UC BERKELEY EXTENSION

Database/Application/Programming Courses

Instructor: Michael Kremer, Ph.D.

Course Title: JavaScript and jQuery: An Introduction

Course Subtitle: Building dynamic HTML web pages

Course Number: X452.1

JavaScript and JQuery: An Overview

Instructor: Michael Kremer, Ph.D.

E-mail: mkremer@berkeley.edu

Canvas LMS: http://online.berkeley.edu

5th Edition

ICON KEY	
📝	Note
👉	Hands-on Example
🧑‍🏫	Demo Example
🛑	Warning

TABLE OF CONTENT

Class 1

1. Introduction to JavaScript

1.1 What is JavaScript

JavaScript is a modern, dynamic programming language. It has many parallel attributes, one of the most important one is that it is a scripting language. A scripting language is not a compiled language but an interpreted one.

Note: Dynamic programming language is a term used in computer science to describe a class of high-level programming languages which, at runtime, execute many common programming behaviors that static programming languages perform during compilation.

JavaScript is classified as a prototype-based scripting language with dynamic typing and first-class functions. This mix of features makes it a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.

Despite some naming, syntactic, and standard library similarities, JavaScript and Java are otherwise unrelated and have very different semantics. The syntax (grammar) of JavaScript is actually derived from C, while the semantics (meaning) and design are influenced by the Self and Scheme programming languages.

JavaScript is most commonly used as part of web browsers, whose implementations allow client-side scripts to interact with the user, control the browser, communicate asynchronously, and alter the document content that is displayed.

JavaScript is also used in environments that are not web-based, such as PDF documents, site-specific browsers (SSB), and desktop widgets.

Note: A site-specific browser essentially creates an “app” out of any website, running in a separate browser instance and behaving like a desktop app.

Newer and faster JavaScript virtual machines (VMs) and platforms built upon them have also increased the popularity of JavaScript for server-side web applications. On the client side, JavaScript has been traditionally implemented as an interpreted language, but more recent browsers perform just-in-time compilation.

1.2 History and Standardization of JavaScript

JavaScript was originally developed at Netscape Communications Corporation. Netscape considered for their version of a client-server offering a distributed OS with a portable version of Sun Microsystems' Java providing an environment in which applets could be run. Because Java was a competitor of C++ and aimed at professional programmers, Netscape also wanted a lightweight interpreted language that would complement Java by appealing to nonprofessional programmers, like Microsoft's Visual Basic. This lightweight interpreted language was JavaScript.

Although it was developed under the name Mocha, the language was officially called LiveScript when it first shipped in beta releases of Netscape Navigator 2.0 in September 1995, but it was renamed JavaScript when it was deployed in the Netscape browser version 2.0B3.

Microsoft Windows script technologies including VBScript and JScript were released in 1996. JScript, a part of Netscape's JavaScript, was released on July 16, 1996 and was part of Internet Explorer 3, as well as being available server-side in Internet Information Server. There were major differences between the Netscape and IE browsers. These differences made it difficult for designers and programmers to make a single website work well in both browsers leading to the use of 'best viewed in Netscape' and 'best viewed in Internet Explorer' logos that characterized these early years of the browser wars.

JavaScript began to acquire a reputation for being one of the roadblocks to a cross-platform and standards-driven web, and hence there were obstacles to a widespread acceptance of JavaScript on the web.

In November 1996, Netscape announced that it had submitted JavaScript to ECMA International for consideration as an industry standard, and subsequent work resulted in the standardized version named ECMAScript.

In June 1997, ECMA International published the first edition of the ECMA-262 specification. In June 1998, some modifications were made to adapt it to the ISO/IEC-16262 standard, and the second edition was released. The third edition of ECMA-262 was published on December 1999. Development of the fourth edition of the ECMAScript standard was never completed. The fifth edition was released in December 2009. The current edition of the ECMAScript standard is 8, released in June 2017.

JavaScript has become one of the most popular programming languages on the Web. Initially, however, many professional programmers unfairly criticize the language because its target audience consisted of Web authors and other such "amateurs". The advent of Ajax returned JavaScript to the spotlight and brought more professional programming attention. The result was a proliferation of comprehensive frameworks and libraries, improved JavaScript programming practices, and increased usage of JavaScript outside Web browsers, as seen by the proliferation of server-side JavaScript platforms.

1.3 Features of JavaScript

As mentioned earlier, JavaScript is a multi-paradigm language exhibiting many important programming language modern features as described below.

Imperative Programming Language

JavaScript is first and foremost an imperative programing language (as opposed to declarative programming language). Imperative programming is a programming paradigm that describes computation in terms of statements that change a program state. In much the same way that the imperative mood in natural languages expresses commands to take action, imperative programs define sequences of commands for the computer to perform. Imperative programming is focused on describing how a program operates (declarative programming focuses on the what and not the how, for example SQL).

Structured Programming Language

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures and for and while loops—in contrast to using simple tests and jumps such as the goto statement which could lead to "spaghetti code" which is difficult both to follow and to maintain.

Dynamic

As in most scripting languages, types are associated with values, not with variables. For example, a variable x could be bound to a number, then later rebound to a string. JavaScript supports various ways to test the type of an object.

JavaScript is almost entirely object-based. JavaScript objects are associative arrays, augmented with prototypes (see below). Object property names are string keys. Properties and their values can be added, changed, or deleted at run-time. JavaScript has a small number of built-in objects such as Function and Date.

Functional

Functions in JavaScript are first-class objects; they are objects themselves and can be assigned into a variable. As such, they have properties and methods, such as call() and .bind(). They are also higher order functions, which means a function argument can be another function. JavaScript also supports anonymous functions.

Prototype-based object-oriented programming

JavaScript uses prototypes where many other object-oriented languages use classes for inheritance. It is possible to simulate many class-based features with prototypes in JavaScript.

1.4 Core vs. Client-Side JavaScript

JavaScript is a hosted programming language, meaning it does not exist in its own and must be executed in a host environment. The most common host environment for JavaScript is a web browser.

The core JavaScript language defines a minimal API (Application Programming Interface) for working with text, arrays, dates, and regular expressions but does not include any input or output functionality.

Input and output (and more sophisticated features such as networking, storage, and graphics) are the responsibility of the host environment within JavaScript is embedded. This is client-side JavaScript.

We will in this course first explore the Core JavaScript language without regards to a host environment for reason of simplicity. Therefore, no specific other skills are needed.

During the second part of this course, we will use JavaScript within a web browser. For this section, we do need some HTML and CSS skills.

At the end of this course, we will cover the extensive JavaScript libraries, and take a more detailed look at jQuery and its use.

To learn, write, and execute JavaScript, we need an environment to test JavaScript. Most browsers have built-in tools that you can use to execute JavaScript. Below is a list of the most common JavaScript tools:

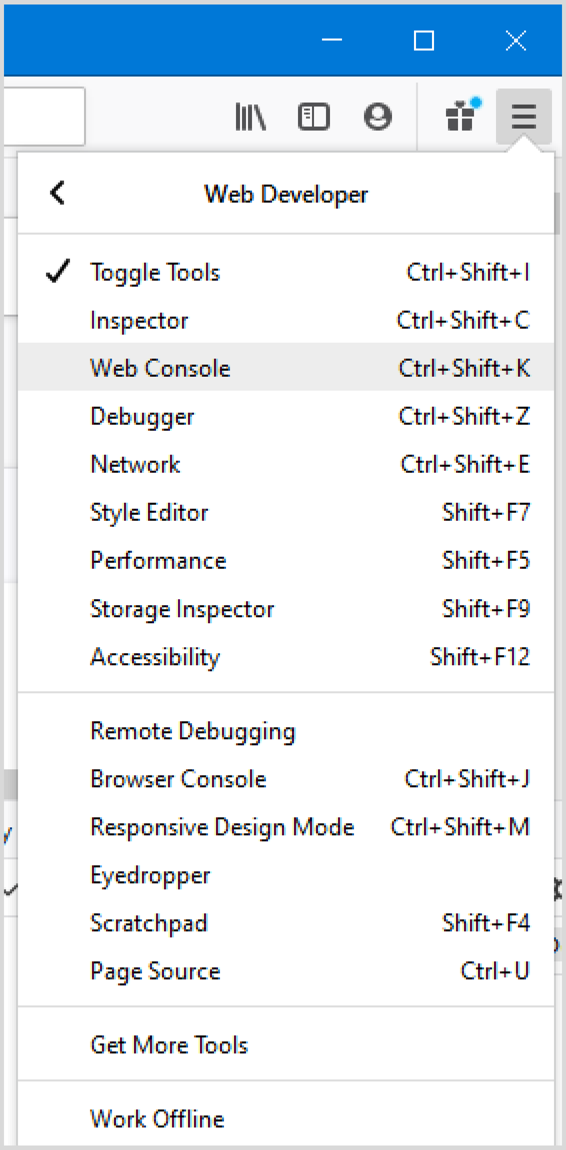
- Firefox→ Console, Developer Tools) (F12), also multi-line editor
- Internet Explorer → Console, Developer Tools (F12)
- Chrome → Console, More Tools (F12)

