Software Engineering for WWW Dr. Vinod Dubey SWE642 George Mason University

JavaScript

OBJECTIVES

- **After completing this section, you should be able to**
  - Demonstrate the understanding of key features of JavaScript
  - Create basic **JavaScript code** using **control structures**
  - Develop **JavaScript functions** and use **JavaScript's built-in objects** available in the browsers
  - Use the browser's **document and window** objects, as well as Global functions.

Introduction

- JavaScript is an **interpreted** programming language that **runs** (is executed) **primarily** in web browsers and is used **for creating interactive** and **dynamic elements** within web pages.
- JavaScript is **often used in conjunction** with **HTML** and **CSS** to **create modern, interactive web applications**.
- It was originally developed by **Brendan Eich** at **Netscape** in 1995 and has since become **one of the most widely used programming languages** on the web.

Introduction

- **Key features of JavaScript include:**
  - **Client-Side Scripting**: JavaScript code is primarily executed on the client-side, meaning it **runs on the user's web browser** rather than the web server.
    - However, it can also be used on the server-side through technologies like Node.js.
  - **Dynamic Content**: JavaScript enables the **modification of web page** content **in real-time**, allowing developers **to create interactive elements that respond to user input.**
    - JavaScript interacts with the Document Object Model (DOM) of a web page, allowing developers to modify the structure, content, and styles of HTML elements dynamically.
  - **Event Handling**: JavaScript functions can be used as **event handlers** that allow developers **to respond to various user actions**, such as **clicks**, **mouse movements**, keyboard inputs, and form submissions.
  - **Cross-Browser Compatibility**: JavaScript is **supported by all major web browsers**, making it a versatile language for web development.

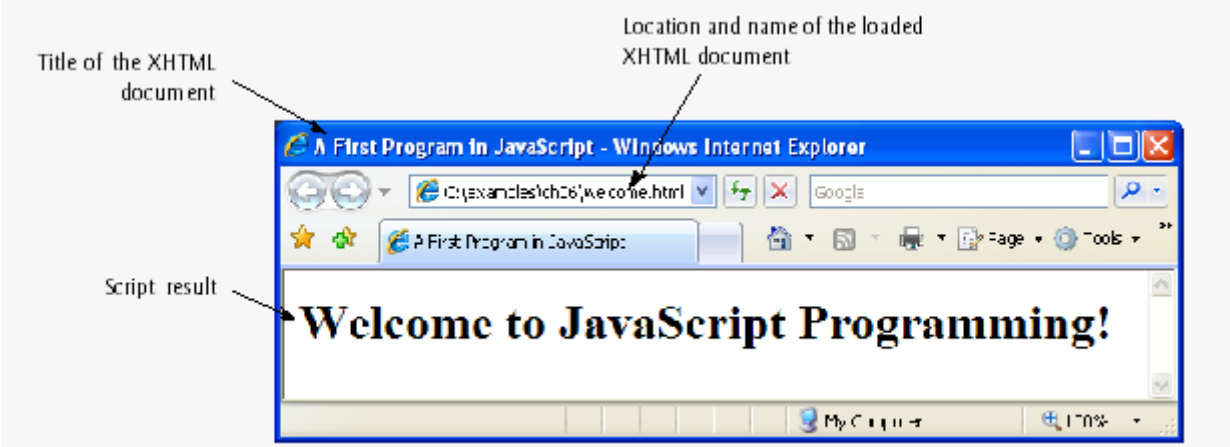
Introduction

- Over time, JavaScript has **evolved** and **expanded beyond just the web browser**, finding use in
  - **server-side** development (with NodeJS),
  - **mobile app** development (e.g., with frameworks like React Native),
  - **game** development
- It has a large and active developer community, offering numerous libraries, **frameworks,** such as **React, Angular, and Vue.js**, and tools that enhance its capabilities and **make web development more efficient**.

A Simple JavaScript example

- Often, JavaScript code appear **in** the **<head> section** of the HTML document and starts with **<script> tag** with **type="text/javascript"**
  - The browser interprets the contents of the <head> section first
- This example uses Browser's **`document` object's writeln()** method **to write an `h1`** element in the html document to display a line of text in Web page
  - The **`document`** object represents the **html document** currently being displayed in the browser

```
8     <head>
9        <title>A First Program in JavaScript</title>
10       <script type = "text/javascript">
11          <!--
12          document.writeln(
13             "<h1>Welcome to JavaScript Programming!</h1>" );
14          // -->
15       </script>
16    </head><body></body>
```



Variables in JavaScript

- **JavaScript** is referred to as a **loosely typed language**
  - JavaScript **doesn't require variables to have a type** before they can be used in a program
  - A variable in JavaScript can **contain a value** of **any data type**
  - Type of a javascript variable is determined by the type of the data it holds
- Keyword `var` is used to declare the names of variables

```
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Class Average Program</title>
      <script type = "text/javascript">
         <!--
         var total; // sum of grades
         var gradeCounter; // number of grades entered
         var grade; // grade typed by user (as a string)
         var gradeValue; // grade value (converted to integer)
         var average; // average of all grades

         // Initialization Phase
         total = 0; // clear total
         gradeCounter = 1; // prepare to loop
```

Variables in JavaScript

- A variable name can be any **valid identifier** consisting of letters, digits, underscores (_) and dollar signs ($) that **does not begin with a digit** and is not a reserved JavaScript keyword.
- Declarations end with a **semicolon (;)**
  - Can be split over several lines, with each variable in the declaration separated by a comma
- When a variable is declared in JavaScript, but is **not given a default value**, it has an **undefined value**.
  - Attempting to use the value of such a variable is normally a logic error.
- Comments
  - **A single-line comment** begins with the characters // and terminates at the end of the line
  - **Multiline comments** begin with delimiter /* and end with delimiter */
    - Comments are ignored by the JavaScript interpreter

