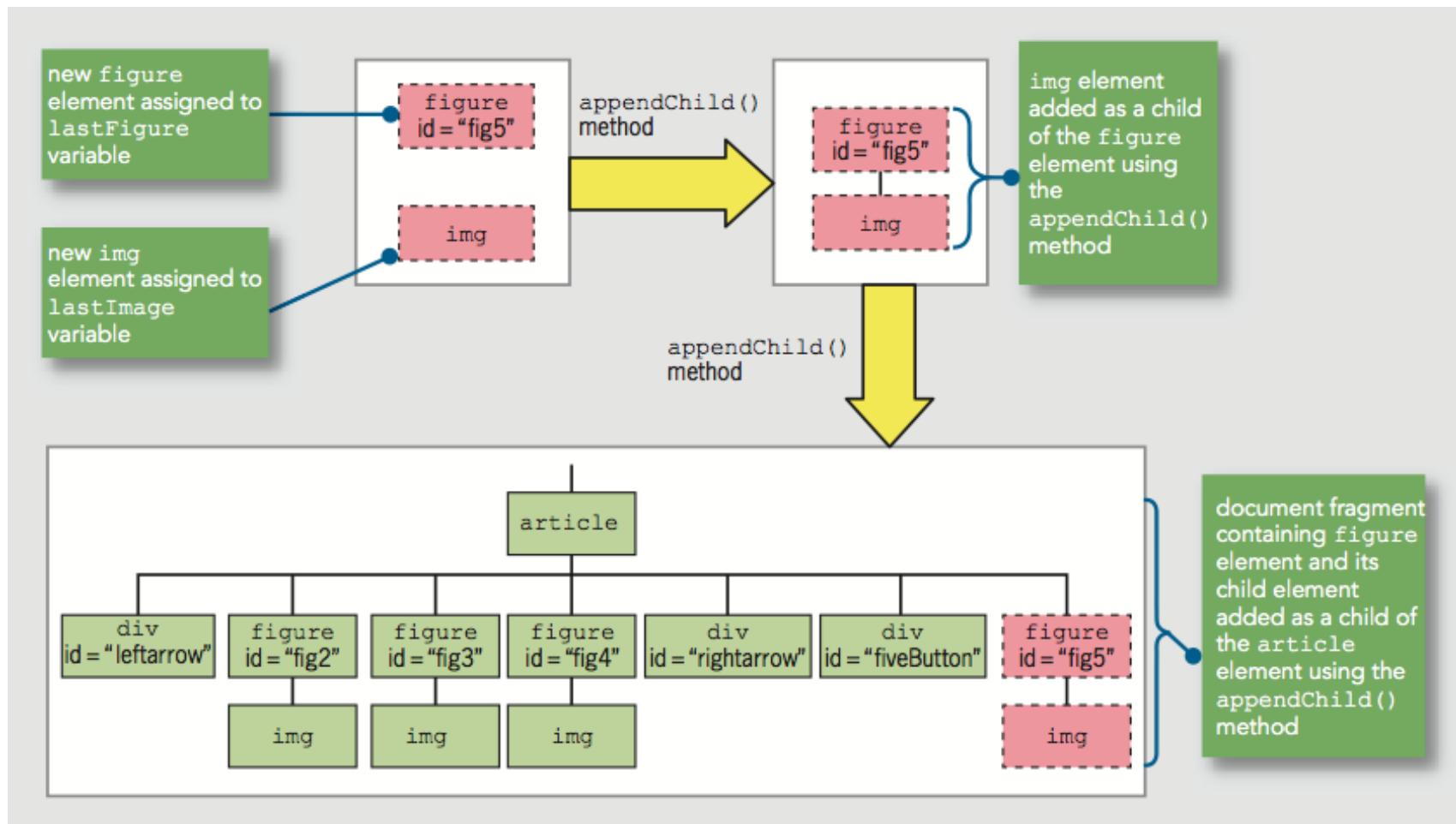
```
var list = document.getElementById("navList");
var contact = document.createElement("li");

list.appendChild(contact);
```

## Document fragment

- Set of connected nodes not part of document
- Can use appendChild() to add document fragment to DOM tree for a document

# Attaching Nodes (cont'd.)



**Table 5-7** Using the `appendChild()` method to attach nodes

# Cloning Nodes

---

Create new node same as existing node

`cloneNode ()` method

Syntax:

- `existingNode.cloneNode(true | false)`
- `true` argument clones child nodes
- `false` argument clones only specified parent node

Example:

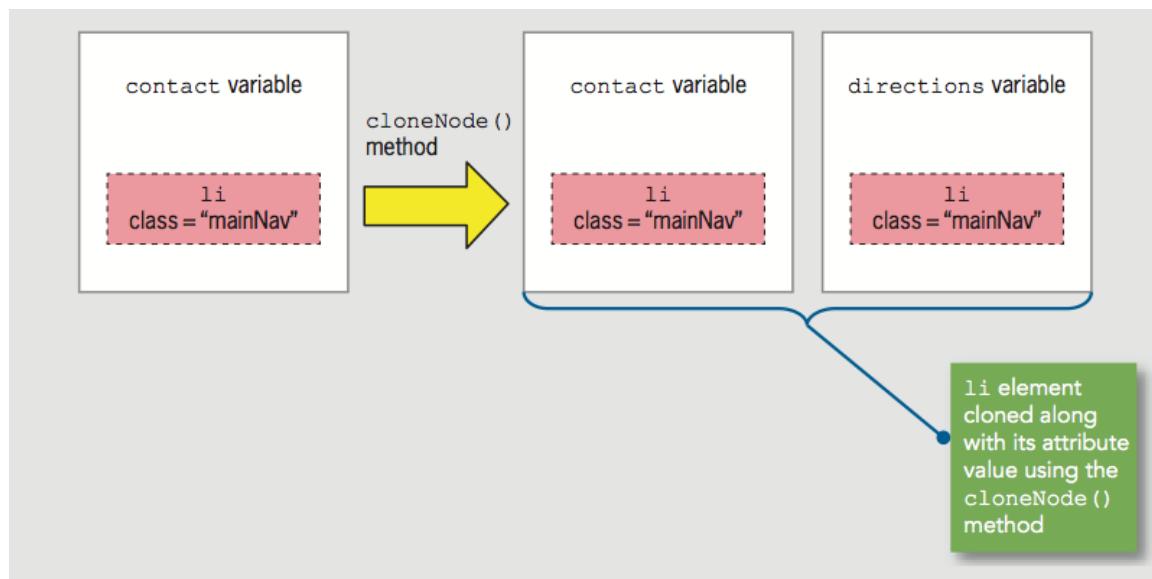
```
document.createElement("div");
```

# Cloning Nodes (cont'd.)

Example:

```
var contact = document.createElement("li");
contact.className = "mainNav";

var directions = contact.cloneNode(true);
```



**Figure 5-10** Using the `cloneNode()` method

# Inserting Nodes at Specific Positions in the Document Tree

---

New node created with `createElement()` is not attached to document tree

`appendChild()` adds node after existing child nodes

To specify a different position, use `insertBefore()`

Syntax:

- `parentNode.insertBefore(newChildNode, existingChildNode)`

# Inserting Nodes at Specific Positions in the Document Tree (cont'd.)

---

## Example:

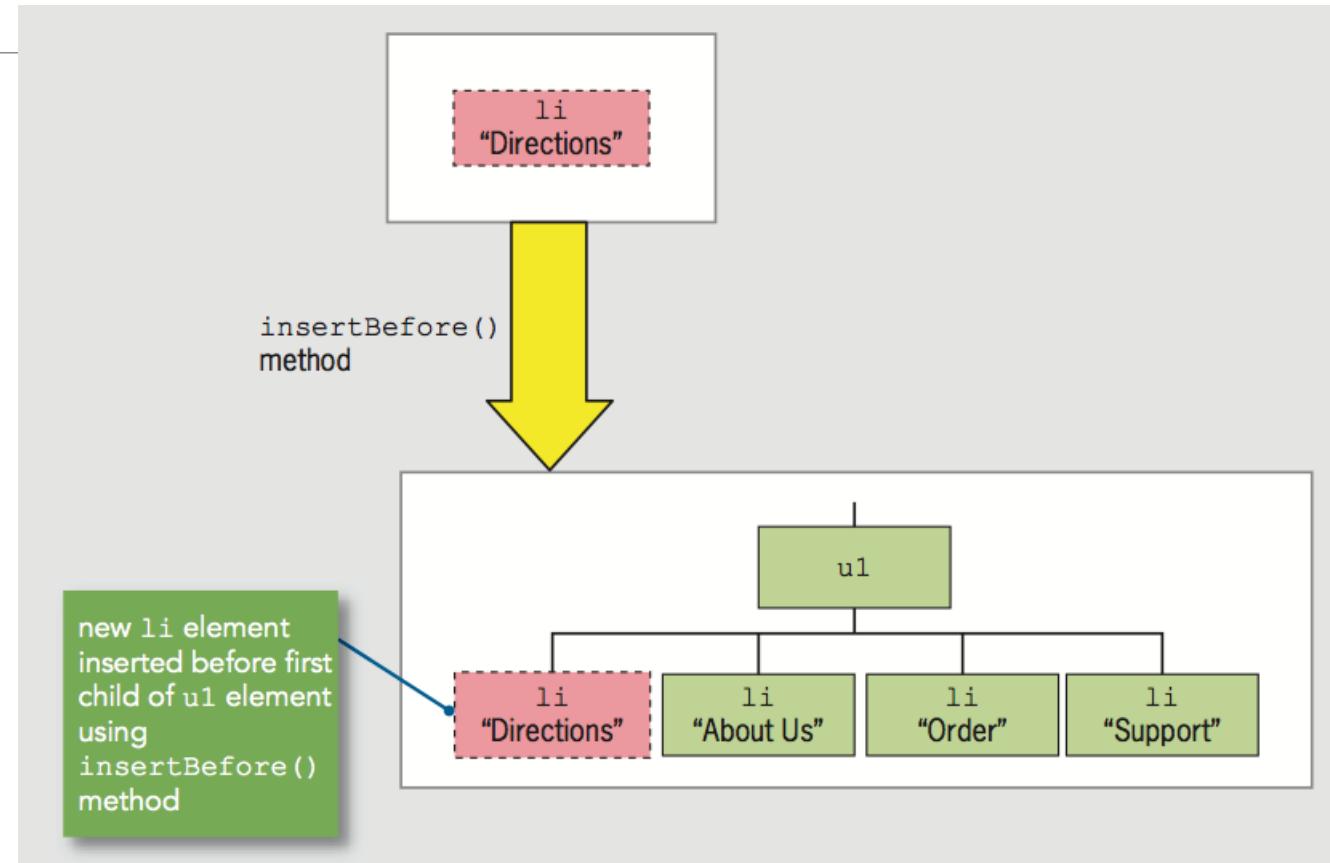
- HTML:

```
<ul id="topnav">
    <li><a href="aboutus.htm">About Us</a></li>
    <li><a href="order.htm">Order</a></li>
    <li><a href="support.htm">Support</a></li>
</ul>
```