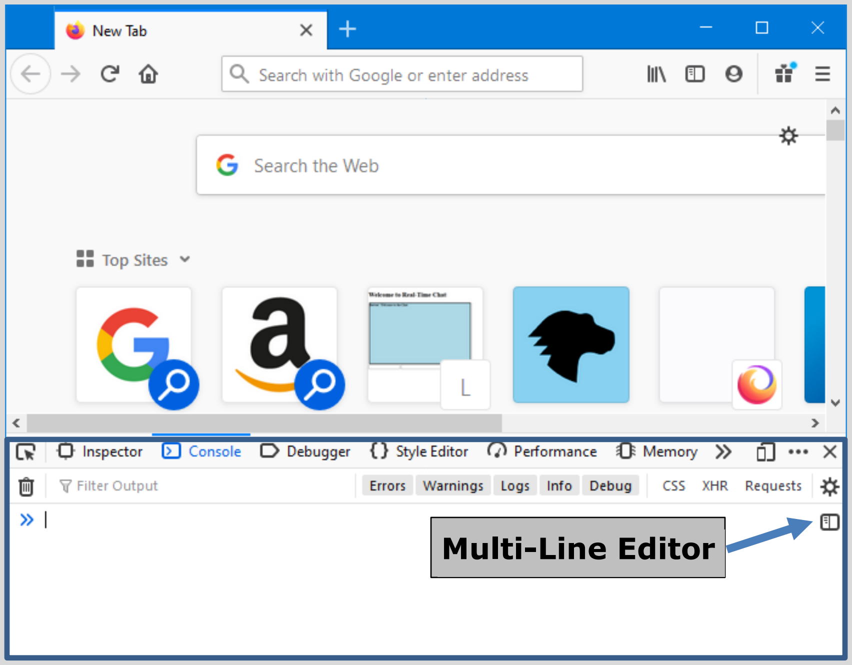
The hands-on examples in this class are written and executed in Firefox multi-line editor or notepad++. Additional examples are provided through the DemoExamples application.

There are also countless numbers of authoring tools for JavaScript. You could use the above web browser tools, you could simply use Windows Notepad, or download Notepad++. Again, there are many others, such as Visual Studio, Eclipse, etc.

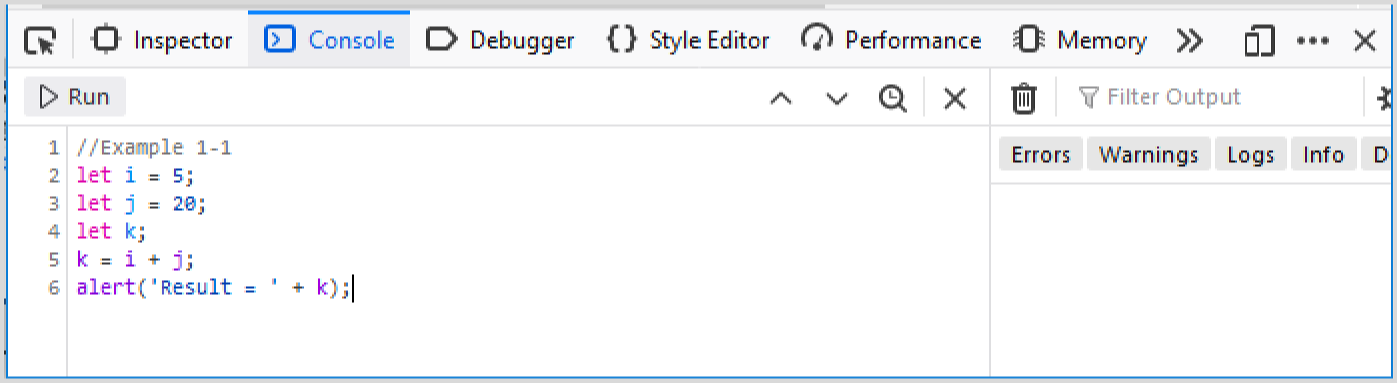
Example 1-1: Executing Hands-On Examples

1. Open Firefox web browser.
2. Open the menu (top right corner).
3. Select Developer Tools.
4. Then select Web Console (Ctrl + Shift + K).
5. The console opens as a docked window below the current page as shown below:





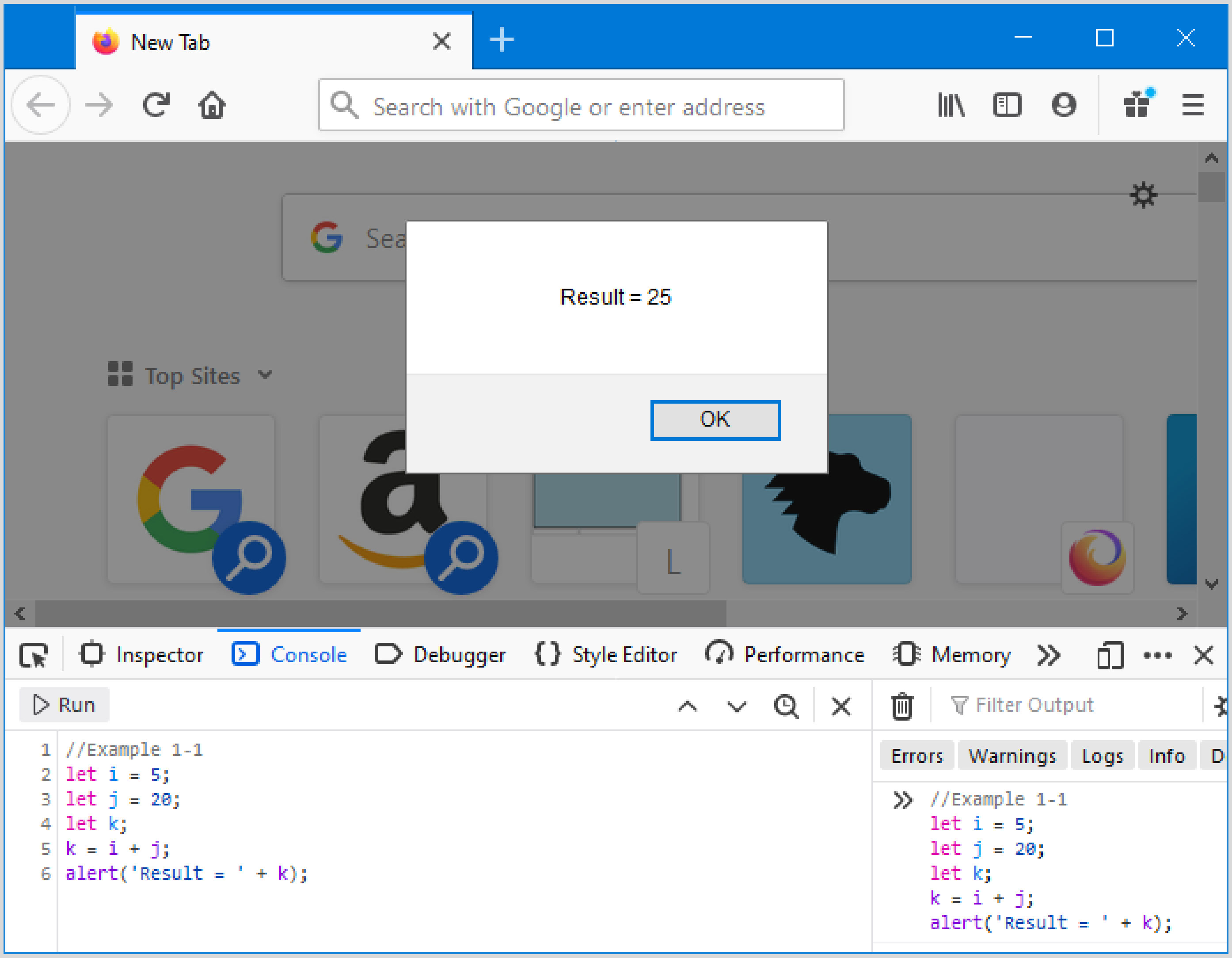
6. Click on the button on the right side (Multi-Line Editor and type the following code:



```
//Example 1-1
let i = 5;
let j = 20;
let k;
k = i + j;
alert('Result = ' + k);
```

Example 1-1 (continued):