MASON

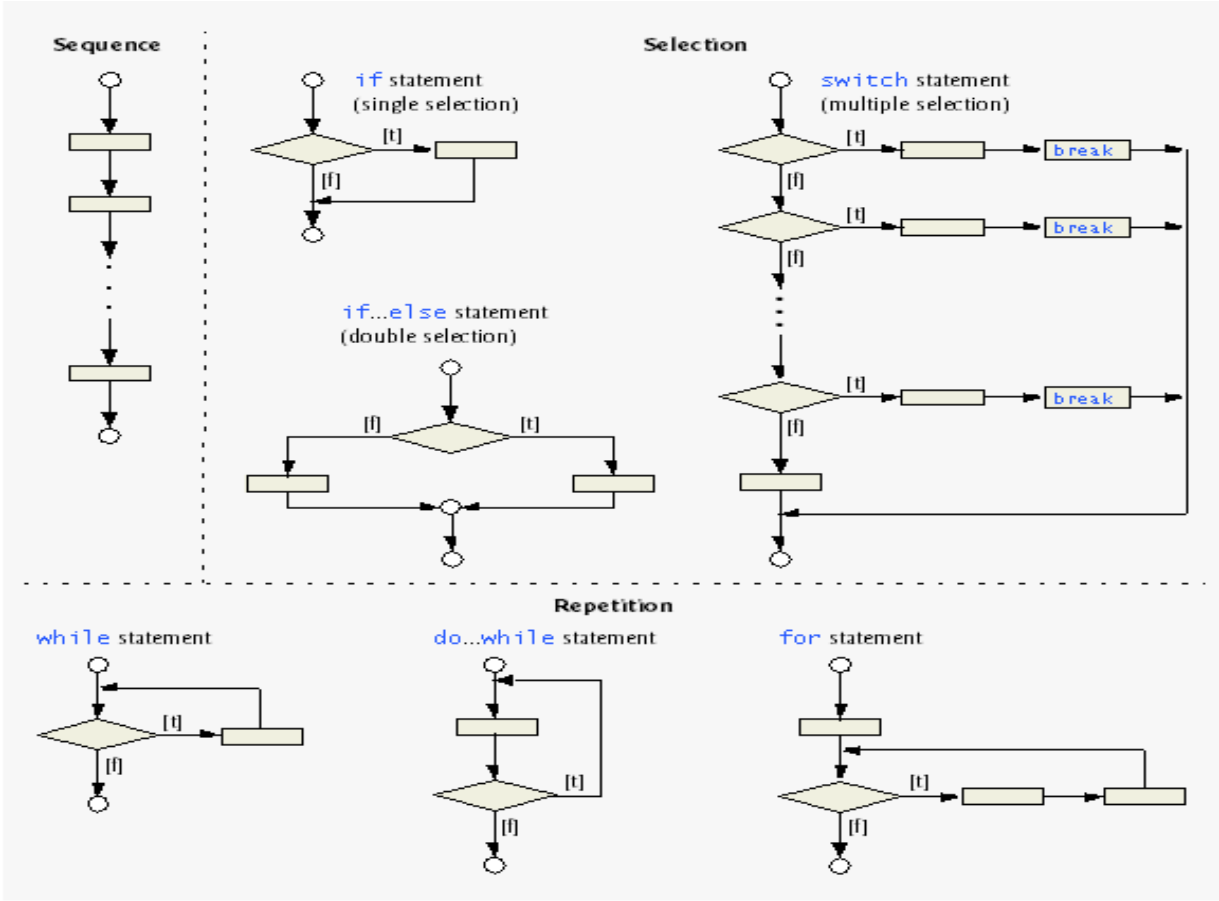
Keywords and Variables in JavaScript

- **Keywords** are words with **special meaning** in JavaScript
  - Keywords can not be used as identifiers
  - Keyword `var` - used to declare the names of variables

JavaScript 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				
abstract	boolean	byte	char	class
const	debugger	double	enum	export
extends	final	float	goto	implements
import	int	interface	long	native
package	private	protected	public	short
static	super	synchronized	throws	transient
volatile				

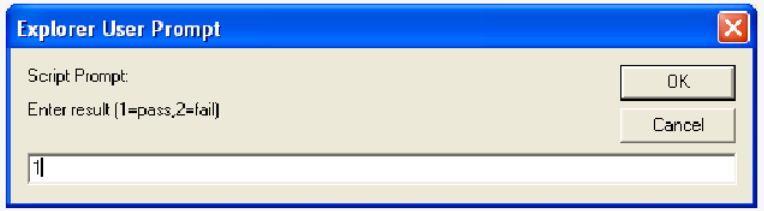
Control Structures in JavaScript

- **Specify the order** in which statements are to be executed in a computer program
- Most programs can be written in terms of three control structures
  - **sequence**
  - **selection**
  - **repetition**
- **Sequential execution -** execute statements in the order they appear in the code
- **Selection** and **repetition include transfer of control**
  - Changing the order in which statements execute

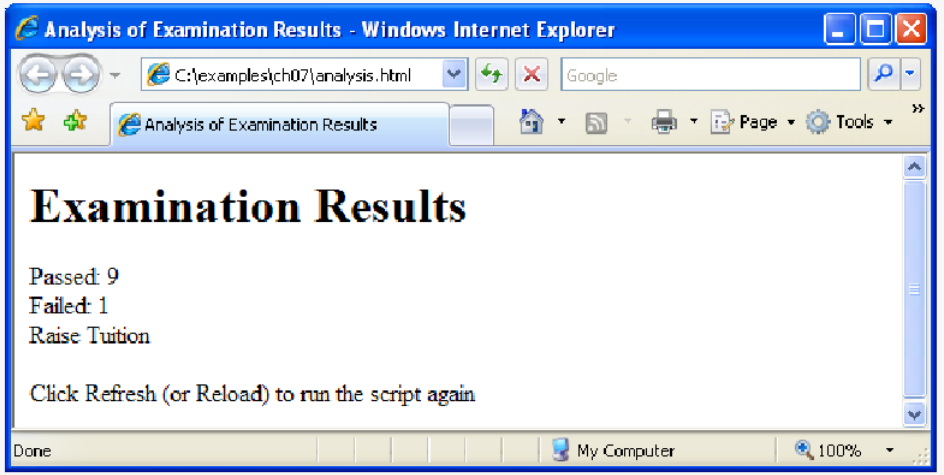


Control Structures Example
Examination-results calculation

```
8     <head>
9        <title>Analysis of Examination Results</title>
10       <script type = "text/javascript">
11          <!--
12          // initializing variables in declarations
13          var passes = 0; // number of passes
14          var failures = 0; // number of failures
15          var student = 1; // student counter
16          var result; // one exam result
17
18          // process 10 students; counter-controlled loop
19          while ( student <= 10 )
20          {
21             result = window.prompt( "Enter result (1=pass,2=fail)", "0" );
22
23             if ( result == "1" )
24                passes = passes + 1;
25             else
26                failures = failures + 1;
27
28             student = student + 1;
29          } // end while
30
31          // termination phase
32          document.writeln( "<h1>Examination Results</h1>" );
33          document.writeln(
34             "Passed: " + passes + "<br />Failed: " + failures );
35
36          if ( passes > 8 )
37             document.writeln( "<br />Raise Tuition" );
38          // -->
39       </script>
40    </head>
41    <body>
42       <p>Click Refresh (or Reload) to run the script again</p>
43    </body>
44 </html>
```



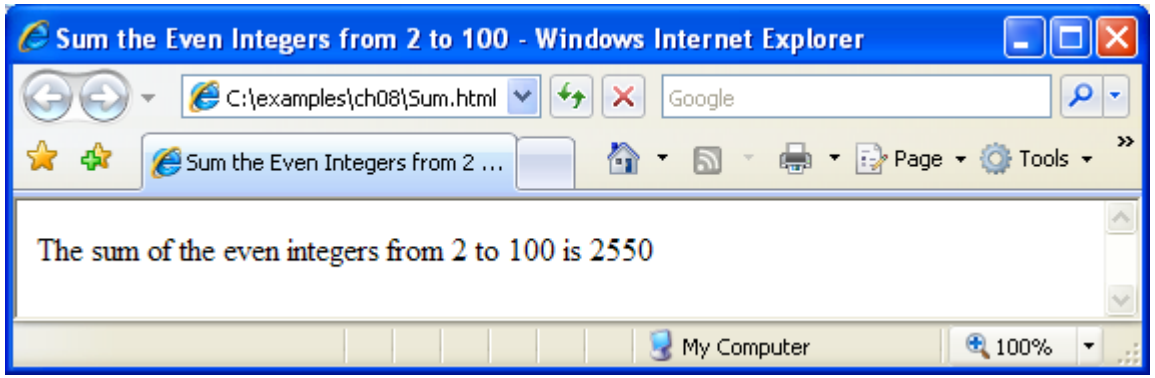
This dialog is displayed 10 times. User input is 1, 2, 1, 1, 1, 1, 1, 1, 1 and 1.