- JavaScript:

```
var list = document.getElementById("topnav");
var directions = document.createElement("li");
directions.innerHTML = "Directions";
var aboutus = document.querySelectorAll("#topnav li")[0];
list.insertBefore(directions, aboutus);
```

# Inserting Nodes at Specific Positions in the Document Tree (cont'd.)



**Figure 5-14** Using the `insertBefore()` method

# Removing Nodes

---

`removeNode()` removes node from DOM tree

Syntax:

- `parentNode.removeChild(childNode)`

Can assign removed node to variable:

```
var list = document.getElementById("topnav");
var aboutus = document.querySelectorAll("#topnav li")[0];
var aboutNode = list.removeChild(aboutus);
```

- Node removed without being assigned to a variable is deleted during garbage collection

# Manipulating the Browser with the Window Object

---

## Window object

- Includes properties containing information about the web browser window or tab
- Contains methods to manipulate the web browser window or tab itself

# Manipulating the Browser with the Window Object

PROPERTY	DESCRIPTION
closed	Boolean value that indicates whether a window or tab has been closed
document	Reference to the Document object
history	Reference to the History object
innerHeight	Height of the window area that displays content, including the scrollbar if present
innerWidth	Width of the window area that displays content, including the scrollbar if present
location	Reference to the Location object
name	Name of the window or tab
navigator	Reference to the Navigator object
opener	Reference to the window that opened the current window or tab
outerHeight	Height of the entire browser window
outerWidth	Width of the entire browser window
screen	Reference to the Screen object
self	Self-reference to the Window object; identical to the window property
status	Temporary text that is written to the status bar
window	Self-reference to the Window object; identical to the self property

**Table 5-3** Window object properties

# Manipulating the Browser with the Window Object (cont'd.)

METHOD	DESCRIPTION
<code>alert()</code>	Displays a simple message dialog box with an OK button
<code>blur()</code>	Removes focus from a window or tab
<code>clearInterval()</code>	Cancels an interval that was set with <code>setInterval()</code>
<code>clearTimeout()</code>	Cancels a timeout that was set with <code>setTimeout()</code>
<code>close()</code>	Closes a web browser window or tab
<code>confirm()</code>	Displays a confirmation dialog box with OK and Cancel buttons
<code>focus()</code>	Makes a <code>Window</code> object the active window or tab
<code>moveBy()</code>	Moves the window relative to the current position
<code>moveTo()</code>	Moves the window to an absolute position
<code>open()</code>	Opens a new web browser window or tab
<code>print()</code>	Prints the document displayed in the current window or tab
<code>prompt()</code>	Displays a dialog box prompting a user to enter information

**Table 5-4** `Window` object methods (*continues*)

# Manipulating the Browser with the Window Object (cont'd.)

METHOD	DESCRIPTION
<code>resizeBy()</code>	Resizes a window by a specified amount
<code>resizeTo()</code>	Resizes a window to a specified size
<code>scrollBy()</code>	Scrolls the window or tab by a specified amount
<code>scrollTo()</code>	Scrolls the window or tab to a specified position
<code>setInterval()</code>	Repeatedly executes a function after a specified number of milliseconds have elapsed
<code>setTimeout()</code>	Executes a function once after a specified number of milliseconds have elapsed

**Table 5-4** Window object methods

# Manipulating the Browser with the Window Object (cont'd.)

---

## self property

- Refers to the current Window object
- Identical to using the window property to refer to the Window object
- Examples:
  - `window.close();`
  - `self.close();`

Web browser assumes reference to global object

## Good practice

- Use window or self references
  - When referring to a Window object property or method

# Opening and Closing Windows

---

Reasons to open a new Web browser window

- To launch a new Web page in a separate window
- To use an additional window to display information

When new Web browser window opened:

- New Window object created
  - Represents the new window

Know how to open a link in a new window using the `a` element's target attribute

```
<a href="http://www.wikipedia.org/">  
    target="wikiWindow">Wikipedia home page</a>
```

# Opening a Window

---

open () method of the Window object

- Opens new windows

## Syntax

```
window.open(url, name, options, replace);
```

METHOD	DESCRIPTION
resizeBy()	Resizes a window by a specified amount
resizeTo()	Resizes a window to a specified size
scrollBy()	Scrolls the window or tab by a specified amount
scrollTo()	Scrolls the window or tab to a specified position
setInterval()	Repeatedly executes a function after a specified number of milliseconds have elapsed
setTimeout()	Executes a function once after a specified number of milliseconds have elapsed

**Table 5-5** Arguments of the Window object's open () method

# Opening a Window (cont'd.)

---

Include all (or none) `window.open()` method arguments

Example:

- `window.open("http://www.wikipedia.org");`

# Opening a Window (cont'd.)

---

Customize new browser window or tab appearance

- Use `window.open()` method options argument

NAME	DESCRIPTION
<code>height</code>	Sets the window's height
<code>left</code>	Sets the horizontal coordinate of the left of the window, in pixels
<code>location</code>	Includes the URL Location text box
<code>menubar</code>	Includes the menu bar
<code>personalbar</code>	Includes the bookmarks bar (or other user-customizable bar)
<code>resizable</code>	Determines if the new window can be resized
<code>scrollbars</code>	Includes scroll bars
<code>status</code>	Includes the status bar
<code>toolbar</code>	Includes the Standard toolbar
<code>top</code>	Sets the vertical coordinate of the top of the window, in pixels
<code>width</code>	Sets the window's width

**Table 5-6** Common options of the `Window` object's `open()` method

# Opening a Window (cont'd.)

---

`window.open()` method name argument

- Same as value assigned to the `target` attribute
  - Specifies window name where the URL should open
- If name argument already in use
  - JavaScript changes focus to the existing Web browser window instead of creating a new window

# Opening a Window (cont'd.)

---

Window object's name property used to specify a target window with a link

- Cannot be used in JavaScript code

Assign the new Window object created with the window.open() method to a variable to control it

focus() method

- Makes a window the active window

# Closing a Window

---

`close()` method

- Closes a web browser window

`window.close()` or `self.close()`

- Closes the current window

# Working with Timeouts and Intervals

---

Window object's timeout and interval methods

- Creates code that executes automatically

`setTimeout ()` method

- Executes code after a specific amount of time
- Executes only once
- Syntax
  - `var variable = setTimeout ("code", milliseconds);`

`clearTimeout ()` method

- Cancel `setTimeout ()` before its code executes

Example on next slide

# Working with Timeouts and Intervals (cont'd.)

```
var buttonNotPressed = setTimeout("window.alert('Your ↵  
changes have been saved')", 10000);  
  
function buttonPressed() {  
  
    clearTimeout(buttonNotPressed);  
  
    window.open(index.htm);  
  
}
```

# Working with Timeouts and Intervals (cont'd.)

---

## setInterval () method

- Repeatedly executes the same code after being called only once
- Syntax:
  - `var variable = setInterval("code", milliseconds);`

## clearInterval () method

- Used to clear setInterval () method call

# The History Object

---

## History object

- Maintains internal list (history list)
  - All documents opened during current web browser session

## Security features

- Will not display URLs contained in the history list

# The History Object (cont'd.)

METHOD	DESCRIPTION
<code>back()</code>	Produces the same result as clicking a browser's Back button
<code>forward()</code>	Produces the same result as clicking a browser's Forward button
<code>go()</code>	Opens a specific document in the history list

**Table 5-7** Methods of the History object

# The History Object (cont'd.)

---

## go() method

- Allows navigation to a specific previously visited web page

## History object length property

- Provides specific number of documents opened during the current browser session
- Example:
  - Return to first document opened in current browser session:

```
history.go(-(history.length - 1));
```

