|  |
| --- |
| [[https://myetudes.org/etudes-melete-tool/images/printer.png](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385313) Send to Printer](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385313) | [Close Window](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385313) |
|  | |
| 2. JavaScript Basics  2.1. Using the Firebug Console  *Copyrighted Material - subject to fair use exception*  To get started with JavaScript, we'll use the Firebug Console.  The Console provides an interactive JavaScript command line to execute little bits of code.   **It is accessible from the Firefox browser.  The first thing that we need to do is to open Firefox.**  From here on, instead of passively reading along, you should start Firebug and try to replicate the actions.  The following discussion is also illustrated in the screencast below:    There are two main ways to open Firebug: you can click on the Firebug icon on the Firefox status bar or press the F12 key on your keyboard.  You can then select the console tab from the Firebug menu.  If the console panel is disabled, you can click on Enable to enable it.  You can type JavaScript code directly at the '>>>' prompt at the bottom of the Firebug window. Whenever you enter a code fragment and press Enter, it will be executed. For instance, typing:  3 + 21  and pressing Enter will cause the following to be displayed on the top part of the window:  >>> 3 + 21  24    Note that throughout this course, the interpreter output will be shown in blue.  Let’s try a few more operators:  >>> 4 \* 5 //  this is a comment.  It will be ignored by the interpreter.  20  >>> 10 – 4  /\* this is also a comment \*/  6  >>> 8 / 2  // this is the division operator  4  >>> 9 / 2  4.5  >>> 10 % 3 // this is the modulo operator also known as the remainder operator  1  When you’re done, you can close Firebug by using the off button in the upper right corner of the Firebug console.  2.2. Using Scratchpad  *Copyrighted Material - subject to fair use exception*  Scratchpad is another tool available from Firefox that we'll use in this course.  It is basically a text editor that knows how to run JavaScript.  It allows us to save JavaScript code in a file, come back to edit it, execute it and so on.  You can access Scratchpad from the Firefox Developer menu as shown below:    The Scratchpad window will open:  Let's write our first program in Scratchpad.  We'll use**console.log()** to display output on the console.  We'll save our program in the file helloworld.js. **js is the extension we use for JavaScript programs.**    Before we run our program, we need to make sure that the Firebug console is open: **when we use console.log() to display results, the output will appear in the Firebug console, not in the Scratchpad file.**  Remember that you can open Firebug by clicking on the Firebug icon in Firefox or by pressing the F12 key on your keyboard.  Now we are ready to run our JavaScript program by pressing on Run in the Scratchpad window.  For complete instructions on how to use Scratchpad, go to:  <https://developer.mozilla.org/en-US/docs/Tools/Scratchpad>    2.3. Comments  *Copyrighted Material - subject to fair use exception*  By now you may have noticed that JavaScript supports two forms of comments:  **Line-ending comments starting with //** such as in:  >>> 4 \* 5   // this is a comment.  It will be ignored by the interpreter.  These comments go on until the end of the line.  **Block comments formed with /\* \*/**  such as in:  >>> 10 – 4  **/\***this is also a comment **\*/**   ' /\*' starts a comment that goes on until a '\*/' is found.  Block comments may span multiple lines, but they may not be nested.  You will often see JavaScript files, classes, methods and properties commented as follows:  /\*\*   \*  This is a multiple line comment.   \* This style is used to conform to the JSDoc markup language.   \* JSDoc is used to generate documentation describing the application programming interface.   \*/  The only delimiters required by JavaScript here are the opening /\* and the closing \*/.  2.4. Operator Precedence  *Copyrighted Material - subject to fair use exception*  Now let’s try the following in the Firebug console to explore the order of operations:  >>> 1 +  2 \* 3  7  Here 2 is multiplied by 3 first and the result 6 is added to 1.  >>> 4 \* 2 - 5  3  Here 4 is multiplied by 2 then 5 is subtracted from the result.  >>> 100 \* 4 - 4 + 50 / 2  421  Here 100 is multiplied by 4 first and 50 is divided by 2 and we end up with: 400 - 4 + 25 which gives us 421.  Multiplication has a higher precedence than addition. **Division and multiplication always come before subtraction and addition. When multiple operators with the same precedence appear next to each other they are applied left-to-right.**  When in doubt, just add parentheses.  Expressions inside the parentheses are always evaluated first.  >>> (1+2)\*3  9  2.5. JavaScript Types  *Copyrighted Material - subject to fair use exception*  The primitive types of JavaScript include numbers (such as 4 and -6.8), strings ('Hello' and "Hi") and booleans (true and false).  