Control Structures Example

Summation with the for-loop repetition structure

```
8     <head>
9        <title>Sum the Even Integers from 2 to 100</title>
10       <script type = "text/javascript">
11          <!--
12          var sum = 0;
13
14          for ( var number = 2; number <= 100; number += 2 )
15             sum += number;
16
17          document.writeln( "The sum of the even integers " +
18             "from 2 to 100 is " + sum );
19          // -->
20       </script>
21    </head><body></body>
```



JavaScript Function – Programmer defined

- In JavaScript, a function is **a block of code** that is designed **to perform a specific task** or calculate a value.
- It is a **reusable piece of code** that can be invoked (called) multiple times from different parts of a program.
- Functions in JavaScript have the following basic structure

```
function functionName(parameter1, parameter2, ...) {
  // Code to be executed
  // It can include statements, expressions, and other function calls
  // Optionally, it can return a value using the return statement
}
```

JavaScript Function – Programmer defined(Cont.)

- Here's an explanation of the **different components of a JavaScript function**
  - **function**: The function keyword is used to declare a function.
  - **functionName**: This is the name given to the function, which can be used to invoke the function later. The name should follow the rules for variable names in JavaScript.
  - **parameters**: Functions can accept zero or more parameters (also known as arguments), which are placeholders for values that are passed into the function when it is called.
  - **code**: The code block enclosed within curly braces {} represents the body of the function. It contains the statements and expressions that define the functionality of the function.
  - **return**: The return statement is used to specify the value that the function should return when it is called.
  - Variables declared in function definitions are **local variables**
    - they can be accessed only in the function in which they are defined
  - A function's **parameters** are considered to be **local variables**

JavaScript Function – Programmer defined

- Here's an example of a simple JavaScript function that calculates the sum of two numbers

```
function addNumbers(a, b) {
  var sum = a + b;
  return sum;
}
```

- To use this function, you can call it with appropriate arguments

```
var result = addNumbers(5, 3);
console.log(result); // Output: 8
```

Program Modules in JavaScript

- JavaScript **programs** are written by **combining** new **functions** that the **programmer writes** with **"prepackaged" functions** and **objects** available in Web browsers
  - JavaScript provides **several built-in objects** that have a rich collection of methods for performing
    - Common **mathematical** calculations
    - **String** manipulations
    - **Date** and time manipulations
    - Manipulations of collections of data called **arrays**
  - The term method implies that a function belongs to a particular object
  - All others are referred to as functions.
- JavaScript is sometimes referred to as an **object-based** programming language

`document` Object

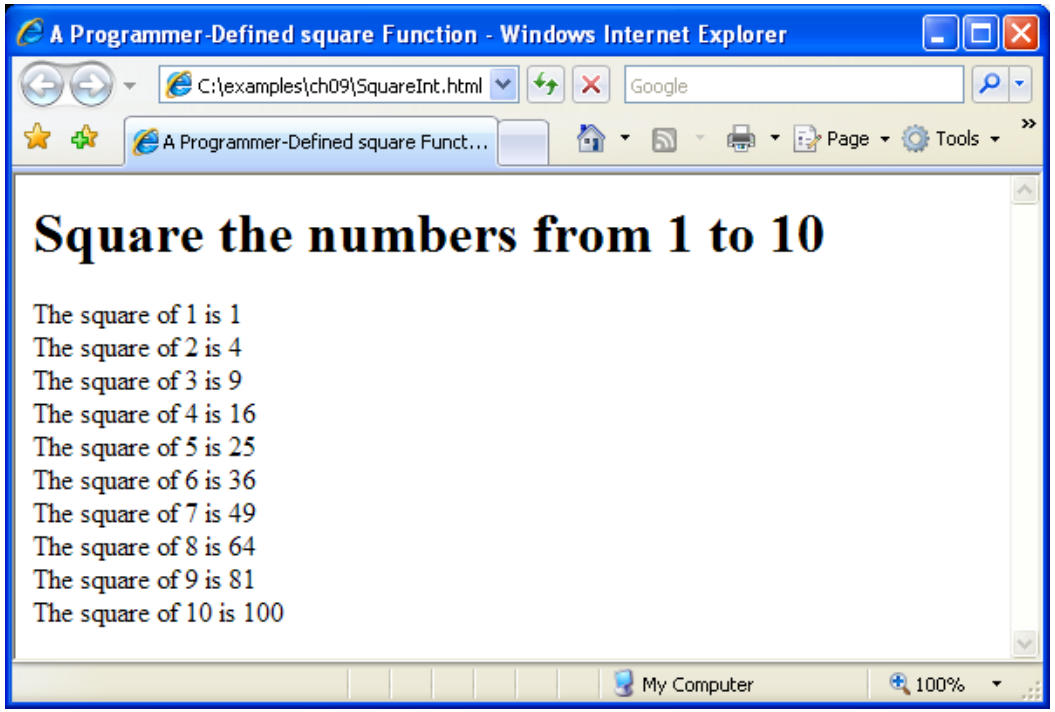
- **`document`** object methods and properties
  - For **manipulating the document** that is **currently visible in the browser** window

`getElementById( id )`	Returns the DOM node representing the XHTML element whose `id` attribute matches *id*.
`write( string )`	Writes the string to the XHTML document as XHTML code.
`writeln( string )`	Writes the string to the XHTML document as XHTML code and adds a newline character at the end.
`cookie`	A string containing the values of all the cookies stored on the user's computer for the current document. See Section 11.9, Using Cookies.
`lastModified`	The date and time that this document was last modified.

Programmer-defined function square()

- document's **writeln()** method writes a String to the html document and adds a new line character at the end.

```
8     <head>
9        <title>A Programmer-Defined square Function</title>
10       <script type = "text/javascript">
11          <!--
12          document.writeln( "<h1>Square the numbers from 1 to 10</h1>" );
13
14          // square the numbers from 1 to 10
15          for ( var x = 1; x <= 10; x++ )
16             document.writeln( "The square of " + x + " is " +
17                square( x ) + "<br />" );
18
19          // The following square function definition is executed
20          // only when the function is explicitly called.
21
22          // square function definition
23          function square( y )
24          {
25             return y * y;
26          } // end function square
27          // -->
28       </script>
29    </head><body></body>
```



window Object

- The **window object** provides **methods for manipulating browser windows**

`open(` *url*, *name*, *options* `)`	Creates a new window with the URL of the window set to *url*, the name set to *name* to refer to it in the script, and the visible features set by the string passed in as *option*.
`prompt(` *prompt*, *default* `)`	Displays a dialog box asking the user for input. The text of the dialog is *prompt*, and the default value is set to *default*.
`close()`	Closes the current window and deletes its object from memory.
`focus()`	This method gives focus to the window (i.e., puts the window in the foreground, on top of any other open browser windows).
`blur()`	This method takes focus away from the window (i.e., puts the window in the background).
`window.document`	This property contains the `document` object representing the document currently inside the window.
`window.closed`	This property contains a boolean value that is set to true if the window is closed, and false if it is not.
`window.opener`	This property contains the `window` object of the window that opened the current window, if such a window exists.