# The Location Object

---

## Location object

- Allows changes to a new web page from within JavaScript code

Location object properties allow modification of URL individual portions

- Web browser automatically attempts to open that new URL

# The Location Object (cont'd.)

PROPERTIES	DESCRIPTION
hash	URL's anchor
host	Host and domain name (or IP address) of a network host
hostname	Combination of the URL's host name and port sections
href	Full URL address
pathname	URL's path
port	URL's port
protocol	URL's protocol
search	URL's search or query portion

**Table 5-8** Properties of the Location object

METHOD	DESCRIPTION
assign()	Loads a new web page
reload()	Causes the page that currently appears in the web browser to open again
replace()	Replaces the currently loaded URL with a different one

**Table 5-9** Methods of the Location object

# The Location Object (cont'd.)

---

## Location object's assign () method

- Same action as changing the href property
- Loads a new web page

## Location object's reload () method

- Equivalent to the browser Reload or Refresh button
- Causes current page to open again

## Location object's replace () method

- Replaces currently loaded URL with a different one

# The Navigator Object

---

## Navigator object

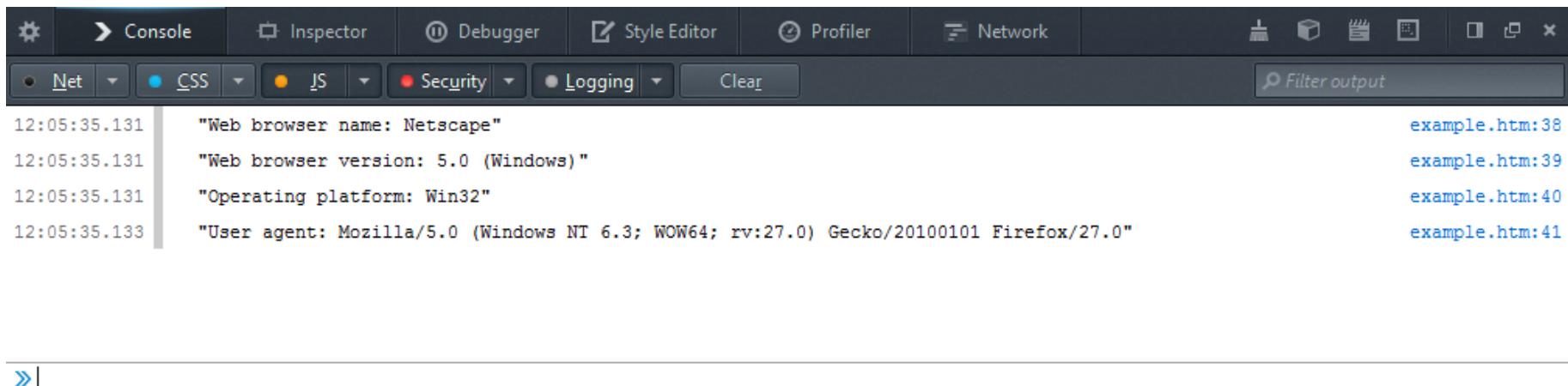
- Obtains information about current web browser
- Example: determine type of web browser running

PROPERTIES	DESCRIPTION
appName	Name of the web browser displaying the page
appVersion	Version of the web browser displaying the page
geolocation	API for accessing the user's current location and user permission settings denying or allowing access to that information
onLine	Whether the browser currently has a network connection
platform	Operating system in use on the client computer
userAgent	String stored in the HTTP user-agent request header, which contains information about the browser, the platform name, and compatibility

**Table 5-10** Properties of the Navigator object

# The Navigator Object (cont'd.)

```
console.log("Web browser name: " + navigator.appName);  
console.log("Web browser version: " + navigator.appVersion);  
console.log("Operating platform: " + navigator.platform);  
console.log("User agent: " + navigator.userAgent);
```



The screenshot shows the Firefox Developer Tools interface with the 'Console' tab selected. The top navigation bar includes tabs for 'Console', 'Inspector', 'Debugger', 'Style Editor', 'Profiler', and 'Network'. Below the tabs are dropdown menus for 'Net', 'CSS', 'JS', 'Security', 'Logging', and a 'Clear' button. A search bar labeled 'Filter output' is also present. The main console area displays the following log entries:

Time	Message	File
12:05:35.131	"Web browser name: Netscape"	example.htm:38
12:05:35.131	"Web browser version: 5.0 (Windows)"	example.htm:39
12:05:35.131	"Operating platform: Win32"	example.htm:40
12:05:35.133	"User agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:27.0) Gecko/20100101 Firefox/27.0"	example.htm:41

**Figure 5-19** Navigator object properties in Firefox console

# The Screen Object

---

## Screen object

- Obtains information about display screen's size, resolution, color depth

## Common use of Screen object properties

- Centering a web browser window in the middle of the display area

# The Screen Object (cont'd.)

PROPERTIES	DESCRIPTION
availHeight	Height of the display screen, not including operating system features such as the Windows taskbar
availWidth	Width of the display screen, not including operating system features such as the Windows taskbar
colorDepth	Display screen's bit depth if a color palette is in use; if a color palette is not in use, returns the value of the <code>pixelDepth</code> property
height	Height of the display screen
pixelDepth	Display screen's color resolution in bits per pixel
width	Width of the display screen

**Table 5-11** Properties of the `Screen` object

# The Screen Object (cont'd.)

---

Common Screen object properties uses

- Center a web browser window
- Example:

```
var winWidth = 300;  
  
var winHeight = 200;  
  
var leftPosition = (screen.width - winWidth) / 2;  
  
var topPosition = (screen.height - winHeight) / 2;  
  
var optionString = "width=" + winWidth + ",height=" +  
    + winHeight + ",left=" + leftPosition + ",top=" +  
    + topPosition;  
  
var openWin = window.open("", "CtrlWindow", optionString);
```

# Summary

---

Browser object model (BOM) or client-side object model

- Hierarchy of objects

Top-level object in the browser object model

- Window object

Document object: most important object

DOM represents web page displayed in window

# Summary (cont'd.)

---

Access elements with `getElementById()`,  
`getElementsByName()`, `getElementsByClassName()`,  
`getElementsByTagName()`, `querySelector()`, or  
`querySelectorAll()`

Access element content with `textContent()` or `innerHTML()` property

Access CSS properties using an element's `style` property

# Summary (cont'd.)

---

Create new node with `createElement()`

Attach node with `appendChild()`

Clone node with `cloneNode()`

Attach node in specific place with `insertBefore()`

Remove node with `removeNode()`

Open new window with `window.open()`

Close window with `window.close()`

# Summary (cont'd.)

---

`setTimeout ()` executes code after specific amount of time

`clearTimeout ()` cancels `setTimeout ()`

`setInterval ()` executes code repeatedly

`clearInterval ()` cancels `setInterval ()`

# Summary (cont'd.)

---

History object maintains an opened documents history list

Location object allows changes to a new web page from within JavaScript code

Navigator object obtains information about the current web browser

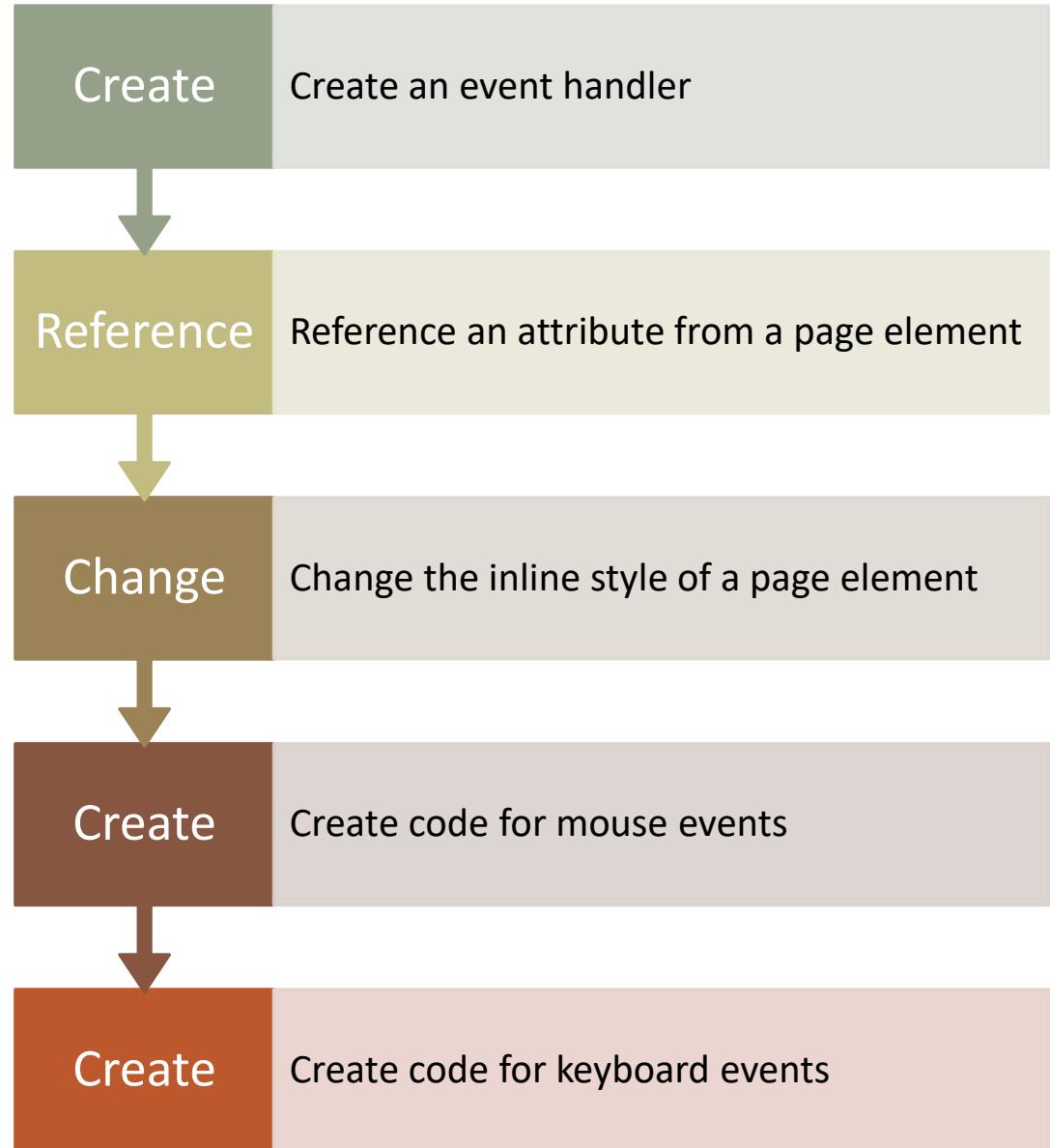
Screen object obtains information about the display screen's size, resolution, color depth