The special values **null** and **undefined**are also primitive values.  They are used to refer to some missing data. For instance an uninitialized variable is undefined.   The value null may be used to represent a non-existent reference.  The difference between the two will become clearer later in this course.  Everything else is an object (member of type object).  The **typeof** operator is very useful.  It returns a **string value** naming the type of the operand we give it.  Let's try the following in the Firebug console:  >>> typeof 4  "number"  >>> typeof -6.8  "number"  >>> typeof "Hello"  "string"  >>> typeof true  "boolean"  >>> typeof "true"  "string"  Note that the value true is a boolean but "true" (enclosed in quotes) is a string.    >>> typeof undefined  "undefined"  >>> typeof null  "object"  Note that even though null is a primitive value, typeof null is "object".  This is considered a mistake in the original design.  2.6. Numbers  *Copyrighted Material - subject to fair use exception*  There is no distinction in JavaScript between integer and floating point numbers.  All numbers are represented as floating point numbers.  Let’s try a few more things with numbers in the Firebug console:  >>> 1.2e3 + 1000  2200  1.2e3 is the exponential notation (also known as the scientific notation) for 1200.  It is the same as 1.2 x 103.  >>> 4 / 0  Infinity  In JavaScript, dividing by 0 does NOT result in an error.  >>> -5.39 / 0  -Infinity  The value Infinity represents all values greater than 1.79769313486231570e+308.  >>> typeof  (4/0)  "number"  >>> typeof  (5.39/0)  "number"  >>> typeof Infinity  "number"  >>> 0 / 0  NaN  Dividing 0 by 0 results in NaN, which is a special value. It stands for 'not a number'.  However NaN itself is of type number. Contradictory?  Maybe.  >>> typeof (0/0)  "number"  >>> typeof NaN  "number"  NaN is also not equal to anything including itself.  2.7. Booleans, Comparison & Logical Operators  *Copyrighted Material - subject to fair use exception*  Booleans can only have two values: true or false.  **Comparison operators** return Boolean values:  >>> 5 > 3  true  >>> 2 < =1  false  >>> 2 == 1 + 1  // == is the (lenient) equality comparison operator: is 2 equal to 1 + 1?  true  >>> 2 != 1 + 1  // != is the not equal comparison operator  false  We also have **the strict equality operators.**  We'll see how they differ from ==  and != shortly.  >>> 2 === 1 + 1  // === means equal value and equal type  true  >>> 2 !==1 + 1  // different value or different type  false  **Comparison vs Assignment:**  **Note that a single equal sign = is  NOT a comparison operator.**  The single equal sign indicates an assignment.  It is used to assign a value to a variable.  We'll talk more about variables in an upcoming section.  total = 200; // assign the value 200 to the variable total  total === 200 // this is a boolean: is total equal to 200?    **Falsy and Truthy:**  The following values are all considered to be false: 0, -0, null, ""(empty string), undefined, NaN and of course false.  Any other value is considered to be true (even the string "false").    **Logical operators:**  JavaScript supports three **logical operators**: && (and), || (or) , and !(not).  >The ! (not)  operator returns true if the value given to it is false and false if the value given to it is true.  **Special behavior of and and or operators in JavaScript:**  We are generally used to the following behavior of and and or logical operators:  The and operator is supposed to return true only if both of the values given to it are true.  The or operator is supposed to return true if either of the values given to it is true.  There is a bit more to this in JavaScript: the and and or operators do not require that their operands be boolean values.  The && (and) operator in JavaScript produces **the value of the first operand if the first operand is falsy; it does not even evaluate the second operand.  Otherwise it produces the value of the second operand.**  Similarly the || (or) operator in JavaScript **produces the value of the first operand if the first operand is truthy; it does not even evaluate the second operand.    Otherwise it produces the value of the second operand.**  The above distinction does not change the behavior of the operators when dealing with pure Boolean values or expressions:  >>> 3 > 2 && 8 < 100  true  >>> 3 > 2 && 8 < 5  false  >>> 3 != 2 || 8 >  10  true  However, when dealing with  non-Booleans, that distinction is important:  When the first operand of && is falsy, the second operand is not even evaluated:  **we call this a short-circuit evaluation.**  >>> null && 7 > 2  // null is falsy so the first operand is returned  null  >>> "hello" && 5  //  "hello" is truthy so the second operand is returned  5  We'll see later how to **use && to avoid null references**.  Similarly when the first operand of || is truthy, the second operand is not even evaluated.  >>> 700 || 2 < 0  // 700 is truthy so the first operand is returned  700  >>> 0 || 500 // 0 is falsy so the second operand is returned  500  **This behavior is sometimes used to supply default values**as shown below:  name = name || 'Friend'  When the variable name is undefined, it evaluates to falsy so the default value 'Friend' will be used.    **Operator Precedence:**  Operators with higher precedence are performed before those with lower precedence.  Of the operators we just covered, ! (not) has the highest precedence, then the comparison operators (<, ==, ===, etc.), then the logical and (&&) and finally the logical or(||).  Again when in doubt, use parentheses.    2.8. Strings  *Copyrighted Material - subject to fair use exception*  Strings are sequences of characters (16-bit Unicode characters, to be precise).  **Strings may be enclosed in single quotes as in 'Hi' or in double quotes as in "Hello".**  Single quotes allow us to embed double quotes:  '"I love coffee", she said'  Double quotes allow us to embed apostrophes (single quotes):  "Isn’t this great?"  An**empty string** is denoted by a pair of single or double quotes with nothing in between them: '' or "".  Note that when we call console.log with an empty string, the interpreter outputs *(an empty string)*because otherwise we simply can't see any output.  >>> console.log('');  *(an empty string)*  The \ (backslash) is the escape character.  It can be used to insert new lines, apostrophes, quotes, and other special characters into a string.  Whenever a backslash is found inside a string, it indicates that the character after it has a special meaning.  To see how this works, let’s use the function console.log() to display output on the console.  From the Firebug console, try the following:  >>> console.log('Hello World');  Hello World  We can embed single quotes inside a single quoted string by using the escape character \'.  The quote, now preceded by a backslash, will not end the string, but be part of it.  >>> console.log('Isn**\'**t this great?');  Isn't this great?  When an n character occurs after a backslash, it is interpreted as a new line.   >>> console.log('first line**\n**second line');  first line  second line  Similarly \t is the control character for a tab.  >>> console.log('Foothill\tJavaScript for Programmers');  Foothill                 JavaScript for Programmers  Now what if we want to print an actual backslash?  We’ll need to escape it with another backslash.  Only one will be printed.  >>> console.log('**\\**');  \  **Multi-line Strings:**  In ECMAScript 3, strings must be written on a single line.  In ECMAScript 5, we can break a string across multiple lines by ending each line but the last with a backslash (\). The backslashes and the new line characters that follow are NOT part of the string.  To see how that works, let’s first make sure the Firebug console is open.  Then we open a Scratchpad window and type the following:  console.log('Here \  is \  a one line \  string \  written \  on 5 lines');      We then press on the Run tab in the Scratchpad window, and the following will appear in the Firebug console:  Here is a one line string written on 5 lines    **String length:**  Strings have a length property.  >>> 'Foothill'.length  8  An empty string has a length of 0.  >>> ''.length  0  **Character Access:**  There are two ways to access an individual character in a string.  Both ways use**zero-based indexing**: the first character is at position 0, the second at position 1 and so on.  charAt:  >>> 'Foothill'.charAt(0)  "F"  >>> 'Foothill'.charAt(4)  "h"  >>> 'Foothill'.charAt(7)  "l"  >>> 'Foothill'.charAt(8)  ""  The square bracket notation**:** this method was introduced in ECMAScript 5:  >>> 'Foothill'[0]  "F"  >>> 'Foothill'[5]  "i"  **>>> 'Foothill'[8]**  **undefined**  **>>> 'Foothill'[-1]**  **undefined**    **Note that strings are immutable. Changing individual characters inside a string is not possible.**  **Concatenating Strings:**  You can make a new string by concatenating other strings together with the + operator.  >>>'Hello ' + 'World'+'!!!'  "Hello World!!!"  Note that this just puts the strings together.  It does not add any space in between.  If you need a space character, you need to include it.  **Other useful string methods:**  >>> 'Foothill'.**toUpperCase**()  "FOOTHILL"  >>> 'Foothill'.**toLowerCase**()  "foothill"  Remember that strings are immutable in JavaScript. **Methods like toUpperCase() and toLowerCase() return new strings: they do not modify the string on which they are invoked.**  To look for a substring in a given string, we can use indexOf:  >>> 'Foothill'.**indexOf**('hi')   // **returns the index of the first ‘hi’**  4  >>> 'Foothill'.indexOf('o', 2)  // **returns the index of the first ‘o’ starting at position 2**  2  >>> 'Foothill'.**indexOf**('hello')   // **returns -1 if the specified text is not found**  -1  >>> 'Foothill'.**lastIndexOf**('o') // returns the index of the last ‘o’  2  We can also use search to look for a substring in a given string.  