7. Click on the button Run to execute this script.
8. You should see an alert dialog box displayed in your browser.

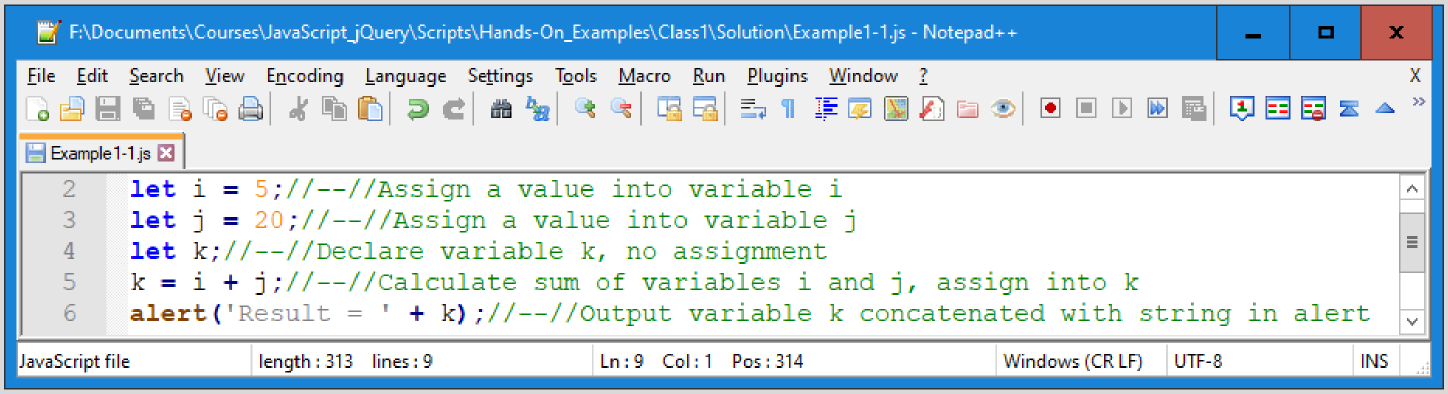


```
//Example 1-1
let i = 5;
let j = 20;
let k;
k = i + j;
alert('Result = ' + k);
```

```
>> //Example 1-1
   let i = 5;
   let j = 20;
   let k;
   k = i + j;
   alert('Result = ' + k);
```

Example 1-1 (continued):

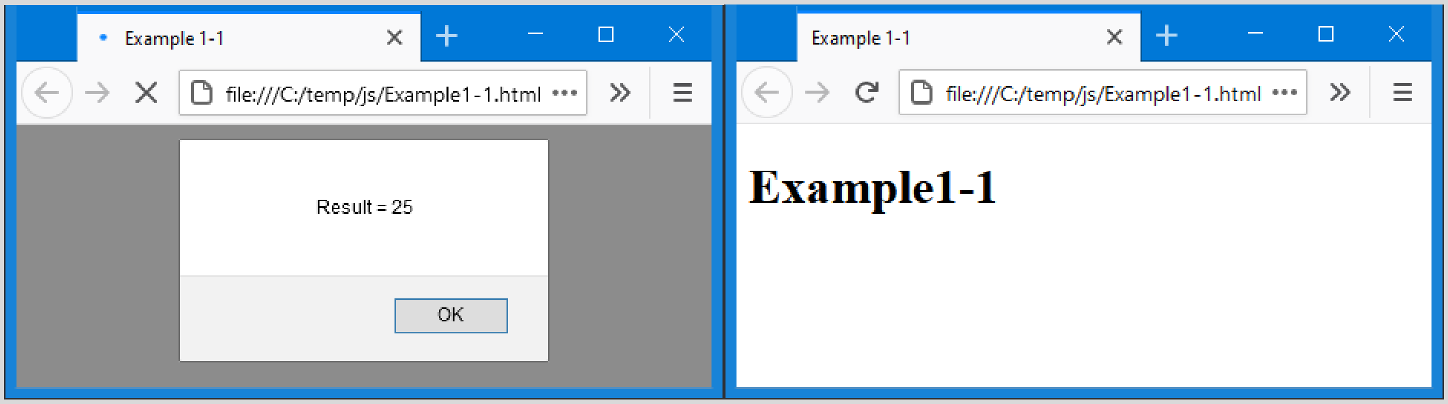
9. Alternatively, you may use the class files for each module to run the JavaScript examples by following these steps:
   a) Navigate to module 1 in Canvas, click on tab Lecture
   b) Download the Class Files.zip file.
   c) Unzip the file into a folder of your choice on your local computer.
   d) Open the file Example1-1.js in the Lecture folder and type the same code example into this file.
   e) Save the JavaScript file and now open the corresponding html file in your browser.
   f) You should see now the same alert dialog box as shown above.



F:\Documents\Courses\JavaScript_jQuery\Scripts\Hands-On_Examples\Class1\Solution\Example1-1.js - Notepad++

```
let i = 5;//--//Assign a value into variable i
let j = 20;//--//Assign a value into variable j
let k;//--//Declare variable k, no assignment
k = i + j;//--//Calculate sum of variables i and j, assign into k
alert('Result = ' + k);//--//Output variable k concatenated with string in alert
```

JavaScript file length : 313 lines : 9 Ln : 9 Col : 1 Pos : 314 Windows (CR LF) UTF-8 INS



file:///C:/temp/js/Example1-1.html

Result = 25

OK

Example1-1

Note: In the previous example, we have already used an output mechanism (alert) that is part of the host environment (browser) and not the core JavaScript language.

The core JavaScript language includes the following basic components:

- Lexical structure
- Types, Values, Variables
- Expressions and Operators
- Statements
- Objects
- Arrays
- Functions
- Pattern Matching with Regular Expressions

In this course, we will then cover the client-side JavaScript technology in modern web browsers:

- JavaScript in Web Browsers
- Window object
- DOM object
- Scripting CSS
- Handling Events
- jQuery library

2. Lexical Structure

The lexical structure of a programming language is a set of basic rules that specify how you write programs in this particular language. This structure specifies the lowest level of syntax and structure used in the programming language

2.1 Character Set

JavaScript supports the Unicode character set. Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. The Unicode Standard, the latest version of Unicode contains a repertoire of more than 120,000 characters covering 129 modern and historic scripts, as well as multiple symbol sets. The latest version of JavaScript requires Unicode 3 or later.

There are two ways in which JavaScript handles Unicode source code:

- internally (during parsing of JavaScript code)
- externally (while loading a file using <script> tag)

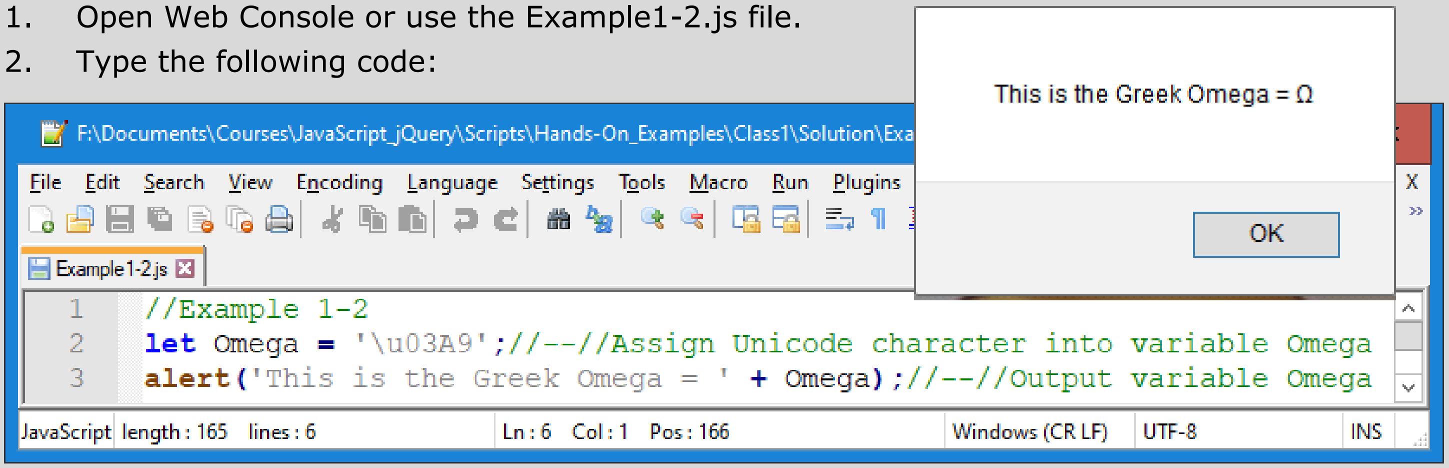
Internally

Internally, JavaScript source code is treated as a sequence of UTF-16 code units. In identifiers, string literals, and regular expression literals, any code unit can also be expressed via a Unicode escape sequence \uHHHH, where HHHH are four hexadecimal digits.

Note: UTF stands for **U**nicode **T**ransformation **F**ormat and is an algorithmic mapping from every Unicode code point to a unique byte sequence.

Example 1-2: Using Unicode character set

1. Open Web Console or use the Example1-2.js file.
2. Type the following code:



F:\Documents\Courses\JavaScript_jQuery\Scripts\Hands-On_Examples\Class1\Solution\Exa

```
//Example 1-2
let Omega = '\u03A9';//--//Assign Unicode character into variable Omega
alert('This is the Greek Omega = ' + Omega);//--//Output variable Omega
```

JavaScript length : 165 lines : 6 Ln : 6 Col : 1 Pos : 166 Windows (CR LF) UTF-8 INS

This is the Greek Omega = Ω

3. Execute the code by clicking on the Run button or running Example1-2.html.
4. You will see the dialog boxes as shown to the right:

That means that you can use Unicode characters in literals and variable names, without leaving the ASCII range in the source code.

Demo 1-1: Using Unicode character set

Execute the demo example by using Demo Example application.