WEB  
DEVELOPMENT

# Programming for Web Form

---

# Objectives



# Objectives



Design and apply custom cursors



Create and apply anonymous functions



Work with alert, confirm, and prompt dialog boxes

# Introducing JavaScript Events



JavaScript programs run in response to events



**Events:** Actions initiated by the user or by the browser



Example:

Clicking an object on a form

Closing a web page



JavaScript events can be used to build a Hanjie puzzle, i.e., a grid in which each grid cell is either filled or left empty

Figure 11-4     HTML code for the puzzle1 table generated by the drawPuzzle() function

column and row totals are calculated by the drawPuzzle() function

filled cells are placed in the filled class

empty cells are placed in the empty class

```
<table id="hanjieGrid">
  <caption>Triangle (Easy)</caption>
  <tr>
    <th></th><th class="cols">5</th><th class="cols">4</th>
    <th class="cols">3</th><th class="cols">2</th><th class="cols">1</th>
  </tr>
  <tr>
    <th class="rows">5</th><td class="filled"></td><td class="filled"></td>
    <td class="filled"></td><td class="filled"></td><td class="filled"></td>
  </tr>
  <tr>
    <th class="rows">4</th><td class="filled"></td><td class="filled"></td>
    <td class="filled"></td><td class="filled"></td><td class="empty"></td>
  </tr>
  <tr>
    <th class="rows">3</th><td class="filled"></td><td class="filled"></td>
    <td class="filled"></td><td class="empty"></td><td class="empty"></td>
  </tr>
  <tr>
    <th class="rows">2</th><td class="filled"></td><td class="filled"></td>
    <td class="empty"></td><td class="empty"></td><td class="empty"></td>
  </tr>
  <tr>
    <th class="rows">1</th><td class="filled"></td><td class="empty"></td>
    <td class="empty"></td><td class="empty"></td><td class="empty"></td>
  </tr>
</table>
```

## Introducing JavaScript Events

THE HANJIE PUZZLE APP CONTAINS THREE PUZZLES NAMED PUZZLE1, PUZZLE2, AND PUZZLE3

## Creating an Event Handler

**Event handler:** A property that controls how an object will respond to an event

Event handler waits until the event occurs and then responds by running a function or command block to execute an action



# Creating an Event Handler

---

Event handlers can be added to a page element using the following attribute:

```
<element onevent =  
"script">
```

where *element* is the element in which the event occurs, *event* is the name of the event, and *script* are the commands that the browser runs in response to the event

# Creating an Event Handler

Event handlers can also be defined as object properties using the command

```
object.onevent = function;
```

where *object* is the object in which the event occurs, *event* is the name of the event, and *function* is the name of a function run in response to the event

**Figure 11-6**

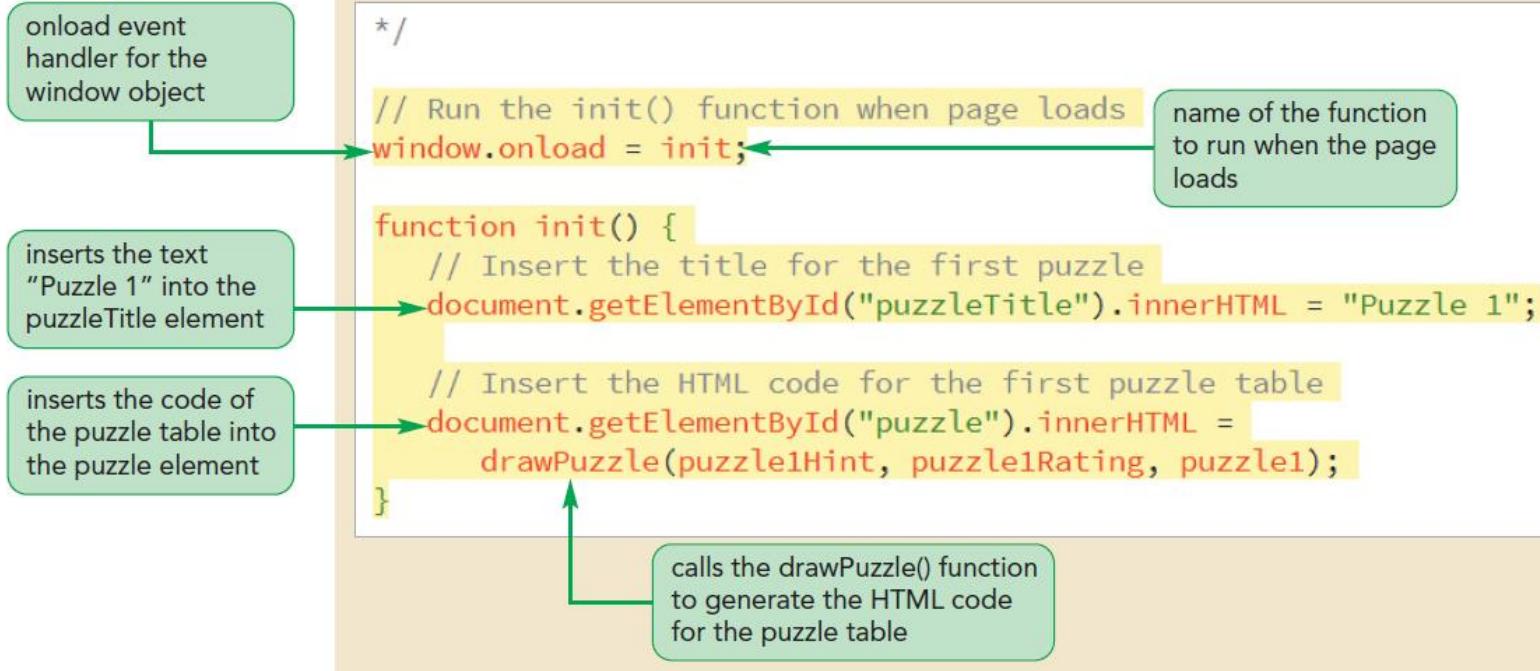
**Event handlers for the browser window**

<b>Event Handler</b>	<b>Run Script</b>
onbeforeunload	when page is about to be unloaded by the browser
oncopy	when the user copies the content of an element
oncut	when the user cuts the content of an element
onerror	when an error occurs while loading an external file, such as an image or a video clip
onload	after the page has finished loading
onpaste	when the user pastes some content into an element
onresize	when the browser window is resized
onunload	when the page is unloaded by the browser (or the browser window is closed)