Global object

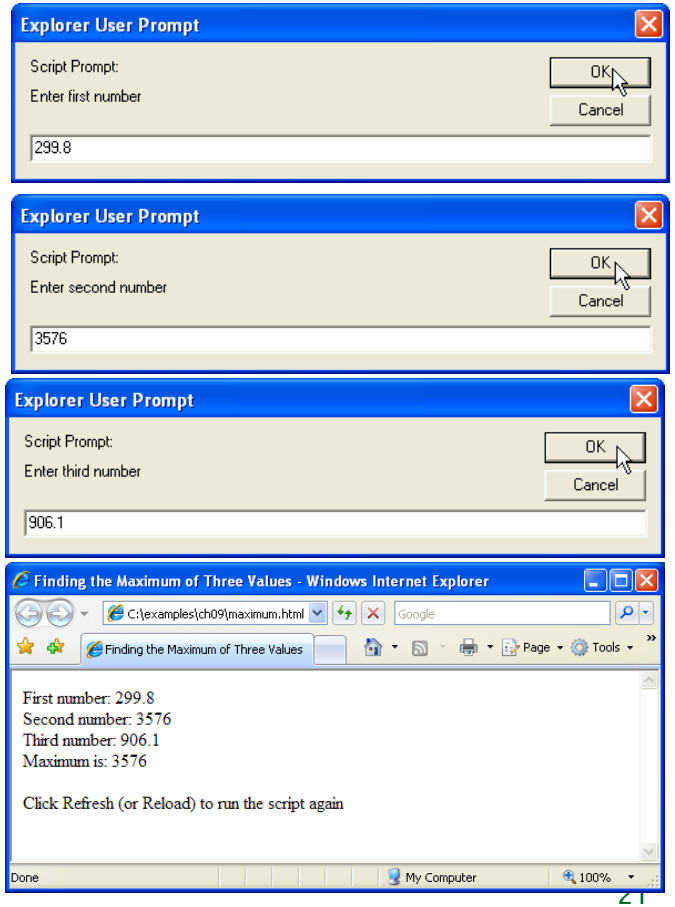
- JavaScript provides **seven global functions** as part of a `Global` object
- Contains
  - all the global variables in the script
  - all the user-defined functions in the script
  - all the **built-in global functions** listed in the **adjacent slide**
- You **do not need to use the `Global` object** directly when using these functions
  - JavaScript uses it for you

`escape`	Takes a string argument and returns a string in which all spaces, punctuation, accent characters and any other character that is not in the ASCII character set (see Appendix D, ASCII Character Set) are encoded in a hexadecimal format (see Appendix E, Number Systems) that can be represented on all platforms.
`eval`	Takes a string argument representing JavaScript code to execute. The JavaScript interpreter evaluates the code and executes it when the `eval` function is called. This function allows JavaScript code to be stored as strings and executed dynamically. [*Note:* It is considered a serious security risk to use eval to process any data entered by a user because a malicious user could exploit this to run dangerous code.]
`isFinite`	Takes a numeric argument and returns `true` if the value of the argument is not `NaN`, `Number.POSITIVE_INFINITY` or `Number.NEGATIVE_INFINITY` (values that are not numbers or numbers outside the range that JavaScript supports)—otherwise, the function returns `false`.
`isNaN`	Takes a numeric argument and returns `true` if the value of the argument is not a number; otherwise, it returns `false`. The function is commonly used with the return value of `parseInt` or `parseFloat` to determine whether the result is a proper numeric value.
`parseFloat`	Takes a string argument and attempts to convert the beginning of the string into a floating-point value. If the conversion is unsuccessful, the function returns NaN; otherwise, it returns the converted value (e.g., `parseFloat( "abc123.45" )` returns `NaN`, and `parseFloat( "123.45abc" )` returns the value `123.45`).
`parseInt`	Takes a string argument and attempts to convert the beginning of the string into an integer value. If the conversion is unsuccessful, the function returns NaN; otherwise, it returns the converted value (e.g., parseInt("abc123") returns NaN, and parseInt("123abc") returns the integer value 123). This function takes an optional second argument, from 2 to 36, specifying the radix (or base) of the number. Base 2 indicates that the first argument string is in binary format, base 8 indicates that the first argument string is in octal format and base 16 indicates that the first argument string is in hexadecimal format. See Appendix E, Number Systems, for more information on binary, octal and hexadecimal numbers.
`unescape`	Takes a string as its argument and returns a string in which all characters previously encoded with `escape` are decoded.

Programmer-defined maximum() function

- The return type of Window's **prompt()** method is **String**. If the user clicks "OK", the input value is returned. The **parseFloat**() takes string argument and convrts it into floating point

```
8     <head>
9        <title>Finding the Maximum of Three Values</title>
10       <script type = "text/javascript">
11          <!--
12          var input1 = window.prompt( "Enter first number", "0" );
13          var input2 = window.prompt( "Enter second number", "0" );
14          var input3 = window.prompt( "Enter third number", "0" );
15
16          var value1 = parseFloat( input1 );
17          var value2 = parseFloat( input2 );
18          var value3 = parseFloat( input3 );
19
20       var maxValue = maximum( value1, value2, value3 );
21
22       document.writeln( "First number: " + value1 +
23          "<br />Second number: " + value2 +
24          "<br />Third number: " + value3 +
25          "<br />Maximum is: " + maxValue );
26
27       // maximum function definition (called from line 20)
28       function maximum( x, y, z )
29       {
30          return Math.max( x, Math.max( y, z ) );
31       } // end function maximum
32       // -->
33    </script>
34 </head>
35 <body>
36    <p>Click Refresh (or Reload) to run the script again</p>
37 </body>
```



Math Object

- **Math** object **methods** and properties/constants allow you to perform many common **mathematical calculations**.

Math.E	Base of a natural logarithm (*e*).	Approximately 2.718
Math.LN2	Natural logarithm of 2	Approximately 0.693
Math.LN10	Natural logarithm of 10	Approximately 2.302
Math.LOG2E	Base 2 logarithm of *e*	Approximately 1.442
Math.LOG10E	Base 10 logarithm of *e*	Approximately 0.434
Math.PI	π—the ratio of a circle's circumference to its diameter	Approximately 3.141592653589793
Math.SQRT1_2	Square root of 0.5	Approximately 0.707
Math.SQRT2	Square root of 2.0	Approximately 1.414

abs(x)	absolute value of x	abs(7.2) is 7.2 abs(0.0) is 0.0 abs(-5.6) is 5.6
ceil(x)	rounds x to the smallest integer not less than x	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	trigonometric cosine of x (x in radians)	cos(0.0) is 1.0
exp(x)	exponential method e^x	exp(1.0) is 2.71828 exp(2.0) is 7.38906
floor(x)	rounds x to the largest integer not greater than x	floor(9.2) is 9.0 floor(-9.8) is -10.0
log(x)	natural logarithm of x (base *e*)	log(2.718282) is 1.0 log(7.389056) is 2.0
max(x, y)	larger value of x and y	max(2.3, 12.7) is 12.7 max(-2.3, -12.7) is -2.3
min(x, y)	smaller value of x and y	min(2.3, 12.7) is 2.3 min(-2.3, -12.7) is -12.7
pow(x, y)	x raised to power y (x^y)	pow(2.0, 7.0) is 128.0 pow(9.0, .5) is 3.0
round(x)	rounds x to the closest integer	round(9.75) is 10 round(9.25) is 9
sin(x)	trigonometric sine of x (x in radians)	sin(0.0) is 0.0
sqrt(x)	square root of x	sqrt(900.0) is 30.0 sqrt(9.0) is 3.0
tan(x)	trigonometric tangent of x (x in radians)	tan(0.0) is 0.0

`String` Object

- A **string** is a **sequence of characters** treated as a single unit
- A string may include letters, digits and various special characters, such as +, –, *, /, and $
- **String literals** or string constants (often called anonymous `String` objects) are written as a sequence of characters in **double quotation marks** or single quotation marks
- Includes a **rich set** of **methods** for string manipulations