Externally

While UTF-16 is used internally, JavaScript source code is usually not stored in that format. When a web browser loads a source file via a <script> tag, it determines the encoding as follows:

- If the file starts with a byte-order mark (BOM), the encoding is a UTF variant, depending on what BOM is used
- Otherwise, if the file is loaded via HTTP(S), then the Content-Type header can specify an encoding, via the charset parameter. For example: Content-Type: application/javascript; charset=utf-8
- Otherwise, if the <script> tag has the attribute charset, then that encoding is used
- Otherwise, the encoding of the document is used, in which the `<script>` tag resides

Note: The byte order mark (BOM) is a piece of information used to signify that a text file employs Unicode encoding, while also communicating the text stream’s endianness. The BOM is not interpreted as a logical part of the text stream itself, but is rather an invisible indicator at its head.

Note: In almost all modern embedded systems, memory is organized into bytes. CPUs, however, process data as 8-, 16- or 32-bit words. As soon as this word size is larger than a byte, a decision needs to be made with regard to how the bytes in a word are stored in memory. There are two obvious options and a number of other variations. The property that describes this byte ordering is called “endianness” (or, sometimes, “endianity”).

Case Sensitivity

JavaScript is a case-sensitive language. This means that language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters.

The while keyword, for example, must be typed "while", not "While" or "WHILE". Similarly, online, Online, OnLine, and ONLINE are four distinct variable names.

Note, however, that HTML is not case-sensitive. Because of its close association with client-side JavaScript, this difference can be confusing.

Many JavaScript objects and properties have the same names as the HTML tags and attributes they represent. While these tags and attribute names can be typed in any case in HTML, in JavaScript they typically must be all lowercase.

For example, the HTML onclick event handler attribute is commonly specified as onClick in HTML, but it must be referred to as onclick in JavaScript code.

White Space, Line Breaks, and Format Control Characters

JavaScript ignores spaces that appear between tokens (constants, identifiers, operators, reserved words, and separators) in programs. For the most part, JavaScript also ignores line breaks. Because you can use spaces and newlines freely in your programs, you can format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

JavaScript also recognizes the following characters as whitespace:

- regular space character (\u0020), tab (\u0009), vertical tab (\u000B), form feed (\u000C), nonbreaking space (\u00A0), byte order mark (\uFEFF), and any character in Unicode category Zs (Separator, Space)

JavaScript recognizes the following characters as line terminators (A carriage return followed by a line feed sequence is treated as a single line terminator):

- line feed (\u000A), carriage return (\u000D), line separator (\u2028), and paragraph separator (\u2029)

Unicode format control characters (category Cf), such as RIGHT-TO-LEFT MARK (\u200F) and LEFT-TO-RIGHT MARK (\u200E), control the visual presentation of the text they occur in. They are important for the proper display of some non-English languages and are allowed in JavaScript comments, string literals, and regular expression literals, but not in the identifiers (e.g., variable names) of a JavaScript program. As noted above, the byte order mark format control character (\uFEFF) is treated as a space character

2.2 Comments

JavaScript supports two styles of comments:

- Any text between a // and the end of a line is treated as a comment and is ignored by JavaScript
- Any text between the characters /* and */ is also treated as a comment; these comments may span multiple lines but may not be nested

The following lines of code are all legal JavaScript comments:

Syntax:

```
// This is a single-line comment.
/* This is also a comment */ // and here is another comment.
/*
This is yet another comment.
It has multiple lines.
*/
```

2.3 Literals

A literal is a data value that appears directly in a program.

The following are all literals followed by comments:

- 12 // The number twelve
- 1.2 // The number one point two
- "hello world" // A string of text
- 'Hi' // Another string
- true // A Boolean value
- false // The other Boolean value
- /javascript/gi // A "regular expression" literal (for pattern matching)
- null // Absence of an object or value

We will cover more details on numeric and string literals later in this course.

More complex expressions can serve as array and object literals:

- {x:1, y:2} // An object initializer
- [1,2,3,4,5] // An array initializer

2.4 Identifiers and Reserved Words

An identifier is simply a name. In JavaScript, identifiers are used to name variables and functions and to provide labels for certain loops in JavaScript code. A JavaScript identifier must comply with the following rules:

- Must begin with a letter, an underscore (_), or a dollar sign ($)
- Subsequent characters can be letters, digits, underscores, or dollar signs

Warning: Digits are not allowed as the first character so that JavaScript can easily distinguish identifiers from numbers.

These are all legal identifiers:

- i
- my_variable_name
- v13
- _dummy
- $str

For portability and ease of editing, it is common to use only ASCII letters and digits in identifiers.

Note, however, that JavaScript allows identifiers to contain letters and digits from the entire Unicode character set. This allows programmers to use variable names from non-English languages and also to use mathematical symbols:

- let sí = true;
- let π = 3.14;

Like any language, JavaScript reserves certain identifiers for use by the language itself. These "reserved words" cannot be used as regular identifiers.

JavaScript reserves a number of identifiers as the keywords of the language itself. You cannot use those words as identifiers in your programs. Some basic keywords are listed below:

- break, case, if, else, void, while

JavaScript also reserves certain keywords that are not currently used by the language but which might be used in future versions. Some examples from ECMAScript 6 version words:

- class, const, import

When JavaScript is run in strict mode (see later in this course), additional keywords may not be used, such as:

- implement, private, public

ECMAScript 3 reserved all the keywords of the Java language, and although this has been relaxed in ECMAScript 5, you should still avoid all of those identifiers if you plan to run your code under an ECMAScript 3 implementation of JavaScript. Some examples include:

- Boolean, enum, interface, protected, static, throws

JavaScript predefines a number of global variables and functions, and you should avoid using their names for your own variables and functions, such as:

- Arguments, Infinity, Object

Keep in mind that JavaScript implementations may define other global variables and functions, and each specific JavaScript embedding (client-side, server-side, etc.) will have its own list of global properties. See the Window object for a list of the global variables and functions defined by client-side JavaScript.

2.5 Statement-Ending Characters

Like many programming languages, JavaScript uses the semicolon (;) to separate statements from each other. This is important to make the meaning of your code clear: without a separator, the end of one statement might appear to be the beginning of the next, or vice versa.

In JavaScript, you can usually omit the semicolon:

- Between two statements if those statements are written on separate lines
- At the end of a program
- If the next token in the program is a closing curly brace (})

Note: Many JavaScript programmers (and the code in this book) use semicolons to explicitly mark the ends of statements, even where they are not required.

Another style is to omit semicolons whenever possible, using them only in the few situations that require them. Whichever style you choose, there are a few details you should understand about optional semicolons in JavaScript.

Code Example:

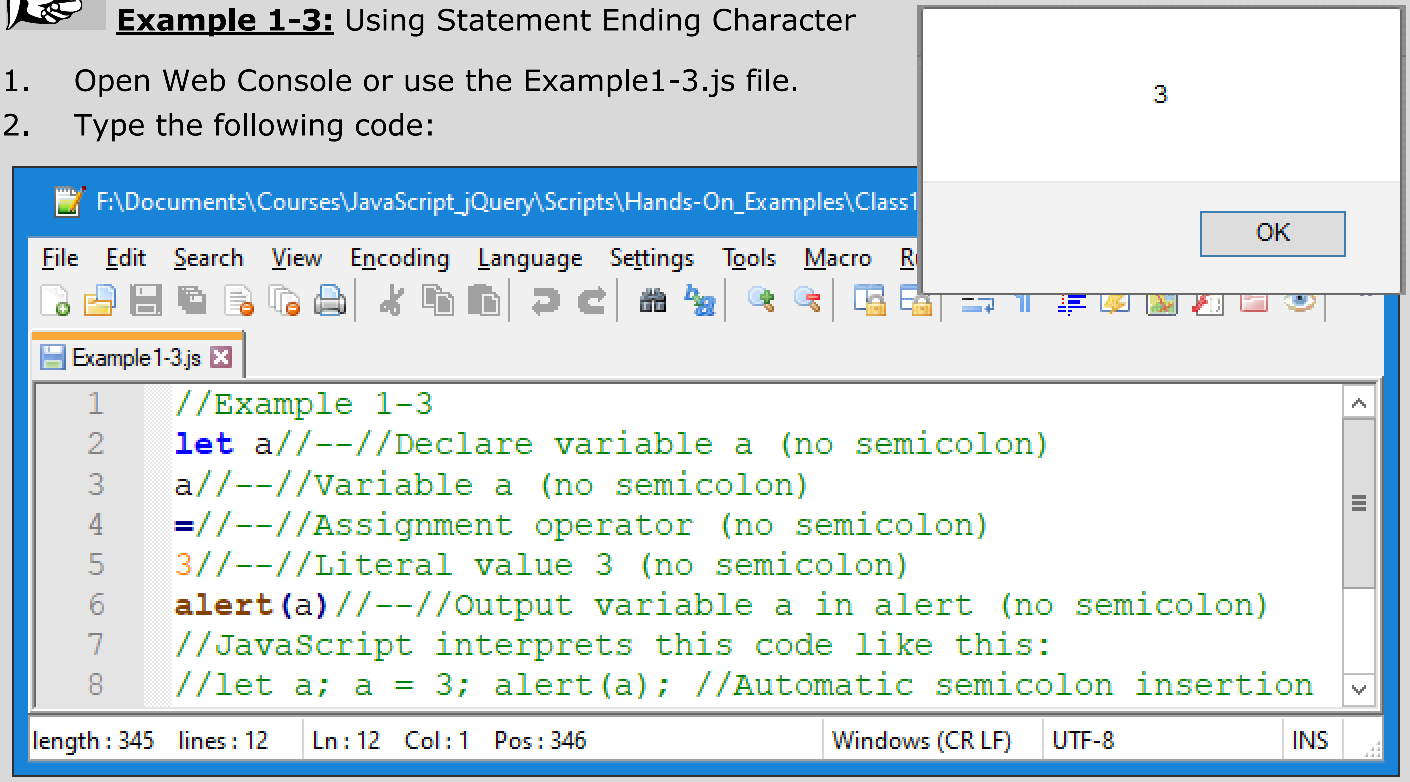
```
a = 3; //Semicolon could be omitted since statements appear on separate lines
b = 4;
//Written as follows, however, the first semicolon is required:
a = 3; b = 4;
```