## Creating an Event Handler

Figure 11-7

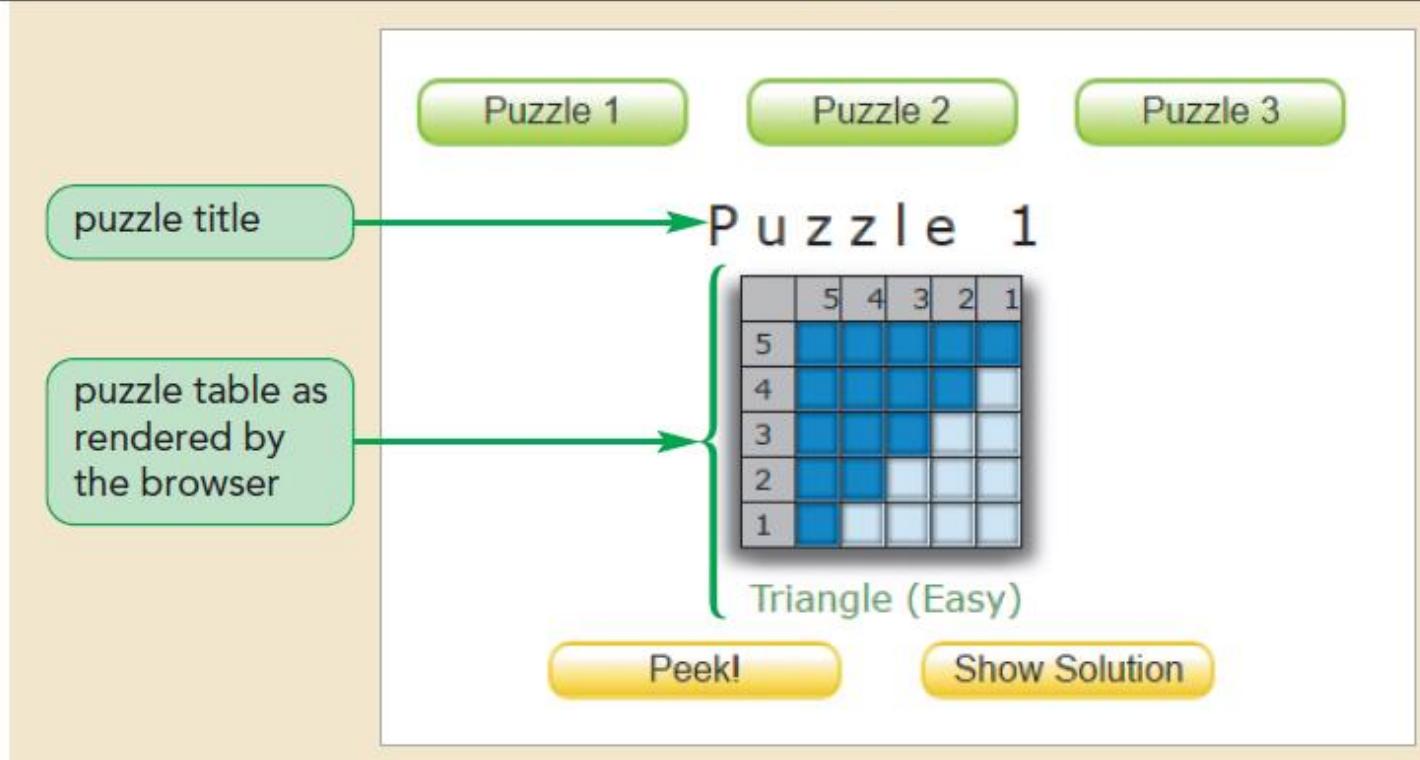
Applying the onload event handler



## Creating an Event Handler

# Creating an Event Handler

**Figure 11-8** The Puzzle 1 table loaded into the web page



# Creating an Event Handler

---

Add an event handler to the buttons in the Figure 11-8

Apply the following `onclick` event handler to respond to a mouse click:

*object.onclick = function;*

where *object* is the page element that is being clicked and *function* is the function run in response to the `click` event

Figure 11-9

### Assigning the onclick event handler

```
function init() {  
    // Insert the title for the first puzzle  
    document.getElementById("puzzleTitle").innerHTML = "Puzzle 1";
```

```
// Insert the HTML code for the first puzzle table  
document.getElementById("puzzle").innerHTML =  
    drawPuzzle(puzzle1Hint, puzzle1Rating, puzzle1);
```

```
//Add event handlers for the puzzle buttons  
var puzzleButtons = document.getElementsByClassName("puzzles");
```

```
for (var i = 0; i < puzzleButtons.length; i++) {  
    puzzleButtons[i].onclick = swapPuzzle;  
}
```

creates an object collection of all elements in the puzzles class

loops through every object in the puzzleButtons collection

attaches an onclick event handler to each puzzle button

runs the swapPuzzle() function when the button is clicked

# Creating an Event Handler

# Creating an Event Handler

The challenge lies in determining which button was clicked

There is no way of knowing which puzzle to load into the page without knowing which button activated the onclick event handler

This information can be determined using the event object



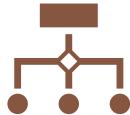
EVENT OBJECT IS AN OBJECT  
THAT CONTAINS  
PROPERTIES AND METHODS  
ASSOCIATED WITH AN  
EVENT



EXAMPLE: THE ACTION OF  
CLICKING A MOUSE BUTTON  
GENERATES AN EVENT  
OBJECT CONTAINING  
INFORMATION SUCH AS  
WHICH MOUSE BUTTON  
WAS CLICKED, WHERE IN  
THE PAGE IT WAS CLICKED,  
THE TIME AT WHICH IT WAS  
CLICKED, AND SO FORTH

# Using the Event Object

# Using the Event Object



**Event object is passed as an argument to the function handling the event to retrieve the information contained in the event object**



**Example:**

```
function myFunction(evt) {  
    function code  
}
```



**where *evt* is the name assigned to the parameter receiving the event object from the event handler**

# Using the Event Object

Figure 11-10 Event object properties and methods

Property	Description
<code>evt.bubbles</code>	Returns a Boolean value indicating whether the event is bubbling up through the object hierarchy, where <code>evt</code> is the event object for the event
<code>evt.cancelable</code>	Returns a Boolean value indicating whether the event can have its default action canceled
<code>evt.currentTarget</code>	Returns the object that is currently experiencing the event
<code>evt.defaultPrevented</code>	Returns a Boolean value indicating whether the <code>preventDefault()</code> method was called for the event
<code>evt.eventPhase</code>	Returns the phase of the event propagation the event object is currently at, where 0 = NONE, 1 = CAPTURING_PHASE, 2 = AT_TARGET, and 3 = BUBBLING_PHASE
<code>evt.isTrusted</code>	Returns a Boolean value indicating whether the event is trusted by the browser
<code>evt.target</code>	Returns the object in which the event was initiated
<code>evt.timeStamp</code>	Returns the time (in milliseconds) when the event occurred
<code>evt.type</code>	Returns the type of the event
<code>evt.view</code>	Reference the browser window in which the event occurred
Method	Description
<code>evt.preventDefault()</code>	Cancels the default action associated with the event
<code>evt.stopImmediatePropagation()</code>	Prevents other event listeners of the event from being called
<code>evt.stopPropagation()</code>	Prevents further propagation of the event in the capturing and bubbling phase

# Using the Event Object

Figure 11-11

Creating the swapPuzzle() function

```
//Add event handlers for the puzzle buttons
var puzzleButtons = document.getElementsByClassName("puzzles");
for (var i = 0; i < puzzleButtons.length; i++) {
    puzzleButtons[i].onclick = swapPuzzle;
}

function swapPuzzle(e) {
    var puzzleID = e.target.id;
    var puzzleTitle = e.target.value;
    document.getElementById("puzzleTitle").innerHTML = puzzleTitle;

    switch (puzzleID) {
        case "puzzle1":
            document.getElementById("puzzle").innerHTML =
                drawPuzzle(puzzle1Hint, puzzle1Rating, puzzle1);
            break;
        case "puzzle2":
            document.getElementById("puzzle").innerHTML =
                drawPuzzle(puzzle2Hint, puzzle2Rating, puzzle2);
            break;
        case "puzzle3":
            document.getElementById("puzzle").innerHTML =
                drawPuzzle(puzzle3Hint, puzzle3Rating, puzzle3);
            break;
    }
}
```

event object for the event handler

retrieves the value of the clicked button

displays the puzzle based on the value of the puzzleID variable

the target property references the button that was clicked

retrieves the ID of the clicked button

# Exploring Object Properties



JavaScript object properties that mirror HTML attributes follow certain conventions



All JavaScript properties must begin with a lowercase letter



If the HTML attribute consists of multiple words, JavaScript property follows a format known as **camel case**, i.e., the initial word is in lowercase and the first letter of each subsequent word is in uppercase



IF THE NAME OF THE  
HTML ATTRIBUTE IS  
A RESERVED  
JAVASCRIPT NAME  
OR KEYWORD, THE  
CORRESPONDING  
JAVASCRIPT  
PROPERTY IS OFTEN  
PREFACED WITH THE  
TEXT STRING HTML



CLASS ATTRIBUTE IS  
AN EXCEPTION TO  
THIS CONVENTION  
BECAUSE THE CLASS  
NAME IS RESERVED  
BY JAVASCRIPT FOR  
OTHER PURPOSES



REFERENCES TO THE  
HTML CLASS  
ATTRIBUTE USE THE  
CLASSNAME  
PROPERTY

# Exploring Object Properties

# Object Properties and Inline Styles

Inline styles for each page element can be applied using the following style attribute:

```
<element style =  
"property:value"> ...
```

where

- *element* is the page element
- *property* is a CSS style property
- *value* is the value assigned to that property

# Object Properties and Inline Styles (continued)

The equivalent command in JavaScript is

- *object.style.property = "value";*

with the *property* style written in camel case

Inline styles have precedence over style sheets

# Creating Object Collections with CSS Selectors

Change the background color of all table cells by applying the following CSS style rule for all td elements:

```
table#hanjieGrid td {  
background-color: rgb(233,  
207, 29);  
}
```

# Creating Object Collections with CSS Selectors

In JavaScript, to change the background color of all table cells, you must first define an object collection based on a CSS selector using the following `querySelectorAll()` method:

```
document.querySelectorAll(  
selector)
```

where `selector` is the CSS selector that the object collection is based on

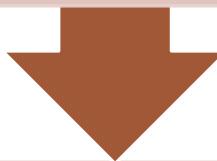
# Creating Object Collections with CSS Selectors

Once the object collection has been defined, change the background-color style of each td element by applying the backgroundColor property to the objects in the object collection