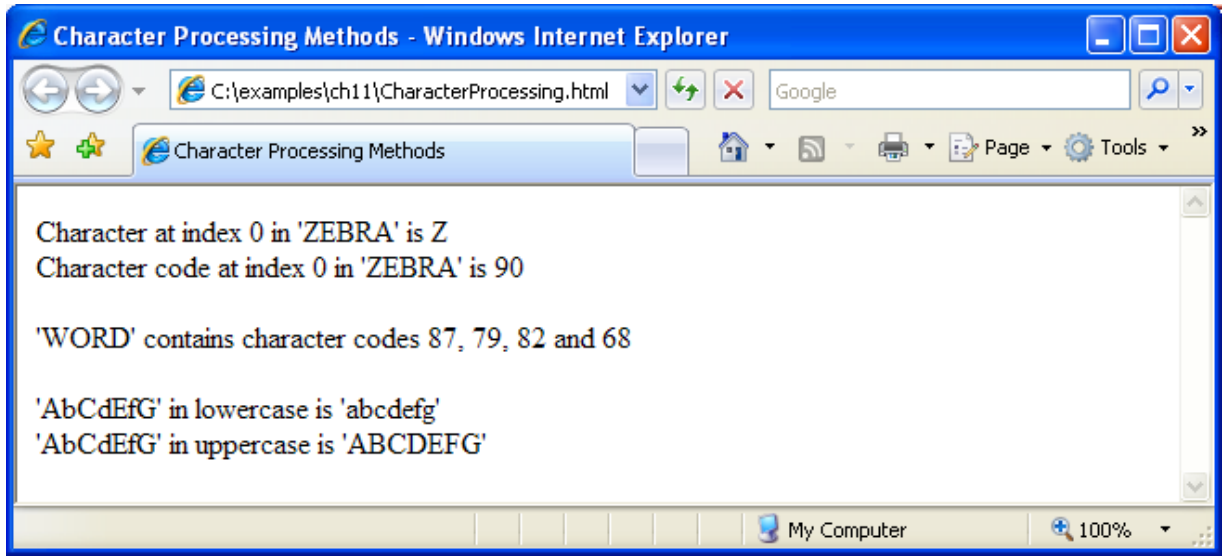
`String` Object methods

`charAt(` *index* `)`	Returns a string containing the character at the specified *index*. If there is no character at the *index*, charAt returns an empty string. The first character is located at *index* 0.
`charCodeAt(` *index* `)`	Returns the Unicode value of the character at the specified *index*, or NaN (not a number) if there is no character at that *index*.
`concat(` *string* `)`	Concatenates its argument to the end of the string that invokes the method. The string invoking this method is not modified; instead a new `String` is returned. This method is the same as adding two strings with the string-concatenation operator + (e.g., s1.`concat(s2)` is the same as s1 + s2).
`fromCharCode(` *value1*`,` *value2*`,` `)`	Converts a list of Unicode values into a string containing the corresponding characters.
`indexOf(` *substring*`,` *index* `)`	Searches for the first occurrence of *substring* starting from position *index* in the string that invokes the method. The method returns the starting index of *substring* in the source string or –1 if *substring* is not found. If the *index* argument is not provided, the method begins searching from index 0 in the source string.
`lastIndexOf(` *substring*`,` *index* `)`	Searches for the last occurrence of *substring* starting from position *index* and searching toward the beginning of the string that invokes the method. The method returns the starting index of *substring* in the source string or –1 if *substring* is not found. If the *index* argument is not provided, the method begins searching from the end of the source string.
`replace(` *searchString*`,` *replaceString* `)`	Searches for the substring *searchString*, and replaces the first occurrence with *replaceString* and returns the modified string, or the original string if no replacement was made.

`slice(` *start*`,` *end* `)`	Returns a string containing the portion of the string from index *start* through index *end*. If the *end* index is not specified, the method returns a string from the *start* index to the end of the source string. A negative *end* index specifies an offset from the end of the string, starting from a position one past the end of the last character (so –1 indicates the last character position in the string).
`split(` *string* `)`	Splits the source string into an array of strings (tokens), where its *string* argument specifies the delimiter (i.e., the characters that indicate the end of each token in the source string).
`substr(` *start*`,` *length* `)`	Returns a string containing *length* characters starting from index *start* in the source string. If *length* is not specified, a string containing characters from *start* to the end of the source string is returned.
`substring(` *start*`,` *end* `)`	Returns a string containing the characters from index *start* up to but not including index *end* in the source string.
`toLowerCase()`	Returns a string in which all uppercase letters are converted to lowercase letters. Nonletter characters are not changed.
`toUpperCase()`	Returns a string in which all lowercase letters are converted to uppercase letters. Nonletter characters are not changed.
Methods that generate XHTML tags	
`anchor(` *name* `)`	Wraps the source string in an anchor element (`<a></a>`) with *name* as the anchor name.
`fixed()`	Wraps the source string in a `<tt></tt>` element (same as `<pre></pre>`).
`link(` *url* `)`	Wraps the source string in an anchor element (`<a></a>`) with *url* as the hyperlink location.
`strike()`	Wraps the source string in a `<strike></strike>` element.
`sub()`	Wraps the source string in a `<sub></sub>` element.
`sup()`	Wraps the source string in a `<sup></sup>` element.

Using String Object methods

```
9     <head>
10       <title>Character Processing Methods</title>
11       <script type = "text/javascript">
12          <!--
13          var s = "ZEBRA";
14          var s2 = "AbCdEfG";
15
16          document.writeln( "<p>Character at index 0 in '" +
17             s + "' is " + s.charAt( 0 ) );
18          document.writeln( "<br />Character code at index 0 in '"
19             + s + "' is " + s.charCodeAt( 0 ) + "</p>" );
20
21          document.writeln( "<p>'" +
22             String.fromCharCode( 87, 79, 82, 68 ) +
23             "' contains character codes 87, 79, 82 and 68</p>" )
24
25          document.writeln( "<p>'" + s2 + "' in lowercase is '" +
26             s2.toLowerCase() + "'" );
27          document.writeln( "<br />'" + s2 + "' in uppercase is '"
28             + s2.toUpperCase() + "'</p>" );
29          // -->
30       </script>
31    </head><body></body>
```



`Date` Object

- **`Date` object** provides methods for **date** and **time manipulations**
- You can **create new Date objects** using **Date constructors**
  - The **`Date` constructor** with no arguments initializes the `Date` object with the **local computer's current date and time**
  - Can create a new `Date` object by supplying arguments to the **`Date` constructor for year, month, date**, hours, minutes, seconds and milliseconds.
    - Hours, minutes, seconds and milliseconds arguments are all optional
- Includes a **rich set** of **methods** for date and time manipulations