Note that JavaScript does not treat every line break as a semicolon:

- It usually treats line breaks as semicolons only if it cannot parse the code without the semicolons
- More formally (and with two exceptions described below), JavaScript treats a line break as a semicolon if the next non-space character cannot be interpreted as a continuation of the current statement

Example 1-3: Using Statement Ending Character

1. Open Web Console or use the Example1-3.js file.
2. Type the following code:


F:\Documents\Courses\JavaScript_jQuery\Scripts\Hands-On_Examples\Class1

```
//Example 1-3
let a//--//Declare variable a (no semicolon)
a//--//Variable a (no semicolon)
=//--//Assignment operator (no semicolon)
3//--//Literal value 3 (no semicolon)
alert(a)//--//Output variable a in alert (no semicolon)
//JavaScript interprets this code like this:
//let a; a = 3; alert(a); //Automatic semicolon insertion
```

```
3
```

File Edit Search View Encoding Language Settings Tools Macro R
Example1-3.js
length : 345 lines : 12 Ln : 12 Col : 1 Pos : 346 Windows (CR LF) UTF-8 INS
OK

JavaScript does treat the first line break as a semicolon because it cannot parse the code let a a without a semicolon. The second a could stand alone as the statement a;, but JavaScript does not treat the second line break as a semicolon because it can continue parsing the longer statement a = 3;.

These statement termination rules lead to some surprising cases. This code looks like two separate statements separated with a newline:

Code Example:

```
let y = x + f //Looks like two separate statements
(a+b).toString()
```

But the parentheses on the second line of code can be interpreted as a function invocation of f from the first line, and JavaScript interprets the code like this:

```
let y = x + f(a+b).toString();
```

More likely than not, this is not the interpretation intended by the author of the code. In order to work as two separate statements, an explicit semicolon is required in this case.

In general, if a statement begins with (, [, /, +, or -, there is a chance that it could be interpreted as a continuation of the statement before. Statements beginning with /, +, and -are quite rare in practice, but statements beginning with (and [are not uncommon at all, at least in some styles of JavaScript programming.

Some programmers like to put a defensive semicolon at the beginning of any such statement so that it will continue to work correctly even if the statement before it is modified and a previously terminating semicolon removed.

Code Example:

```
let x = 0 // Semicolon omitted here
;[x,x+1,x+2].forEach(console.log) // Defensive ; keeps this statement separate
```

There are two exceptions to the general rule that JavaScript interprets line breaks as semicolons when it cannot parse the second line as a continuation of the statement on the first line.

The first exception involves the return, break, and continue statements:

- These statements often stand alone, but they are sometimes followed by an identifier or expression
- If a line break appears after any of these words (before any other tokens), JavaScript will always interpret that line break as a semicolon

Code Example:

```
return
true;
//JavaScript assumes you meant:
return; true;
//However, you probably meant:
return true;
```

What this means is that you must not insert a line break between return, break or continue and the expression that follows the keyword. If you do insert a line break, your code is likely to fail in a nonobvious way that is difficult to debug.

The second exception involves the ++ and –operators:

- These operators can be prefix operators that appear before an expression or postfix operators that appear after an expression
- If you want to use either of these operators as postfix operators, they must appear on the same line as the expression they apply to
- Otherwise, the line break will be treated as a semicolon, and the ++ or -- will be parsed as a prefix operator applied to the code that follows

Code Example:

```
x
++
y
//It is parsed as x; ++y;, not as x++; y;
```

3. (Data) Types

Computer programs work by manipulating values, such as the number 2.53 or the text "This is a test." The kinds of values that can be represented and manipulated in a programming language are known as types, and one of the most fundamental characteristics of a programming language is the set of types it supports.

3.1 Overview of Types

When a program needs to retain a value for future use, it assigns the value to (or "stores" the value in) a variable. A variable defines a symbolic name for a value and allows the value to be referred to by name. The way that variables work is another fundamental characteristic of any programming language. This chapter explains types, values, and variables in JavaScript.

JavaScript types can be divided into two main categories:

- primitive types: numbers, strings, Booleans, JavaScript special values (null, undefined)
- object types (unordered collection of named values, arrays, function)

JavaScript's primitive types include numbers, strings of text (known as strings), and Boolean truth values (known as Booleans).

The special JavaScript values null and undefined are primitive values, but they are not numbers, strings, or Booleans. Each value is typically considered to be the sole member of its own special type.

Any JavaScript value that is not a number, a string, a Boolean, or null or undefined is an object. An object (that is, a member of the type object) is a collection of properties where each property has a name and a value (either a primitive value, such as a number or string, or an object).

An ordinary JavaScript object is an unordered collection of named values. The language also defines a special kind of object, known as an array, which represents an ordered collection of numbered values. The JavaScript language includes special syntax for working with arrays, and arrays have some special behavior that distinguishes them from ordinary objects.

JavaScript defines another special kind of object, known as a function. A function is an object that has executable code associated with it. A function may be invoked to run that executable code and return a computed value. Like arrays, functions behave differently from other kinds of objects, and JavaScript defines a special language syntax for working with them.

In addition to the Array and Function classes, core JavaScript defines three other useful classes:

- The Date class defines objects that represent dates
- The RegExp class defines objects that represent regular expressions (a powerful pattern-matching tool)
- And the Error class defines objects that represent syntax and runtime errors that can occur in a JavaScript program

The JavaScript interpreter performs automatic garbage collection for memory management. This means that a program can create objects as needed, and the programmer never needs to worry about destruction or deallocation of those objects. When an object is no longer reachable—when a program no longer has any way to refer to it—the interpreter knows it can never be used again and automatically reclaims the memory it was occupying.

JavaScript is an object-oriented language. Loosely, this means that rather than having globally defined functions to operate on values of various types, the types themselves define methods for working with values. To sort the elements of an array a, for example, we do not pass a to a sort() function. Instead, we invoke the sort() method of a:

```
a.sort(); // The object-oriented version of sort(a)
```

Technically, it is only JavaScript objects that have methods. But numbers, strings, and Boolean values behave as if they had methods. In JavaScript, null and undefined are the only values that methods cannot be invoked on.

As we already know, JavaScript's types can be divided into primitive types and object types. They can be further classified as:

- Types with methods and types without
- Mutable and immutable types

A value of a mutable type can change. Objects and arrays are mutable: a JavaScript program can change the values of object properties and array elements.

Numbers, Booleans, null, and undefined are immutable—it does not even make sense to talk about changing the value of a number, for example. Strings can be thought of as arrays of characters, and you might expect them to be mutable. In JavaScript, however, strings are immutable: you can access the text at any index of a string but JavaScript provides no way to alter the text of an existing string.

JavaScript converts values liberally from one type to another. If a program expects a string, for example, and you give it a number, it will automatically convert the number to a string for you. If you use a non-Boolean value where a Boolean is expected, JavaScript will convert accordingly. JavaScript's liberal value conversion rules affect its definition of equality, and the == equality operator performs type conversions.

JavaScript variables are untyped: you can assign a value of any type to a variable, and you can later assign a value of a different type to the same variable.

3.2 Numbers

Unlike many languages, JavaScript does not make a distinction between integer values and floating-point values. All numbers in JavaScript are represented as floating-point values. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard, and it can represent numbers in the range of:

$\pm 5 \times 10^{-324}$ to $\pm 1.7976931348623157 \times 10^{308}$ (smallest to largest)

The JavaScript number format allows you to exactly represent all integers between −9007199254740992 (-2^{53}) and 9007199254740992 (2^{53}), inclusive. If you use integer values larger than this, you may lose precision in the trailing digits. Note, however, that certain operations in JavaScript (such as array indexing and the bitwise operators) are performed with 32-bit integers.

When a number appears directly in a JavaScript program, it is called a numeric literal. JavaScript supports numeric literals in several formats, as described in the following sections. Note that any numeric literal can be preceded by a minus sign (-) to make the number negative. Technically, however, - is the unary negation operator and is not part of the numeric literal syntax.