Reference only the first element that matches a selector pattern using the following JavaScript method:

**`document.querySelector(selector)`**



where *selector* is a CSS selector

Figure 11-13

### Creating the setupPuzzle() function

loops through every td element in the puzzleCells collection

changes the value of the background-color inline style for each td element to gold

```
}
```

```
function setupPuzzle() {  
    /* Match all of the data cells in the puzzle */  
    puzzleCells = document.querySelectorAll("table#hanjieGrid td");  
  
    /* Set the initial color of each cell to gold */  
    for (var i = 0; i < puzzleCells.length; i++) {  
        puzzleCells[i].style.backgroundColor = "rgb(233, 207, 29)";  
    }  
}
```

creates an object collection of all of the td elements in the hanjieGrid table

## Creating Object Collections with CSS Selectors

Figure 11-14

Revising the init() function

defines puzzleCells as a global variable so it can be used in all functions

```
var puzzleCells;

function init() {
    // Insert the title for the first puzzle
    document.getElementById("puzzleTitle").innerHTML = "Puzzle 1";

    // Insert the HTML code for the first puzzle table
    document.getElementById("puzzle").innerHTML =
        drawPuzzle(puzzle1Hint, puzzle1Rating, puzzle1);

    //Add event handlers for the puzzle buttons
    var puzzleButtons = document.getElementsByClassName("puzzles");
    for (var i = 0; i < puzzleButtons.length; i++) {
        puzzleButtons[i].onclick = swapPuzzle;
    }
}

sets up the initial puzzle displayed in the web page
```

## Creating Object Collections with CSS Selectors

# Working with Mouse Events

Figure 11-17 Mouse and pointer events

Event	Description
click	The mouse button has been pressed and released
contextmenu	The right mouse button has been pressed and released
dblclick	The mouse button has been double-clicked
mousedown	The mouse button is pressed
mouseenter	The mouse pointer is moved onto the element
mouseleave	The pointer is moved off the element
mousemove	The pointer is moving over the element
mouseout	The pointer is moved off the element and any nested elements
mouseover	The pointer is moved onto the element and any nested elements
mouseup	The mouse button is released
select	Text is selected by the pointer
wheel	The mouse scroll wheel has been rotated

JavaScript supports events associated with the mouse such as clicking, right-clicking, double-clicking, and moving the pointer over and out of page elements

# Working with Mouse Events

A mouse action can be comprised of several events

The action of clicking the mouse button is comprised of three events, fired in the following order:

- mousedown (the button is pressed down)
- mouseup (the button is released)
- click (the button has been pressed and released)

# Working with Mouse Events (continued 2)

The event object for mouse events has a set of properties that can be used to give specific information about the state of the mouse

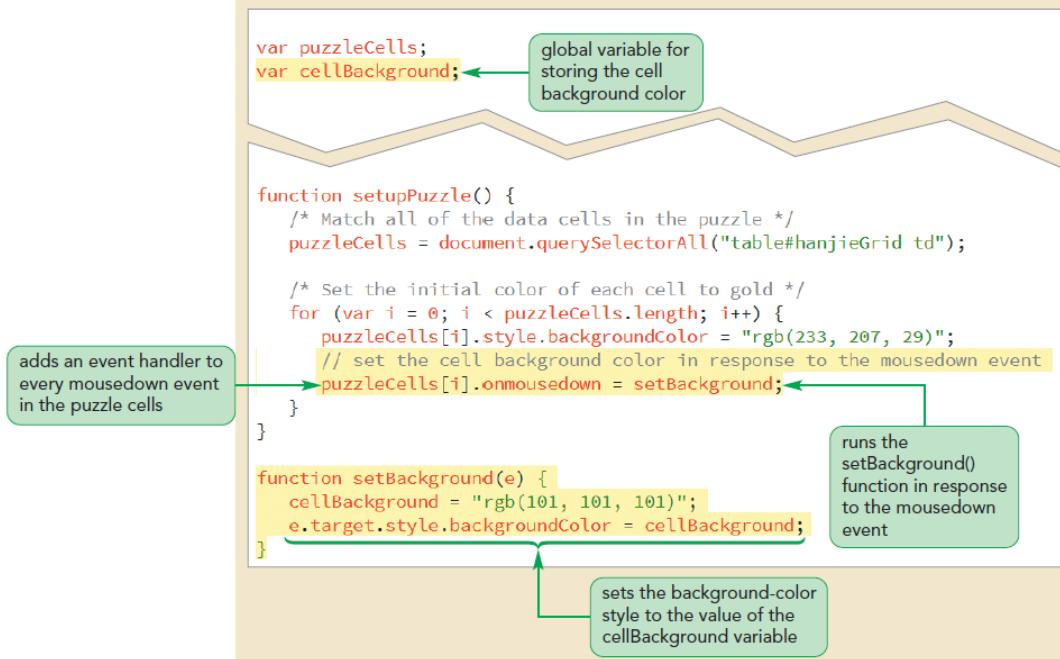
**Figure 11-18** Mouse event object properties

Event Property	Description
<code>evt.button</code>	Returns a number indicating the mouse button that was pressed, where 0 = left, 1 = wheel or middle, and 3 = right and <code>evt</code> is event object for the mouse event
<code>evt.buttons</code>	Returns a number indicating the mouse button or buttons that were pressed, where 1 = left, 2 = right, 4 = wheel or middle, 8 = back, 16 = forward, and other multiple buttons are indicated by the sum of their numbers
<code>evt.clientX</code>	Returns the horizontal coordinate (in pixels) of the mouse pointer relative to the browser window
<code>evt.clientY</code>	Returns the vertical coordinate (in pixels) of the mouse pointer relative to the browser window
<code>evt.detail</code>	Returns the number of times the mouse button was clicked
<code>evt.pageX</code>	Returns the horizontal coordinate (in pixels) of the mouse pointer relative to the document
<code>evt.pageY</code>	Returns the vertical coordinate (in pixels) of the mouse pointer relative to the document
<code>evt.relatedTarget</code>	References the secondary target of the event; for the <code>mouseover</code> event this is the element that the pointer is leaving and for the <code>mouseout</code> event this is the element that the pointer is entering
<code>evt.screenX</code>	Returns the horizontal coordinate (in pixels) of the mouse pointer relative to the physical screen
<code>evt.screenY</code>	Returns the vertical coordinate (in pixels) of the mouse pointer relative to the physical screen
<code>evt.which</code>	Returns a number indicating the mouse button that was pressed, where 0 = none, 1 = left, 2 = wheel or middle, and 3 = right

# Working with Mouse Events

Figure 11-19

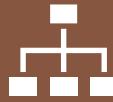
Applying the onmousedown event handler



# Introducing the Event Model



**Event model:** Describes how events and objects interact within the web page and web browser



The process in which a single event is applied to a hierarchy of objects is part of the event model

# Introducing the Event Model

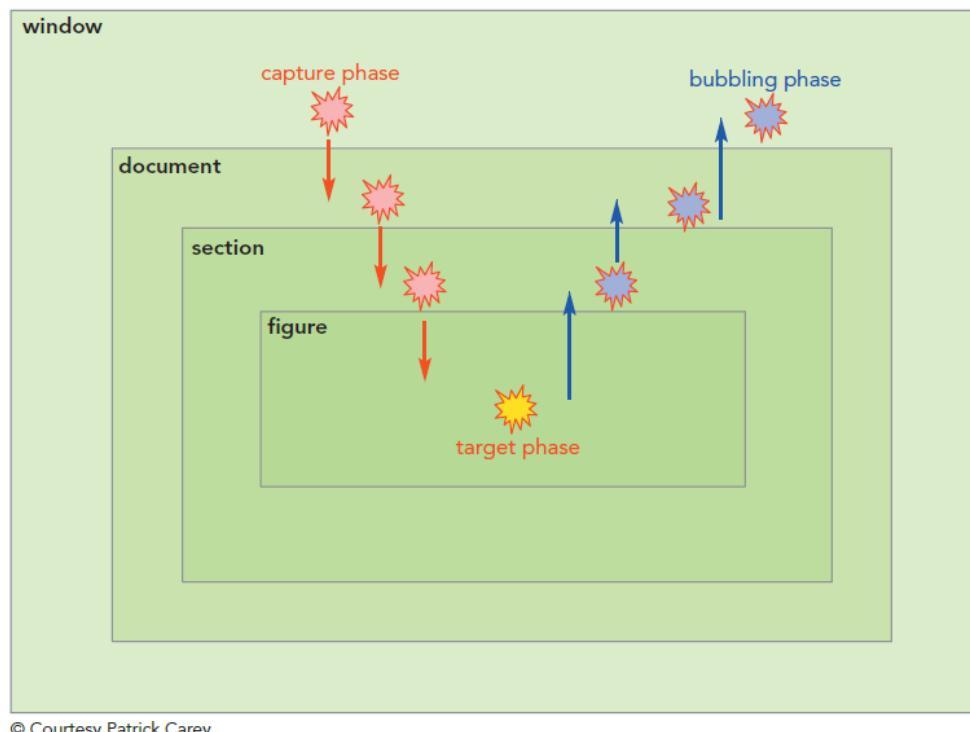
Once an event has been initiated, it propagates through the object hierarchy in three phases

- **Capture phase:** The event moves down the object hierarchy starting from the root element (the browser window) and moving inward until it reaches the object that initiated the event
- **Target phase:** The event has reached the target of the event object and no longer moves down the object hierarchy