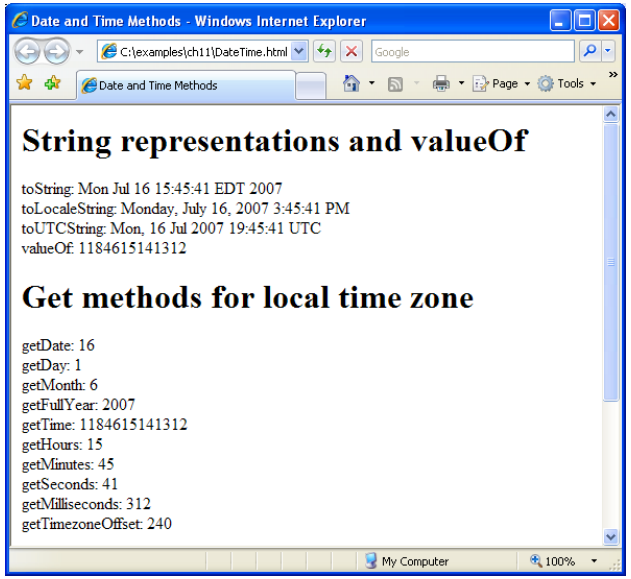
Date object methods

`getDate()` `getUTCDate()`	Returns a number from 1 to 31 representing the day of the month in local time or UTC.
`getDay()` `getUTCDay()`	Returns a number from 0 (Sunday) to 6 (Saturday) representing the day of the week in local time or UTC.
`getFullYear()` `getUTCFullYear()`	Returns the year as a four-digit number in local time or UTC.
`getHours()` `getUTCHours()`	Returns a number from 0 to 23 representing hours since midnight in local time or UTC.
`getMilliseconds()` `getUTCMilliSeconds()`	Returns a number from 0 to 999 representing the number of milliseconds in local time or UTC, respectively. The time is stored in hours, minutes, seconds and milliseconds.
`getMinutes()` `getUTCMinutes()`	Returns a number from 0 to 59 representing the minutes for the time in local time or UTC.
`getMonth()` `getUTCMonth()`	Returns a number from 0 (January) to 11 (December) representing the month in local time or UTC.
`getSeconds()` `getUTCSeconds()`	Returns a number from 0 to 59 representing the seconds for the time in local time or UTC.
`getTime()`	Returns the number of milliseconds between January 1, 1970, and the time in the Date object.
`getTimezoneOffset()`	Returns the difference in minutes between the current time on the local computer and UTC (Coordinated Universal Time).
`setDate(` *val* `)` `setUTCDate(` *val* `)`	Sets the day of the month (1 to 31) in local time or UTC.

`setFullYear(` *y*, *m*, *d* `)` `setUTCFullYear(` *y*, *m*, *d* `)`	Sets the year in local time or UTC. The second and third arguments representing the month and the date are optional. If an optional argument is not specified, the current value in the Date object is used.
`setHours(` *h*, *m*, *s*, *ms* `)` `setUTCHours(` *h*, *m*, *s*, *ms* `)`	Sets the hour in local time or UTC. The second, third and fourth arguments, representing the minutes, seconds and milliseconds, are optional. If an optional argument is not specified, the current value in the Date object is used.
`setMilliSeconds(` *ms* `)` `setUTCMilliseconds(` *ms* `)`	Sets the number of milliseconds in local time or UTC.
`setMinutes(` *m*, *s*, *ms* `)` `setUTCMinutes(` *m*, *s*, *ms* `)`	Sets the minute in local time or UTC. The second and third arguments, representing the seconds and milliseconds, are optional. If an optional argument is not specified, the current value in the Date object is used.
`setMonth(` *m*, *d* `)` `setUTCMonth(` *m*, *d* `)`	Sets the month in local time or UTC. The second argument, representing the date, is optional. If the optional argument is not specified, the current date value in the Date object is used.
`setSeconds(` *s*, *ms* `)` `setUTCSeconds(` *s*, *ms* `)`	Sets the second in local time or UTC. The second argument, representing the milliseconds, is optional. If this argument is not specified, the current millisecond value in the Date object is used.
`setTime(` *ms* `)`	Sets the time based on its argument—the number of elapsed milliseconds since January 1, 1970.
`toLocaleString()`	Returns a string representation of the date and time in a form specific to the computer's locale. For example, September 13, 2007, at 3:42:22 PM is represented as *09/13/07 15:47:22* in the United States and *13/09/07 15:47:22* in Europe.
`toUTCString()`	Returns a string representation of the date and time in the form: *15 Sep 2007 15:47:22 UTC*
`toString()`	Returns a string representation of the date and time in a form specific to the locale of the computer (*Mon Sep 17 15:47:22 EDT 2007* in the United States).
`valueOf()`	The time in number of milliseconds since midnight, January 1, 1970. (Same as `getTime`.)

Using Date object methods

```
8     <head>
9        <title>Date and Time Methods</title>
10       <script type = "text/javascript">
11          <!--
12          var current = new Date();
13
14          document.writeln(
15             "<h1>String representations and valueOf</h1>" );
16          document.writeln( "toString: " + current.toString() +
17             "<br />toLocaleString: " + current.toLocaleString() +
18             "<br />toUTCString: " + current.toUTCString() +
19             "<br />valueOf: " + current.valueOf() );
20
```




```
21          document.writeln(
22             "<h1>Get methods for local time zone</h1>" );
23          document.writeln( "getDate: " + current.getDate() +
24             "<br />getDay: " + current.getDay() +
25             "<br />getMonth: " + current.getMonth() +
26             "<br />getFullYear: " + current.getFullYear() +
27             "<br />getTime: " + current.getTime() +
28             "<br />getHours: " + current.getHours() +
29             "<br />getMinutes: " + current.getMinutes() +
30             "<br />getSeconds: " + current.getSeconds() +
31             "<br />getMilliseconds: " + current.getMilliseconds() +
32             "<br />getTimezoneOffset: " + current.getTimezoneOffset() );
33
34          document.writeln(
35             "<h1>Specifying arguments for a new Date</h1>" );
36          var anotherDate = new Date( 2007, 2, 18, 1, 5, 0, 0 );
37          document.writeln( "Date: " + anotherDate );
38
39          document.writeln( "<h1>Set methods for local time zone</h1>" );
40          anotherDate.setDate( 31 );
41          anotherDate.setMonth( 11 );
42          anotherDate.setFullYear( 2007 );
43          anotherDate.setHours( 23 );
44          anotherDate.setMinutes( 59 );
45          anotherDate.setSeconds( 59 );
46          document.writeln( "Modified date: " + anotherDate );
47          // -->
48       </script>
49    </head><body></body>
```

OBJECTIVES

- **After completing this section, you should be able to**
  - Create and use cookies to store user information
  - Demonstrate an understanding of events and event handlers
  - Create and register event handlers that respond to mouse and keyboard events.
  - To recognize and respond to many common events.

MASON

Using Cookies in JavaScript

- Cookies are small pieces of **data** that are **stored on a user's computer.**
  - Used **to store user's information** or to **track** their **browsing behavior**.
- Your web **browser stores** it **locally** to remember the "**name-value pair**" that **identifies you**.
- **When a user returns to** that **site** in the future, the **web browser returns** that **data** to the **web server** in the form of **a cookie**.
- Here's how you can work with cookies in JavaScript:
  - **Setting a cookie**
  - **Getting a cookie**
  - **Deleting a cookie**

MASON

Using Cookies in JavaScript

- **Setting a cookie**
  - To set a cookie, you can use the **document.cookie** property**.**
  - **Syntax** to set cookie**:** "***identifier=value***" where
    - ***identifier*** is any valid JavaScript variable identifier, and
    - ***value*** is the value of the cookie variable
  - When **multiple cookies** exist for one website**, *identifier-value*** pairs are **separated by semicolons (;)** in the `document.cookie` string
  - ***expires*** property in a cookie string **sets an expiration date**, after which the web browser deletes the cookie
    - If a cookie's expiration date is not set, then the cookie expires by default after the user closes the browser window
  - Here's an **example** that sets a **cookie** named **"username"** with the **value "John Doe"** and an **expiration date** of **one year** from the current date:
    - **document.cookie** = "**username**=John Doe; **expires**=" + new Date(new Date().getFullYear() + 1, new Date().getMonth(), new Date().getDate()).toUTCString();
    - Assignment operator does not overwrite the entire list of cookies, but appends a cookie to the end of it