Integer Literals

In a JavaScript program, a base-10 integer is written as a sequence of digits.

Code Example:

```
0
3
10000000
```

In addition to base-10 integer literals, JavaScript recognizes hexadecimal (base-16) values. A hexadecimal literal begins with “0x” or “0X”, followed by a string of hexadecimal digits. A hexadecimal digit is one of the digits 0 through 9 or the letters a (or A) through f (or F), which represent values 10 through 15. Here are examples of hexadecimal integer literals.

Code Example:

Hexadecimal (Base 16):	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal (Base 10):	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

0xff // $15*16^1 + 15*16^0$ = 15*16 + 15*1 = 255 (base 10)

Floating-Point Literals

Floating-point literals can have a decimal point; they use the traditional syntax for real numbers. A real value is represented as the integral part of the number, followed by a decimal point and the fractional part of the number.

Floating-point literals may also be represented using exponential notation: a real number followed by the letter e (or E), followed by an optional plus or minus sign, followed by an integer exponent. This notation represents the real number multiplied by 10 to the power of the exponent.

Syntax

```
[digits][.digits][(E|e)[(+|-)]digits]
```

Code Example:

```
3.14
2345.789
.333333333333333333
6.02e23 // 6.02 × 10^23
1.4738223E-32 // 1.4738223 × 10^−32
```

Arithmetic Operations

JavaScript programs work with numbers using the arithmetic operators that the language provides:

- + addition
- - subtraction
- * multiplication
- / division
- % modulo (remainder after division)

In addition to these basic arithmetic operators, JavaScript supports more complex mathematical operations through a set of functions and constants defined as properties of the Math object.

Demo 1-2: Using Global Math Object

Execute the demo example by using Demo Example application.

Arithmetic in JavaScript does not raise errors in cases of overflow, underflow, or division by zero. When the result of a numeric operation is larger than the largest representable number (overflow), the result is a special infinity value, which JavaScript prints as Infinity.

Similarly, when a negative value becomes larger than the largest representable negative number, the result is negative infinity, printed as -Infinity. The infinite values behave as you would expect: adding, subtracting, multiplying, or dividing them by anything results in an infinite value (possibly with the sign reversed).

Underflow occurs when the result of a numeric operation is closer to zero than the smallest representable number. In this case, JavaScript returns 0. If underflow occurs from a negative number, JavaScript returns a special value known as "negative zero." This value is almost completely indistinguishable from regular zero and JavaScript programmers rarely need to detect it.

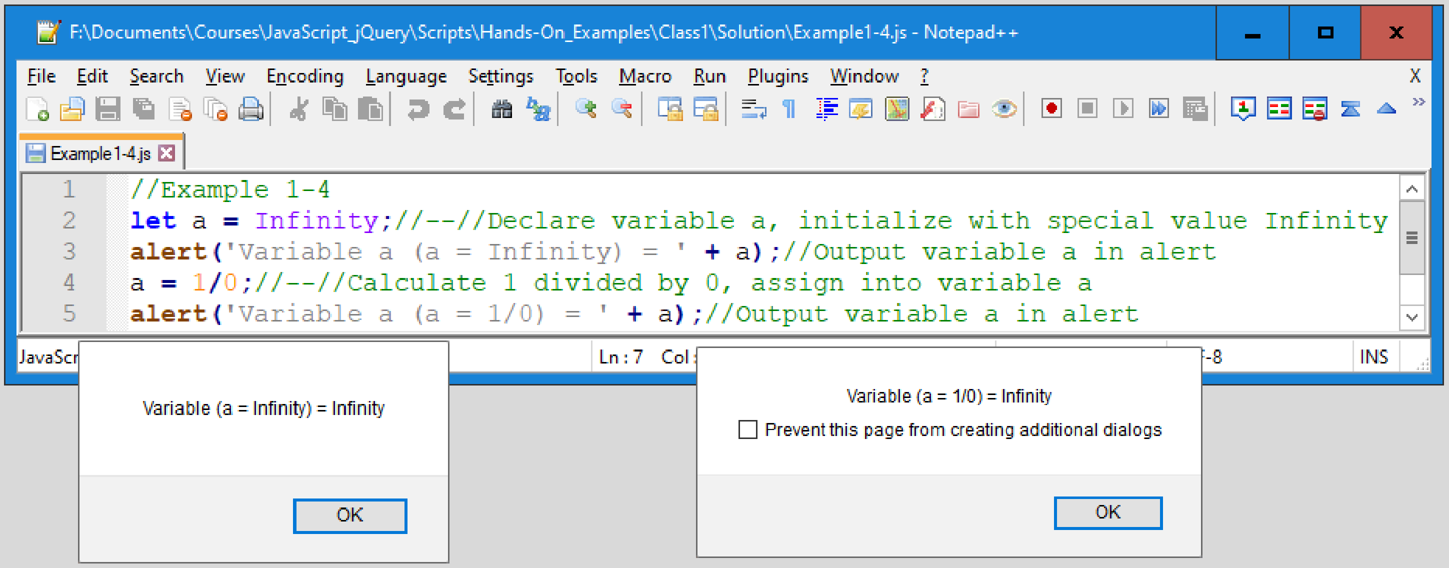
Division by zero is not an error in JavaScript: it simply returns infinity or negative infinity. There is one exception, however: zero divided by zero does not have a well-defined value, and the result of this operation is the special not-a-number value, printed as NaN.

NaN also arises if you attempt to divide infinity by infinity, or take the square root of a negative number or use arithmetic operators with non-numeric operands that cannot be converted to numbers.

JavaScript predefines global variables Infinity and NaN to hold the positive infinity and not-a-number value. These global variables are read-only.

Example 1-4: Using Infinity

1. Open Web Console or use the Example1-4.js file.
2. Type the following code:


F:\Documents\Courses\JavaScript_jQuery\Scripts\Hands-On_Examples\Class1\Solution\Example1-4.js - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Example1-4.js
Ln : 7 Col
INS

```
//Example 1-4
let a = Infinity;//--//Declare variable a, initialize with special value Infinity
alert('Variable a (a = Infinity) = ' + a);//Output variable a in alert
a = 1/0;//--//Calculate 1 divided by 0, assign into variable a
alert('Variable a (a = 1/0) = ' + a);//Output variable a in alert
```

```
Variable (a = Infinity) = Infinity
```

```
Variable (a = 1/0) = Infinity
```

Prevent this page from creating additional dialogs
OK
OK

The demo example below show some other examples that result in Infinity, -Infinity, NaN, and -0.

Demo 1-3: Using Infinity, NaN, and Zero

Execute the demo example by using Demo Example application.

The not-a-number value has one unusual feature in JavaScript: it does not compare equal to any other value, including itself. This means that you cannot write x == NaN to determine whether the value of a variable x is NaN. Instead, you should write x != x.

That expression will be true if, and only if, x is NaN. The function is NaN() is similar. It returns true if its argument is NaN, or if that argument is a non-numeric value such as a string or an object. The related function isFinite() returns true if its argument is a number other than NaN, Infinity, or -Infinity.

The negative zero value is also somewhat unusual. It compares equal (even using JavaScript's strict equality test) to positive zero, which means that the two values are almost indistinguishable, except when used as a divisor.

Code Example:

```
let zero = 0; // Regular zero
let negz = -0; // Negative zero
zero === negz // => true: zero and negative zero are equal
1/zero === 1/negz // => false: infinity and -infinity are not equal
```