In addition to a simple substring value (such as 'hi'), search allows us to specify some more complicated patterns (regular expressions).  We will not cover regular expressions in this course.  >>> 'Foothill'.**search**('hi')  // returns the index of the match if the search is successful  4  >>> 'Foothill'.**search**('Hi')  //  returns -1 if the search fails  -1  >>> 'Welcome to CS 21A!'.**replace**('21A', '22A') // replace 21A by 22A  "Welcome to CS 22A!"  The **substring()**method returns a subset of a string starting at one index and up to but not including the other.  >>>  'Foothill'.substring(4,6)  "hi"  If the second index is omitted, it defaults to the string length.  >>> 'Foothill'.substring(4)  "hill"  The substr() method is NOT the same as substring().  The substr() method  returns the characters in a string beginning at the specified index through the **specified number of characters.**  Another difference is that substr() lets you specify a negative index, to start from the end of the string whereas substring does not.  >>> 'Foothill'.substr(4,2)  // returns 2 characters, starting at position 4  "hi"  >>> 'Foothill'.substr(4,4)  // returns 4 characters, starting at position 4  "hill"  The following will return 3 characters, starting at position -3 which is the 3rd character from the end.  >>> 'Foothill'.substr(-3,3)  "ill"  The **split()**method breaks a string into an array of strings based on the separator specified.  >>>  'JavaScript for Programmers'.split(' ') // split on space character  ["JavaScript", "for", "Programmers"]  If the separator is omitted, the array returned contains one element consisting of the entire string.  This is different than Python where the separator defaults to the space character.  >>> 'JavaScript for Programmers'.split()  ["JavaScript for Programmers"]  If the separator is an empty string, you get an array of characters.  >>>  'JavaScript'.split('') //split on empty string  ["J", "a", "v", "a", "S", "c", "r", "i", "p", "t"]  If you have been following along in the Firebug console, you may have noticed by now that the command line provides automatic code completion. As you enter an expression, a popup shows a list of all available methods or properties. Using Tab or Enter you can autocomplete the selected suggestion. You can cycle through the different suggestions via Up and Down.  This is a useful feature that may save us from common misspelling errors.    **Comparing strings:**  Two strings containing exactly the same characters in the same order are considered to be the same string.  >>> 'Foot'+ 'hill' === 'Foothill'  true  You can also use the comparison operators <,  >, <= and >= on strings.  The comparison is similar to the alphabetical sort (the way names are sorted in a phone book or words are sorted in a dictionary).  The characters in the strings are compared from left to right.  Upper case letters come before lower case ones.   The unicode value of the characters in the string is actually used to determine their order.  >>> 'Alice' < 'Bob'  true  >>> 'Charlie' <= 'Bob'  false  >>> 'Charlie' >= 'Bob'  true  >>> 'Charlie' < 'Charles'  false  >>> 'alice' < 'Alice'  false  >>> 'Z' < 'a'  true  >>> '?' < '!'  false  2.9. Strict Mode  *Copyrighted Material - subject to fair use exception*  Strict mode is a new feature in ECMAScript 5 that lets us impose stricter rules on a program, or a function. This strict context prevents certain previously allowed but unsafe actions from being taken.   It provides stronger error checking and throws exceptions where errors used to pass silently.  It also disables some ‘features’ of the language that are confusing or deficient.  To enable strict mode, we just add the following string at the top of the program or inside a function:  **'use strict';**  When covering various topics in this course, we will highlight the differences between strict mode and non-strict mode as they arise.  In our assignments, we'll always enable strict mode:  our code will be more robust as a result.    2.10. Variables  *Copyrighted Material - subject to fair use exception*  Variables are used to store information that can change over time.  A variable has a name and that name is used to access the information stored in that variable.  In JavaScript, **variable names must begin with a letter, a $ character or an underscore.**  Subsequent characters can be letters, digits, underscores, or $.  grade, grade1, $grade and \_grade are all valid variable names.  However 1grade and #grade are NOT.  **Variable names are case sensitive**.  grade and Grade are not the same.  The convention is to start all variable names in JavaScript with lower case letters.  A variable name cannot be a reserved word.  Reserved words are words that have a special meaning in JavaScript such as true, false and typeof.  **Before you use a variable in a JavaScript program, you should declare it.**  You declare JavaScript variables with the **var** keyword:  **var** grade;   // semicolons are used to separate JavaScript statements  After the declaration, the variable has no value.  