Using Cookies in JavaScript

- **Getting a cookie**
  - To retrieve the value of a cookie, you can **read** the **document.cookie** property.
  - The document.cookie property **returns** a string that contains **all the cookies** set on the current domain, so you **need to parse** it to get the specific cookie you're interested in.
  - Here's an example that retrieves the value of the "username" cookie->
- **Deleting a cookie**
  - To delete a cookie, you can set its expiration date to a past date. This will cause the browser to remove the cookie.
  - Here's an example that deletes the "username" cookie:

```
function getCookie(name) {
  const cookieArr = document.cookie.split(";");

  for (let i = 0; i < cookieArr.length; i++) {
    const cookiePair = cookieArr[i].split("=");

    if (name === cookiePair[0].trim()) {
      return decodeURIComponent(cookiePair[1]);
    }
  }

  return null;
}

const username = getCookie("username");
console.log(username); // Output: John Doe
```

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC;";
```


MASON

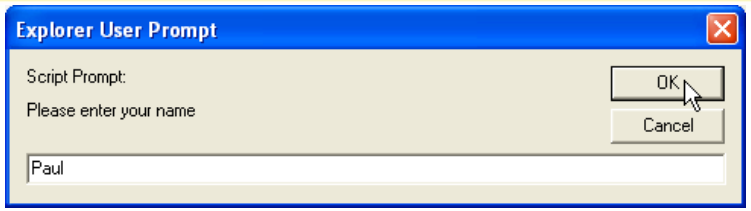
Greeting of the Day - Using cookie to store user identification data

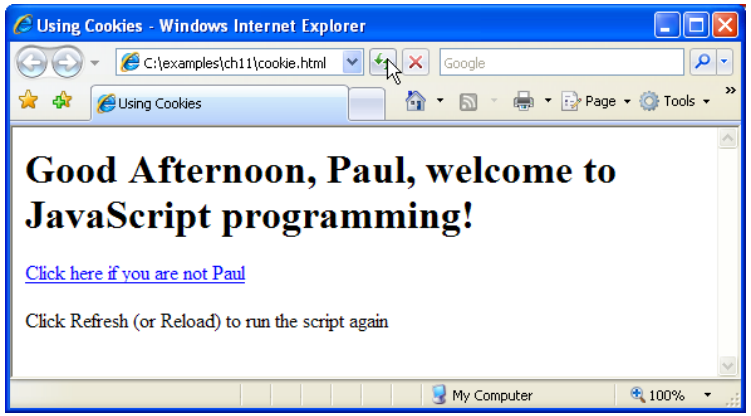
```
8     <head>
9        <title>Using Cookies</title>
10       <script type = "text/javascript">
11          <!--
12          var now = new Date(); // current date and time
13          var hour = now.getHours(); // current hour (0-23)
14          var name;
15
16          if ( hour < 12 ) // determine whether it is morning
17             document.write( "<h1>Good Morning, " );
18          else
19          {
20             hour = hour - 12; // convert from 24-hour clock to PM time
21
22             // determine whether it is afternoon or evening
23             if ( hour < 6 )
24                document.write( "<h1>Good Afternoon, " );
25             else
26                document.write( "<h1>Good Evening, " );
27          } // end else
28
```

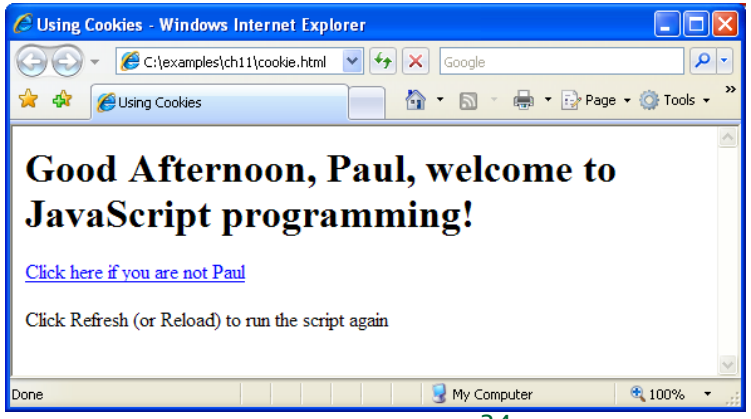
```
29          // determine whether there is a cookie
30          if ( document.cookie )
31          {
32             // convert escape characters in the cookie string to their
33             // English notation
34             var myCookie = unescape( document.cookie );
35
36             // split the cookie into tokens using = as delimiter
37             var cookieTokens = myCookie.split( "=" );
38
39             // set name to the part of the cookie that follows the = sign
40             name = cookieTokens[ 1 ];
41          } // end if
42          else
43          {
44             // if there was no cookie, ask the user to input a name
45             name = window.prompt( "Please enter your name", "Paul" );
46
47             // escape special characters in the name string
48             // and add name to the cookie
49             document.cookie = "name=" + escape( name );
50          } // end else
51
```

Greeting of the Day - Using cookie to store user identification data

```
52          document.writeln(
53             name + ", welcome to JavaScript programming!</h1>" );
54          document.writeln( "<a href = 'javascript:wrongPerson()'> " +
55             "Click here if you are not " + name + "</a>" );
56
57          // reset the document's cookie if wrong person
58          function wrongPerson()
59          {
60             // reset the cookie
61             document.cookie= "name=null;" +
62                " expires=Thu, 01-Jan-95 00:00:01 GMT";
63
64             // reload the page to get a new name after removing the cookie
65             location.reload();
66          } // end function wrongPerson
67
68          // -->
69       </script>
70    </head>
71    <body>
72       <p>Click Refresh (or Reload) to run the script again</p>
73    </body>
```







Even-driven Programming

- In event-driven programming
  - **A user interacts** with a GUI component that **creates events**
  - A script is **notified of the event**
  - The script **processes the event**
- The **function that is called** when an event occurs is known as an **event-handling function** or **event handler.**
- **Events** and **event handling** help make **web applications** more **responsive, dynamic** and **interactive**
- For **example**:
  - An HTML element's **`onclick` attribute** indicates the **action to take when the user** of the HTML document **clicks on the element**
- **Event Handler**
  - **JavaScript functions** that is called **to handle events**
- **Registering** an event handler
  - Refers to **assigning** an **event handler** to **an event** on a DOM node

Registering Event Handlers

- **Two models** for **registering** event handlers**: Inline** and **Traditional**
- **Inline model**
  - Treats **events as attributes** of HTML elements
  - **Assigns** the name of the **function** to an **HTML event attribute**
  - **Value of the event attribute** is a JavaScript **statement/function to be executed** when the event occurs
- **Traditional model**
  - **Assigns** the **name** of the **function** to the **event property** of a **DOM node**
  - The **value of the event property** of a DOM node **is the name of a function** to be called when the event occurs

Common Events

- A list of some events supported by Browsers

`onabort`	Fires when image transfer has been interrupted by user.
`onchange`	Fires when a new choice is made in a `select` element, or when a text input is changed and the element loses focus.
`onclick`	Fires when the user clicks using the mouse.
`ondblclick`	Fires when the mouse is double clicked.
`onfocus`	Fires when a form element gains focus.
`onkeydown`	Fires when the user pushes down a key.
`onkeypress`	Fires when the user presses then releases a key.
`onkeyup`	Fires when the user releases a key.
`onload`	Fires when an element and all its children have loaded.
`onsubmit`	Fires when a form is submitted.
`onunload`	Fires when a page is about to unload.

`onmousedown`	Fires when a mouse button is pressed down.
`onmousemove`	Fires when the mouse moves.
`onmouseout`	Fires when the mouse leaves an element.
`onmouseover`	Fires when the mouse enters an element.
`onmouseup`	Fires when a mouse button is released.
`onreset`	Fires when a form resets (i.e., the user clicks a reset button).
`onresize`	Fires when the size of an object changes (i.e., the user resizes a window or frame).
`onselect`	Fires when a text selection begins (applies to `input` or `textarea`).
`onsubmit`	Fires when a form is submitted.
`onunload`	Fires when a page is about to unload.

MASON

Registering event handlers

The **`getElementById` method** of the document object, used in this example,

- given an `id` as an argument, **finds the HTML element with a matching `id`** attribute and
- returns a JavaScript object representing the element