Binary vs. Decimal Math

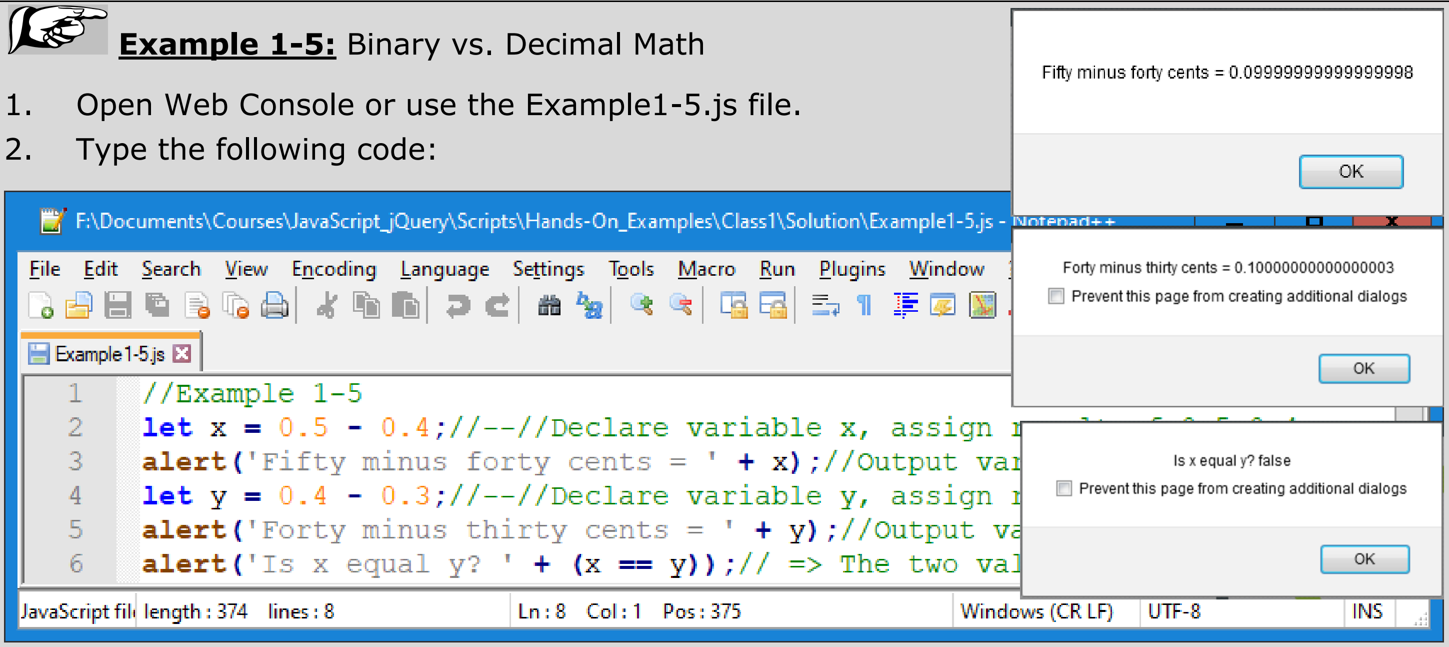
There are infinitely many real numbers, but only a finite number of them can be represented exactly by the JavaScript floating-point format. This means that when you are working with real numbers in JavaScript, the representation will often be an approximation of the actual number.

The IEEE-754 floating-point representation used by JavaScript (and just about every other modern programming language) is a binary representation, which can exactly represent fractions like 1/2, 1/8, and 1/1024. Unfortunately, the fractions we use most commonly (especially when performing financial calculations) are decimal fractions 1/10, 1/100, and so on. Binary floating-point representations cannot exactly represent numbers as simple as 0.1.

JavaScript numbers have plenty of precision and can approximate 0.1 very closely. But the fact that this number cannot be represented exactly can lead to problems.

Example 1-5: Binary vs. Decimal Math

1. Open Web Console or use the Example1-5.js file.
2. Type the following code:


F:\Documents\Courses\JavaScript_jQuery\Scripts\Hands-On_Examples\Class1\Solution\Example1-5.js - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window
Example1-5.js

```
//Example 1-5
let x = 0.5 - 0.4;//--//Declare variable x, assign r
alert('Fifty minus forty cents = ' + x);//Output var
let y = 0.4 - 0.3;//--//Declare variable y, assign r
alert('Forty minus thirty cents = ' + y);//Output va
alert('Is x equal y? ' + (x == y));// => The two val
```

```
JavaScript fil length : 374   lines : 8   Ln : 8   Col : 1   Pos : 375   Windows (CR LF)   UTF-8   INS
```

```
Fifty minus forty cents = 0.09999999999999998
```

```
Forty minus thirty cents = 0.10000000000000003
```

```
Is x equal y? false
```

Because of rounding error, the difference between the approximations of 0.5 and 0.4 is not exactly the same as the difference between the approximations of 0.4 and 0.3. It is important to understand that this problem is not specific to JavaScript: it affects any programming language that uses binary floating-point numbers. Also, note that the values x and y in the code above are very close to each other and to the correct value.

The computed values are adequate for almost any purpose: the problem arises when we attempt to compare values for equality.

A future version of JavaScript may support a decimal numeric type that avoids these rounding issues. Until then you might want to perform critical financial calculations using scaled integers. For example, you might manipulate monetary values as integer cents rather than fractional dollars.

3.3 Dates and Times

Core JavaScript includes a Date() constructor for creating objects that represent dates and times. These Date objects have methods that provide an API for simple date computations. Date objects are not a fundamental type like numbers are. This section presents a quick tutorial on working with dates.

The Date object can be initialized in the following four ways:

- new Date()
- new Date(milliseconds)
- new Date(datestring)
- new Date(year, month [, day, hours, minutes, seconds, ms])

With no arguments, the Date() constructor creates a Date object set to the current date and time. When one numeric argument is passed, it is taken as the internal numeric representation of the date in milliseconds, as returned by the getTime() method. The date representation in milliseconds means the number of milliseconds passed since the reference date of 01 January 1970 00:00:00 UTC.

When one string argument is passed, it is a string representation of a date, in the format accepted by the Date.parse() method. Otherwise, the constructor is passed between two and seven numeric arguments that specify the individual fields of the date and time.

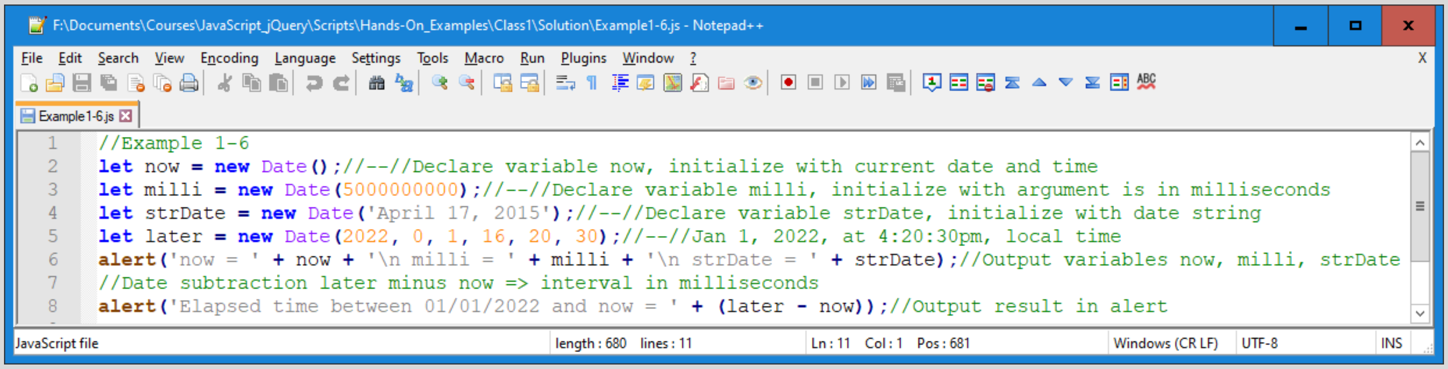
All but the first two arguments—the year and month fields—are optional. Note that these date and time fields are specified using local time, not Coordinated Universal Time (UTC) (which is similar to Greenwich Mean Time [GMT]).

Date() may also be called as a function, without the new operator. When invoked in this way, Date() ignores any arguments passed to it and returns a string representation of the current date and time, it is not a Date object.

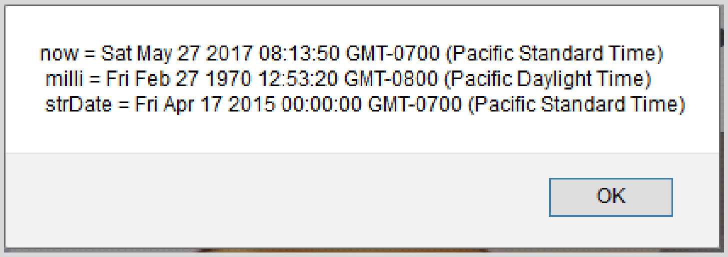
STOP **Warning:** JavaScript uses a convention where month numbers start at zero (so December is 11), yet day numbers start at one. This is confusing, so be careful!

Example 1-6: Date calculations

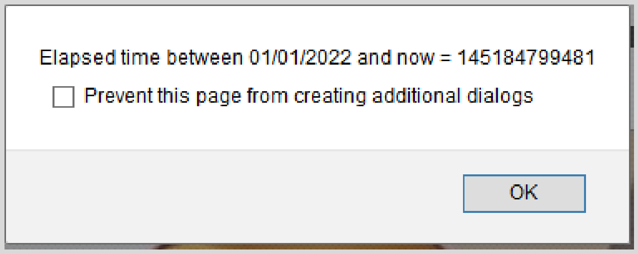
1. Open Web Console or use the Example1-6.js file.
2. Type the following code:


F:\Documents\Courses\JavaScript_jQuery\Scripts\Hands-On_Examples\Class1\Solution\Example1-6.js - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Example1-6.js
JavaScript file
length : 680 lines : 11
Ln : 11 Col : 1 Pos : 681
Windows (CR LF) UTF-8 INS

```
//Example 1-6
let now = new Date();//--//Declare variable now, initialize with current date and time
let milli = new Date(5000000000);//--//Declare variable milli, initialize with argument is in milliseconds
let strDate = new Date('April 17, 2015');//--//Declare variable strDate, initialize with date string
let later = new Date(2022, 0, 1, 16, 20, 30);//--//Jan 1, 2022, at 4:20:30pm, local time
alert('now = ' + now + '\n milli = ' + milli + '\n strDate = ' + strDate);//Output variables now, milli, strDate
//Date subtraction later minus now => interval in milliseconds
alert('Elapsed time between 01/01/2022 and now = ' + (later - now));//Output result in alert
```



```
now = Sat May 27 2017 08:13:50 GMT-0700 (Pacific Standard Time)
 milli = Fri Feb 27 1970 12:53:20 GMT-0800 (Pacific Daylight Time)
 strDate = Fri Apr 17 2015 00:00:00 GMT-0700 (Pacific Standard Time)
```



```
Elapsed time between 01/01/2022 and now = 145184799481
```

The Date object has no properties that can be read and written directly; instead, all access to date and time values is done through methods. Most methods of the Date object come in two forms:

- Local time
- Universal (UTC or GMT) time

If a method has “UTC” in its name, it operates using universal time.