If you try to access it, it has the value undefined.  console.log(**grade**);  undefined  Note that we asked console.log to output the variable grade and we got undefined.  We can also write:  console.log("The value of the variable grade is: ", **grade**);  The value of the variable grade is: undefined  **To assign a value to the variable, we use the = sign:**  grade = 100;  We can also combine the declaration and the assignment in one step:  var grade = 100;  Now when we use console.log to output the variable grade we'll get 100.  console.log("The value of the variable grade is: ", **grade**);  The value of the variable grade is: 100  We can use the variable grade in various expressions and it will be replaced by its current value:  console.log( **grade - 5**);  95  It's a good programming practice to declare all the variables you will need, in one place, at the beginning of your code.  You can declare many variables in one statement.  Just start the statement with **var** and separate the variables by comma:  var homework = 100, midterm = 90, final = 95;  The declaration can also span multiple lines:  var homework = 100,  midterm = 90,  final = 95;  **If we re-declare a JavaScript variable, it will not lose its value**:  var average = 95;  // first declaration and assignment  var average;  // second declaration  console.log(average);  95  The value of average is still 95, not undefined.  **JavaScript variables do not have a declared type:** we can assign a value of any type to a variable, and we can later assign a value of a different type to the same variable.  We can check the type of a variable at any point with typeof.  The type reflects the type of the last assigned value.  >>> var grade = 100;  >>> typeof grade  "number"  >>> grade = 'A';  >>> typeof grade  "string"  >>> grade = true;  >>> typeof grade  "boolean"  If we attempt to read the value of an**undeclared variable**, we get an error.  console.log(friend);  Exception: friend is not defined  However you may be able to assign a value to an undeclared variable **in non-strict mode.**  This is a source of many bugs and **you should always declare your variables whether running in strict mode or not.**  **Variables, Properties and Methods:**  Note that when a variable is assigned a value of a given type, the methods and properties defined for that type become available for that variable.  >>> var name = 'Alice'; // name is a variable that is assigned the value 'Alice'  >>> name.length;   // we can use the length property defined on strings  5  Note that **the variable is substituted with its value** so name.length is actually the length of the string 'Alice' which is 5, NOT the length of the string 'name'.  >>> name.toLowerCase()   // we can use the toLowerCase() method defined on strings.  "alice"  We noted in a previous section that strings are immutable in JavaScript. Methods like toUpperCase() and toLowerCase() return new strings: they do not modify the string on which they are invoked.  We can verify that the value stored in the variable name has not been modified.  >>> name  "Alice"  However that does not mean that we can never change the value stored in the string variable name.  We can simply **assign a new value to it**:  >>> name = 'BOB';  >>> name  "BOB"  To change the value stored in name to lower case, we can write:  >>> name = name.toLowerCase();  // assign a new value to name  >>> name  "bob"        2.11. Semicolons, White Space and Line Breaks  *Copyrighted Material - subject to fair use exception*  Semicolons are used to separate JavaScript statements.  We can use Scratchpad to try the following examples.  **We usually add a semicolon at the end of each executable statement.**  var grade, name;  grade = 100;  name = 'Alice';  Using semicolons also makes it possible to write many statements on one line.  grade = 100;  name = 'Alice';  **However it is best to put at most one statement on a line.**  **Optional Semicolons?**  In the previous sections, we have seen examples with and without semicolons: that’s because ending statements with semicolon is optional in JavaScript.  However JavaScript has some complicated rules as to when the semicolons can be safely omitted.  **It usually treats line breaks as semicolons only if it can’t parse the code without the semicolons.** But there are exceptions to this rule.  One of these exceptions is that JavaScript will always interpret a line break after a ‘return’ as a semicolon.  **So to be safe, it is best to always use semicolons at the end of your statements.**  **White Space:**  JavaScript ignores extra spaces. You can add space characters or indentation to your code to make it more readable. The following lines are equivalent:  var grade=100;  var       grade  =  100;  **Multiline Statements:**  We have seen how you can use a backslash inside a string (in ECMAScript 5) to write a single string on multiple lines.  console.log('Here \  is \  a one line \  string \  written \  on 5 lines');  However, you cannot break up a JavaScript statement with a backslash.  **If a statement does not fit on a single line, it is best to break it after a comma or a binary operator.**Remember that JavaScript usually treats line breaks as semicolons only if it can’t parse the code without the semicolons.  It is also a good practice to indent the remainder of the statement for readability.  grade = 0.7 \* 95 +                0.15 \* 100 +                0.17 \* 92;  However, **never break a return statement after the 'return'.**  If a return statement returns a value, that value expression must begin on the same line as the return.  Otherwise JavaScript will insert a semicolon right after the return and the statement will just return undefined.  return  true  **is NOT the same as:**  return true;  2.12. Type Conversions  *Copyrighted Material - subject to fair use exception*  **Implicit Conversions:**  **JavaScript performs many type conversions implicitly.** We’ve already seen how JavaScript converts non Booleans to true and false values as needed.  The same is true for other types: when evaluating a given expression, JavaScript will convert numbers to strings and strings to numbers without our explicit consent.  **Addition and Concatenation:  +**  If the operands are numbers or Booleans, then JavaScript will add them.  Otherwise, JavaScript will convert ALL operands to strings and concatenate them.  This may lead to some unexpected results.  Let's try the following in the Firebug console:  >>> 10 + 5 + "$";  "15$"   The statement is executed from left to right so 10 and 5 are added first then 15 is converted to string and concatenated with "$": (10 + 5) + '$'  >>> "$" + 10 + 5;  "$105"  Here "$" and 10 are concatenated after 10 is converted to a string.  Then the result is concatenated with "5": ('$' + '10') + '5'  >>> true + 5  6  The Boolean true is converted to a number (1) then added to 5  >>> true + 'hello'  "truehello"  Here the Boolean true is converted to string first then concatenated with ‘hello’.  **Multiplication, Subtraction, Division:**  JavaScript will try to convert the operands to a number, or NaN if that is not possible:  >>>  '    5' \* '8.2'  41  >>> 5 \* 'hello'  NaN  >>> '5' /'hi'  NaN  >>> false – 7  // false is converted to 0  -7  >>> true  \* 4  // true is converted to 1  4  >>> ''  –  8  // the empty string is converted to 0  -8  **Equality:**  Because JavaScript converts values implicitly, the == equality operator does not always behave as you would expect.  >>> '' == 0  // the empty string  converts to number 0 before comparison  true  >>> "0" == 0   // the string "0" converts to number 0 before comparison.  true  >>> 0 == false  // the boolean false converts to number 0 before comparison.  true  >>> "0" == false // Both operands convert to numbers before comparison.  true  >>> null == undefined  true  That is why**it is best to use the strict equality operator** instead (=== and the corresponding ! ==).  >>> 0 === false  false  >>> "0" === false  false  >>> "" === 0  false  >>> "" !== 0  true  >>> 5 - 5 === 0  true  >>> 'foot' + 'hill' === 'foothill'  true  **Explicit Type Conversions:**  To carry out explicit conversions, you can use Number(), String() and Boolean():  **Number(**value**)** - Converts the given value to a number.  If the conversion is not possible, it returns NaN.  >>> Number("   .98")  0.98  >>> Number("Hello")  NaN  >>> Number("2.98e3")  2980  >>> Number("-2")  -2  **String(**value**)** - Converts the given value to a string.  >>> String (-2.7e3)  "-2700"  >>> String(false)  "false"  >>> String(15 + 20)  "35"  **Boolean(**value**) -**Converts the given value to a Boolean.  >>> Boolean(6.9)  true  >>> Boolean('false') // 'false' is a non-empty string  true  >>> Boolean('')  false  2.13. Other JavaScript Editors - Komodo Edit  *Copyrighted Material - subject to fair use exception*  We have introduced and used Scratchpad to create and edit our JavaScript code.  The main advantages of Scratchpad is that it allows us to run standalone JavaScript code and that it is available free of charge with Firefox on both Mac OS and Windows.  However Scratchpad lacks some features such as syntax checking and code completion:  these features become important as our programs grow in size and complexity.  There are several text editors and IDEs (Integrated Development Environments) available for JavaScript that offer these features.  We'll take a closer look at Komodo Edit here.  Komodo Edit is a free text editor available for Windows and Mac OS.  To download it, go to  [http://komodoide.com/komodo-edit/](http://komodoide.com/komodo-edit/" \t "_blank) and follow the instructions.  You may choose to start using Komodo Edit now to  take advantage of the syntax checking and code completion features.   However at this time, you won't be able to run your standalone JavaScript code from within Komodo Edit.  You'll have to save the file and open it in Scratchpad to run it and see your output in the Firebug console.  The following screencast illustrates how to navigate between Komodo Edit and Scratchpad to type in and then run JavaScript code. |  |

v