```
8     <head>
9        <title>Event Registration Models</title>
10       <style type = "text/css">
11          div { padding: 5px;
12                margin: 10px;
13                border: 3px solid #0000BB;
14                width: 12em }
15       </style>
16       <script type = "text/javascript">
17          <!--
18          // handle the onclick event regardless of how it was registered
19          function handleEvent()
20          {
21             alert( "The event was successfully handled." );
22          } // end function handleEvent
23
24          // register the handler using the traditional model
25          function registerHandler()
26          {
27             var traditional = document.getElementById( "traditional" );
28             traditional.onclick = handleEvent;
29          } // end function registerHandler
```

```
30          // -->
31       </script>
32    </head>
33    <body onload = "registerHandler()">
34       <!-- The event handler is registered inline -->
35       <div id = "inline" onclick = "handleEvent()">
36          Inline registration model</div>
37
38       <!-- The event handler is registered by function registerHandler -->
39       <div id = "traditional">Traditional registration model</div>
40    </body>
```

a) The user clicks the **div** for which the event handler was registered using the inline model.

b) The event handler displays an alert dialog.

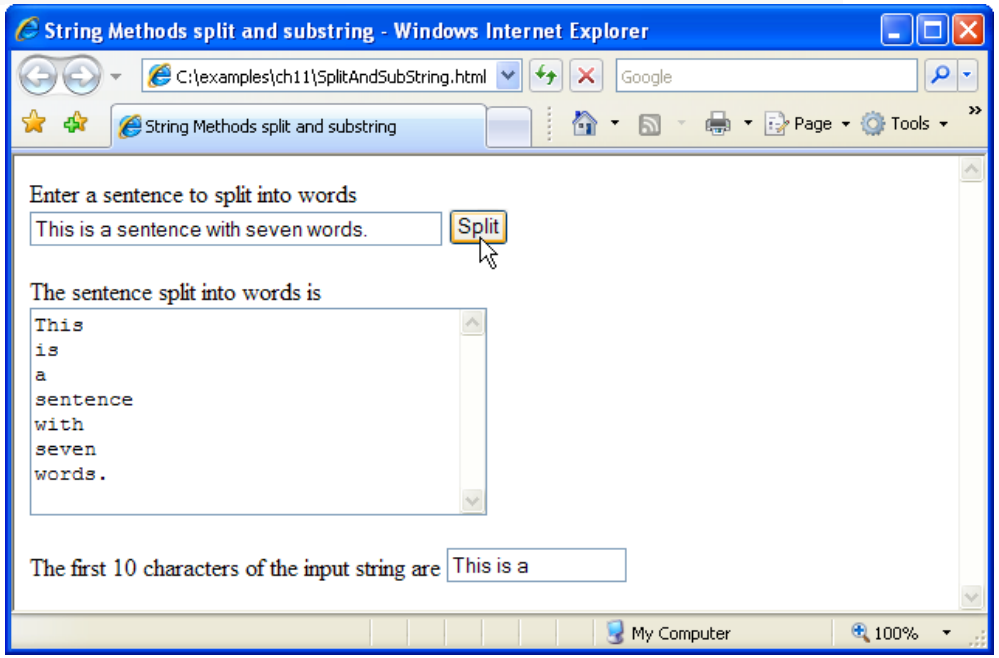
MASON

Example handling onclick event

The **value property** of a JavaScript object, used in this example, represents an HTML **text input element; specifies the text to display** in the text field

```
8     <head>
9        <title>String Methods split and substring</title>
10       <script type = "text/javascript">
11          <!--
12          function splitButtonPressed()
13          {
14             var inputString = document.getElementById( "inputVal" ).value;
15             var tokens = inputString.split( " " );
16             document.getElementById( "output" ).value =
17                tokens.join( "\n" );
18
19             document.getElementById( "outputSubstring" ).value =
20                inputString.substring( 0, 10 );
21          } // end function splitButtonPressed
22          // -->
23       </script>
24    </head>
25    <body>
26       <form action = "">
27          <p>Enter a sentence to split into words<br />
28          <input id = "inputVal" type = "text" size = "40" />
29          <input type = "button" value = "Split"
30             onclick = "splitButtonPressed()" /></p>
31
```

```
32          <p>The sentence split into words is<br />
33          <textarea id = "output" rows = "8" cols = "34">
34          </textarea></p>
35
36          <p>The first 10 characters of the input string are
37          <input id = "outputSubstring" type = "text"
38                size = "15" /></p>
39       </form>
40    </body>
```

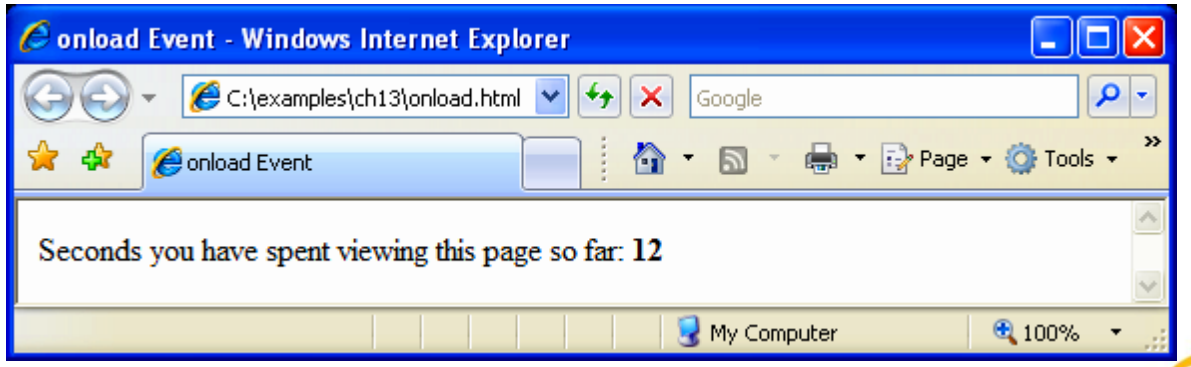


Example handling onload event

- **onload event** fires whenever an element finishes loading successfully
  - The **innerHTML property** of an HTML **container (e.g., div, span, p) element** can be used in a script **to set the contents of the element**

```
8     <head>
9        <title>onload Event</title>
10       <script type = "text/javascript">
11          <!--
12          var seconds = 0;
13
14          // called when the page loads to begin the timer
15          function startTimer()
16          {
17             // 1000 milliseconds = 1 second
18             window.setInterval( "updateTime()", 1000 );
19          } // end function startTimer
20
21          // called every 1000 ms to update the timer
22          function updateTime()
23          {
24             ++seconds;
25             document.getElementById( "soFar" ).innerHTML = seconds;
26          } // end function updateTime
27          // -->
28       </script>
29    </head>
```

```
30    <body onload = "startTimer()">
31       <p>Seconds you have spent viewing this page so far:
32       <strong id = "soFar">0</strong></p>
33    </body>
```



Backup