Date methods may be invoked only on Date objects, and they throw a TypeError exception if you attempt to invoke them on any other type of object.

There are many methods available for the Date object, some of the most important ones are shown in the next example.

Demo 1-4: Using Date Object Methods

Execute the demo example by using Demo Example application.

3.4 Text

A string is an immutable ordered sequence of 16-bit values, each of which typically represents a Unicode character—strings are JavaScript's type for representing text. The length of a string is the number of 16-bit values it contains. JavaScript's strings (and its arrays) use zero-based indexing: the first 16-bit value is at position 0, the second at position 1 and so on. The empty string is the string of length 0. JavaScript does not have a special type that represents a single element of a string. To represent a single 16-bit value, use a string that has a length of 1.

Note:. An immutable string means that the characters within it may not be changed and that any operations on strings actually create new strings.

String Literals

To include a string literally in a JavaScript program, simply enclose the characters of the string within a matched pair of single or double quotes ('or "). Double-quote characters may be contained within strings delimited by single-quote characters, and single-quote characters may be contained within strings delimited by double quotes.

Here are examples of string literals:

- "" // The empty string: it has zero characters
- 'testing'
- "3.14"
- 'name="myform"'
- "Wouldn't you prefer O'Reilly's book?"
- "This string\nhas two lines"
- "π is the ratio of a circle's circumference to its diameter"

Note: When you use single quotes to delimit your strings, you must be careful with English contractions and possessives, such as can't and O'Reilly's. Since the apostrophe is the same as the single-quote character, you must use the backslash character (\) to "escape" any apostrophes that appear in single-quoted strings.

In client-side JavaScript programming, JavaScript code may contain strings of HTML code, and HTML code may contain strings of JavaScript code. Like JavaScript, HTML uses either single or double quotes to delimit its strings. Thus, when combining JavaScript and HTML, it is a good idea to use one style of quotes for JavaScript and the other style for HTML. In the following example, the string "Thank you" is single quoted within a JavaScript expression, which is then double-quoted within an HTML event-handler attribute

Code Example:

```
<button onclick="alert('Thank you')">Click Me</button>
```

Escaping in String Literals

The backslash character (\) has a special purpose in JavaScript strings. Combined with the character that follows it, it represents a character that is not otherwise representable within the string. For example, \n is an escape sequence that represents a newline character.

Another example, mentioned previously, is the \' escape, which represents the single quote (or apostrophe) character. This escape sequence is useful when you need to include an apostrophe in a string literal that is contained within single quotes. You can see why these are called escape sequences: the backslash allows you to escape from the usual interpretation of the single-quote character. Instead of using it to mark the end of the string, you use it as an apostrophe:

'You\'re right, it can\'t be a quote'

Table 1 lists the JavaScript escape sequences and the characters they represent. Two escape sequences are generic and can be used to represent any character by specifying its Latin-1 or Unicode character code as a hexadecimal number. For example, the sequence \xA9 represents the copyright symbol, which has the Latin-1 encoding given by the hexadecimal number A9. Similarly, the \u escape represents an arbitrary Unicode character specified by four hexadecimal digits; \u03c0 represents the character π, for example.

Sequence	**Character represented**
\0	The NUL character (\u0000)
\b	Backspace (\u0008)
\t	Horizontal tab (Character tabulation) (\u0009)
\n	Newline (\u000A)
\v	Vertical tab (Line tabulation) (\u000B)
\f	Form feed (\u000C)
\r	Carriage return (\u000D)
\"	Double quote (\u0022)
\'	Apostrophe or single quote (\u0027)
\\\\	Backslash (\u005C)
\x XX	The Latin-1 character specified by the two hexadecimal digits XX
\u XXXX	The Unicode character specified by the four hexadecimal digits XXXX

Table 1: Escape Sequences

Warning: If the \ character precedes any character other than those shown in Table 1, the backslash is simply ignored (although future versions of the language may, of course, define new escape sequences).

String Processing

One of the built-in features of JavaScript is the ability to concatenate strings. If you use the + operator with numbers, it adds them. But if you use this operator on strings, it joins them by appending the second to the first.

Code Example:

```
msg = "Hello, " + "world"; // Produces the string "Hello, world"
greeting = "Welcome to my blog," + " " + name;
```

To determine the length of a string—the number of 16-bit values it contains—use the length property of the string. Determine the length of a string s like this.

Code Example:

```
s.length
```

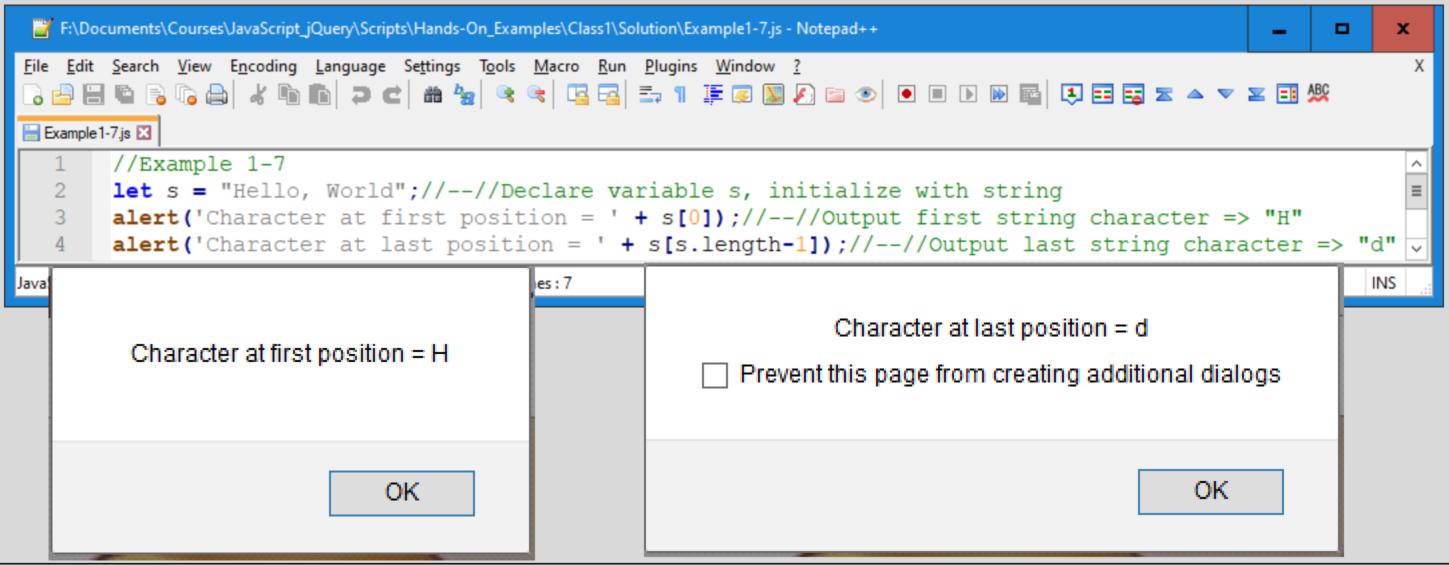
In addition to this length property, there are a number of methods you can invoke on strings, remember, a string is not an object but behaves like one as it does have built-in methods.

Remember that strings are immutable in JavaScript. Methods like replace() and toUpperCase() return new strings: they do not modify the string on which they are invoked.

In ECMAScript 5, strings can be treated like read-only arrays, and you can access individual characters (16-bit values) from a string using square brackets instead of the charAt() method.

Example 1-7: String processing

1. Open Web Console or use the Example1-7.js file.
2. Type the following code:



F:\Documents\Courses\JavaScript_jQuery\Scripts\Hands-On_Examples\Class1\Solution\Example1-7.js - Notepad++

```
//Example 1-7
let s = "Hello, World";//--//Declare variable s, initialize with string
alert('Character at first position = ' + s[0]);//--//Output first string character => "H"
alert('Character at last position = ' + s[s.length-1]);//--//Output last string character => "d"
```

Character at first position = H

Character at last position = d

Prevent this page from creating additional dialogs

Mozilla-based web browsers such as Firefox have allowed strings to be indexed in this way for a long time. Most modern browsers (with the notable exception of IE) followed Mozilla's lead even before this feature was standardized in ECMAScript 5.

Demo 1-5: Using String Methods

Execute the demo example by using Demo Example application.