# Introducing the Event Model

- **Bubbling phase:** The event propagates up the object hierarchy back to the root element (browser window) where the propagation stops

Figure 11-21 Event propagation in the event model



© Courtesy Patrick Carey

# Introducing the Event Model

## Limitations of event handlers

- Event handlers, such as `onclick` and `onmousedown`, respond to events during the target phase, but they do not recognize the propagation of events through the capture and bubbling phases
- Only one function can be applied to an event handler at a time

# Adding an Event Listener

**Event listener:** Listens for events as they propagate through the capture, target, and bubble phases, allowing the script to respond to an event within any phase

Unlike event handlers, more than one function can be applied to an event using event listeners

# Adding an Event Listener

Add an event listener to an object by applying the `addEventListener()` method

```
object.addEventListener(  
  event, function [,  
  capture = false]);
```

where

- *object* is the object in which the event occurs
- *event* is the event
- *function* is the function that is run in response to the event

# Adding an Event Listener

- `capture` is an optional Boolean value
  - `true` indicates that the function is executed during the capture phase
  - `false` (the default) indicates that the function is run during the bubbling phase

Figure 11-23

## Adding an event listener for the mouseup event

```
//Add event handlers for the puzzle buttons
var puzzleButtons = document.getElementsByClassName("puzzles");
for (var i = 0; i < puzzleButtons.length; i++) {
    puzzleButtons[i].onclick = swapPuzzle;
}

setupPuzzle();

// Add an event listener for the mouseup event
document.addEventListener("mouseup", endBackground);
```

runs the `endBackground()` function in response to the mouseup event

listens for the mouseup event occurring anywhere within document

# Removing an Event Listener

---

The event model allows to remove event listeners from the document by applying `removeEventListener()` method

```
object.removeEventListener(event,  
function [, capture = false]);
```

where `object`, `event`, `function`, and `capture` have the same meanings as the `addEventListener()` method

# Removing an Event Listener (continued)

Figure 11-24 Removing an event listener

```
function extendBackground(e) {  
    e.target.style.backgroundColor = cellBackground;  
}  
  
function endBackground() {  
    // Remove the event listener for every puzzle cell  
    for (var i = 0; i < puzzleCells.length; i++) {  
        puzzleCells[i].removeEventListener("mouseenter", extendBackground);  
    }  
}
```

removes the event listener that  
runs the extendBackground()  
function in response to the  
mouseenter event

# Controlling Event Propagation

The browser has its own default responses to events

Apply the following `preventDefault()` method to the event object to prevent the occurrence of the browser's default actions:

- `evt.preventDefault()`

# Controlling Event Propagation



Alternatively, you can prevent the browser's default action by returning the value false from the event handler function



The return false; statement does not prevent default actions if event listeners are used in place of event handlers

Figure 11-26

Preventing the default browser action

```
function setBackground(e) {  
    cellBackground = "rgb(101, 101, 101)";  
    e.target.style.backgroundColor = cellBackground;  
  
    // Create an event listener for every puzzle cell  
    for (var i = 0; i < puzzleCells.length; i++) {  
        puzzleCells[i].addEventListener("mouseenter", extendBackground);  
    }  
  
    // Prevent the default action of selecting table text  
    e.preventDefault();  
}
```

prevents the default browser action of selecting text in the table

# Controlling Event Propagation

# Exploring Keyboard Events

JavaScript supports the keydown, keypress, and keyup events that allow users to interact with the web page and browser through the keyboard

**Figure 11-27** **Keyboard Events**

Event	Description
keydown	A key is pressed down
keypress	A key is pressed down and released, resulting in a character being typed
keyup	A key is released

# Exploring Keyboard Events

---

The `keydown` and `keypress` events are similar in name; the difference between them is as follows:

- The `keydown` and `keyup` events are fired in response to the physical act of pressing the key down and of a key moving up when it is no longer held down
- The `keypress` event is fired in response to the computer generating a character

# Exploring Keyboard Events

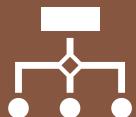
Figure 11-28

Keyboard Event Properties

Event Property	Description
<code>evt.altKey</code>	Returns a Boolean value indicating whether the Alt key was used in the event object, <code>evt</code>
<code>evt.ctrlKey</code>	Returns a Boolean value indicating whether the Ctrl key was used in the event
<code>evt.charCode</code>	Returns the Unicode character code of the key used in the <code>keypress</code> event
<code>evt.key</code>	Returns the text of the key used in the event (not supported by the Safari browser)
<code>evt.keyCode</code>	Returns the Unicode character code of the key used in the <code>keypress</code> event or the key code used in the <code>keydown</code> or <code>keyup</code> events
<code>evt.location</code>	Returns the location number of the key where 0 = key located in the standard position, 1 = a key on the keyboard's left edge, 2 = a key on the keyboard's right edge, and 3 = a key on the numeric keypad
<code>evt.metaKey</code>	Returns a Boolean value indicating whether the meta key (the Command key on Mac keyboards or the Windows key on PC keyboards) was used in the event
<code>evt.shiftKey</code>	Returns a Boolean value indicating whether the Shift key was used in the event
<code>evt.which</code>	Returns the Unicode character code of the key used in the <code>keypress</code> event or the key code used in the <code>keydown</code> or <code>keyup</code> events



The value associated with a key event property is affected by the event itself



For the keypress event, the charCode, keyCode, and which properties all return a Unicode character number



Sample values of the charCode, keyCode, which, and key properties for different keyboard keys and events are shown in Figure 11-29

# Exploring Keyboard Events

# Exploring Keyboard Events

Figure 11-29 Keyboard Event Properties

Key(s)	Values with keypress	Values with keydown and keyup
a, z	charCode = 97, 122 keyCode = 97, 122 which = 97, 122 key = "a", "z"	charCode = 0, 0 keyCode = 65, 90 which = 65, 90 key = "a", "z"
1, 9 (on numeric keypad)	charCode = 49, 57 keyCode = 49, 57 which = 49, 57 key = "1", "9"	charCode = 0, 0 keyCode = 97, 105 which = 97, 105 key = "1", "9"
1, 9 (on top keyboard row)	charCode = 49, 57 keyCode = 49, 57 which = 49, 57 key = "1", "9"	charCode = 0, 0 keyCode = 49, 57 which = 49, 57 key = "1", "9"
Shift, Ctrl, Alt, Meta (Mac Command key; PC Window key) Command	no event detected	charCode = 0, 0, 0, 0 keyCode = 16, 17, 18, 91 which = 16, 17, 18, 91 key = "Shift", "Control", "Alt", "Meta"
←, ↑, →, ↓	no event detected	charCode = 0, 0, 0, 0 keyCode = 37, 38, 39, 40 which = 37, 38, 39, 40 key = "ArrowLeft", "ArrowUp", "ArrowRight", "ArrowDown"

# Exploring Keyboard Events

**Modifier keys:** Alt, Ctrl, Shift, and Command keys

In addition to character keys, JavaScript supports the modifier keys through the use of the altKey, ctrlKey, shiftKey, and metaKey properties

# Exploring Keyboard Events

Figure 11-30

Changing the background color for different modifier keys

if the Shift key is pressed down, changes the background to gold

if the Alt key is pressed down, changes the background to white

otherwise, changes the background to gray

```
function setBackground(e) {  
    // Set the background based on the keyboard key  
    if (e.shiftKey) {  
        cellBackground = "rgb(233, 207, 29)";  
    } else if (e.altKey) {  
        cellBackground = "rgb(255, 255, 255)";  
    } else {  
        cellBackground = "rgb(101, 101, 101)";  
    }  
  
    e.target.style.backgroundColor = cellBackground;
```

sets the background-color style to the value of the cellBackground variable

## Changing the Cursor Style

Cursors can be defined using the following CSS cursor style:

- `cursor: cursorTypes;`

where *cursorTypes* is a comma-separated list of cursor types

# Changing the Cursor Style

JavaScript command to define cursors is as follows:

- *object.style.cursor = cursorTypes;*

where *object* is the page object that will display the cursor style when hovered over by the mouse pointer

# Changing the Cursor Style

Create a customized cursor from an image file using `url(image)` where `image` is an image file

Example: `cursor: url(jpf_pencil.png), pointer`

Figure 11-33 Setting cursor style for the puzzle cells

```
function setupPuzzle() {
    /* Match all of the data cells in the puzzle */
    puzzleCells = document.querySelectorAll("table#hanjieGrid td");

    /* Set the initial color of each cell to gold */
    for (var i = 0; i < puzzleCells.length; i++) {
        puzzleCells[i].style.backgroundColor = "rgb(233, 207, 29)";
        // Set the cell background color in response to the mousedown event
        puzzleCells[i].onmousedown = setBackground;
        // Use a pencil image as the cursor
        puzzleCells[i].style.cursor = "url(jpf_pencil.png), pointer";
    }
}
```

uses the pencil image as the cursor for puzzle cells

if the image file is not supported, uses the generic pointer cursor

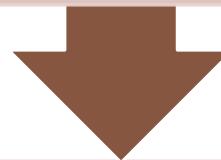
# Changing the Cursor Style

By default, the click point for a cursor is located in the top-left corner of the cursor image at the coordinates (0, 0)



Specify a different location by adding the (x, y) coordinates of the click point to the cursor definition as follows:

`url(image) x y`



where *x* is the x-coordinate and *y* is the y-coordinate of the click point in pixels

# Working with Functions as Objects

Everything in JavaScript is an object, including functions

Anything that can be done with an object can be done with a function, including

- Storing a function as variable
- Storing a function as an object property

# Working with Functions as Objects

Using one function as a parameter in another function

Nesting one function within another function

Returning a function as the result of another function

Modifying the properties of a function



# Function Declarations and Function Operators

---

The following `hello()` function is created using the **function declaration** format:

```
function hello() {  
    alert("Welcome to JS! ");}
```

**Function operator:** The definition of the function becomes the variable's "value"

```
var hello = function () {  
    alert("Welcome to JS! ");  
}
```

# Function Declarations and Function Operators

The two ways of defining the `hello()` function differ in how they are stored

- Functions defined with a function declaration are created and saved as the browser parses the code prior to the script being run
  - Since the function is already stored in memory, the statements that run the function can be placed prior to the statement that declares the function

# Function Declarations and Function Operators

- Function operators are evaluated as they appear in the script after the code has been parsed by the browser
  - Function operators are more flexible than function declarations, allowing a function to be placed anywhere a variable can be placed



Anonymous function has  
function declaration without  
the function name



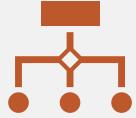
Example: The  
following structure  
is an anonymous  
function:

```
function() {  
  commands  
}
```

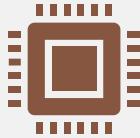


An anonymous function can be  
inserted into any expression  
where a function reference is  
required

# Anonymous Functions



Functions that are named are called  
**named functions**



Anonymous functions are more concise and easier to manage because the function is directly included with the expression that invokes it



Anonymous functions limit the scope of the function to exactly where it is needed

# Anonymous Functions

# Anonymous Functions

Figure 11-37 Using an anonymous function

```
setupPuzzle();  
  
// Add an event handler for the mouseup event  
document.addEventListener("mouseup", endBackground);  
  
// Add an event listener to the Show Solution button  
document.getElementById("solve").addEventListener("click",  
    function() {  
        // Remove the inline backgroundColor style from each cell  
        for (var i = 0; i < puzzleCells.length; i++) {  
            puzzleCells[i].style.backgroundColor = "";  
        }  
    }  
);
```

listens for the Show Solution button click event

inserts the response to the click event as an anonymous function

erases the inline background-color style from each puzzle cell by setting the style value to an empty text string

# Passing Variable Values into Anonymous Functions

JavaScript supports two types of variables:

- Global variables: Declared outside of any function and thus are accessible throughout the app
- Local variables: Declared within a function and are only accessible to code within that function

# Passing Variable Values into Anonymous Functions

Global variables should be avoided when possible because

- global variables are accessible to every function in the application
- the task of tracking which functions are using and modifying the global variables becomes increasingly difficult as the application grows in size and complexity

# Passing Variable Values into Anonymous Functions

An advantage of using anonymous functions is that they reduce the need for global variables because they perform their actions locally within a function



One of the challenges of anonymous functions is keeping track of all of the nested levels of functions and procedures

Figure 11-43

Using an anonymous function with the setTimeout() method

```
// Display incorrect gray cells in red
for (var i = 0; i < empty.length; i++) {
    if (empty[i].style.backgroundColor === "rgb(101, 101, 101)") {
        empty[i].style.backgroundColor = "rgb(255, 101, 101)";
    }
}

// Remove the hints after 0.5 seconds
setTimeout(
    function() {
    },
    500);
});
```

runs anonymous function after a 0.5 second delay

sets the delay time in milliseconds

## Passing Variable Values into Anonymous Functions

# Displaying Dialog Boxes

Alert dialog box can be created using the following `alert()` method:

```
alert(text)
```

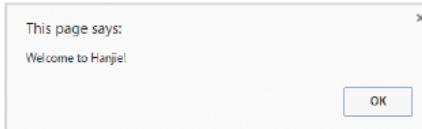
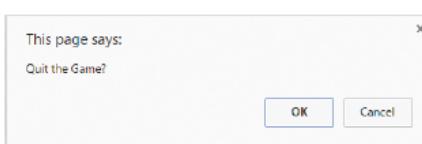
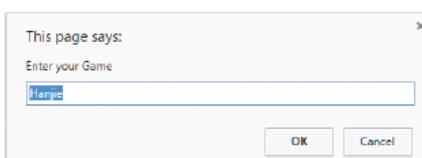
where *text* is the message displayed in the alert dialog box

When an alert dialog box is displayed, the execution of the program code halts until the user clicks the OK button in the dialog box

# Displaying Dialog Boxes

JavaScript supports confirmation and prompt dialog boxes

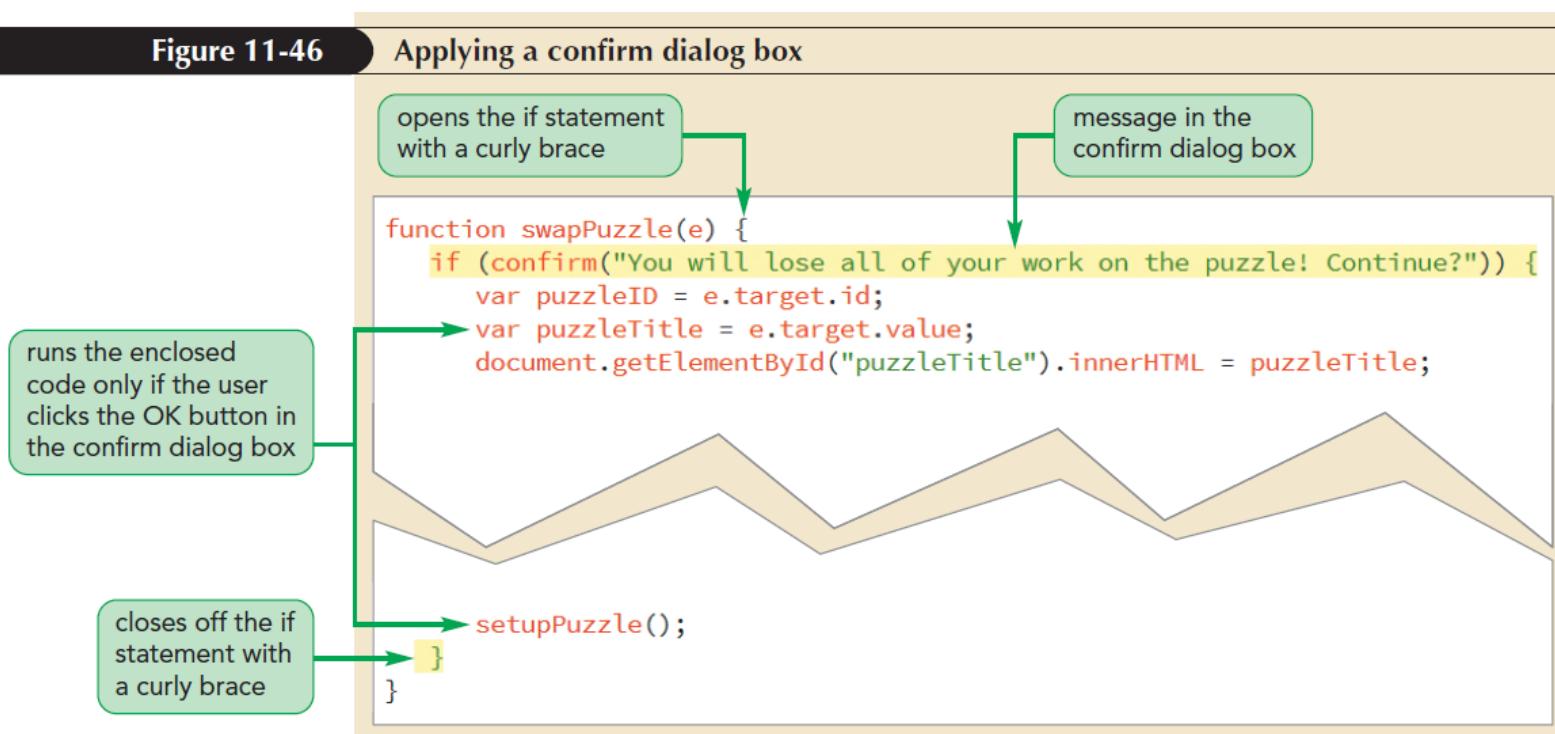
Figure 11-45    JavaScript dialog boxes

Dialog Box	Description	Example
<code>alert(text)</code>	Displays an alert message	<code>alert("Welcome to Hanjie!");</code> 
<code>confirm(text)</code>	Displays a confirmation box, returning a value of <code>true</code> if the user clicks the OK button and <code>false</code> if the Cancel button is clicked	<code>var quitGame = confirm("Quit the Game?");</code> 
<code>prompt(text [, defaultInput])</code>	Displays a prompt box, prompting the user to enter input text, where <code>defaultInput</code> is an optional attribute specifying the default input value; if the user clicks OK the input text is returned, if the Cancel button is clicked a value of <code>null</code> is returned	<code>var gameType = prompt("Enter your Game", "Hanjie");</code> 

# Displaying Dialog Boxes

Figure 11-46

Applying a confirm dialog box



# Displaying Dialog Boxes

Figure 11-50 A successfully completed puzzle

