**JavaScripts Basic Notes**

Contents

[Operators 2](#_Toc431346113)

[Data Types 2](#_Toc431346114)

[ typeof Operator 3](#_Toc431346115)

[ Undefined 3](#_Toc431346116)

[ Empty Values 3](#_Toc431346117)

[ Null 3](#_Toc431346118)

[ Difference between Undefined and Null 3](#_Toc431346119)

[ Boolean 3](#_Toc431346120)

[Functions 3](#_Toc431346121)

[Objects 4](#_Toc431346122)

[Scope of variable 4](#_Toc431346123)

[Strings 4](#_Toc431346124)

[ String Properties 4](#_Toc431346125)

[ String Methods 4](#_Toc431346126)

[ Accessing a String as an Array is Unsafe 5](#_Toc431346127)

[Numbers 5](#_Toc431346128)

[o JavaScript Numbers are Always 64-bit Floating Point 5](#_Toc431346129)

[ Infinity 5](#_Toc431346130)

[ NaN - Not a Number 6](#_Toc431346131)

[ Number Properties 6](#_Toc431346132)

[ Global Methods 6](#_Toc431346133)

[ Number Methods 7](#_Toc431346134)

[Math Object 7](#_Toc431346135)

[ Math Constants 7](#_Toc431346136)

[ Math Object Methods 8](#_Toc431346137)

[Arrays 8](#_Toc431346138)

[ Array Vs Object 8](#_Toc431346139)

[ Deleting Elements 10](#_Toc431346140)

[Miscellaneous 10](#_Toc431346141)

[ Bitwise Operators 10](#_Toc431346142)

[ 4 Different Kinds of Loops 10](#_Toc431346143)

[ The Unary + Operator 11](#_Toc431346144)

[Regular Expression 11](#_Toc431346145)

[ Regular Expression Modifiers 11](#_Toc431346146)

[ Regular Expression Patterns 11](#_Toc431346147)

[ RegExp Object Properties 13](#_Toc431346148)

[ RegExp Object Methods 13](#_Toc431346149)

[ String functions that can be used with Regular Expression patterns 13](#_Toc431346150)

[Errors - Throw and Try to Catch 13](#_Toc431346151)

[Debugging 14](#_Toc431346152)

[ console.log() Method to display JavaScript values in the debugger window 14](#_Toc431346153)

[Hoisting 14](#_Toc431346154)

[Use Strict 14](#_Toc431346155)

[ Why Strict Mode? 15](#_Toc431346156)

[ It is a common mistake to forget that switch statements use strict comparison: 15](#_Toc431346157)

[JSON 15](#_Toc431346158)

[ What is JSON? 15](#_Toc431346159)

[ Much Like XML Because 15](#_Toc431346160)

[ Much Unlike XML Because 15](#_Toc431346161)

[ Why JSON? 15](#_Toc431346162)

JavaScripts Basic Notes

# Operators

* **7** **Arithmetic** Operators **+ - / \* % -- ++**

|  |  |
| --- | --- |
| Operator | Precedence |
| ( ) | Expression grouping |
| ++ -- | Increment and decrement |
| \* / % | Multiplication, division, and modulo division |
| + - | Addition and subtraction |

* When many operations have the same precedence (like addition and subtraction), they are computed from **left to right**.

var x = 100 + 50 - 3;//147

* If you add a number and a string, the result will be a string!

x = 5 + 5; //10   
y = "5" + 5;//55   
z= "Hello" + 5;//Hello5

* **6 Assignment** Operators **=, +=, -=, \*=, /=, %=**

# Data Types

* The latest ECMAScript standard defines **seven** data types:
* **6 Primitive Type:** [String, Number, Boolean, **undefined, null** ], Symbol (new to JavaScript in ECMAScript Edition 6)
* **1 composite (reference) data types:** **Object** (like Array & typed arrays, Date, Function etc regular objects)
* **Intrinsic Object:** **Array**, **Boolean**, **Date**, **Error**, **Function**, **Global**, **JSON**,**Math**, **Number**, **Object**, **RegExp**, and **String** objects
* JavaScript evaluates expressions from left to right

var x = 16 + 4 **+** "Volvo";//20Volvo

var x = "Volvo" + 16 + 4;// Volvo164

var x = 16 + 4 **-** "Volvo";//Na N

* **Array indexes are zero-based**, which means the first item is [0], second is [1], and so on

var cars = ["Saab", "Volvo", "BMW"];

* Object

var person = {firstName:"John", lastName:"Doe", age:50};

* **All data types have a valueOf() and a toString() method**.
* A data type (e.g. Number, String) can be a primitive value (typeof = number) or an object (typeof = object).

## typeof Operator

* An **array** is a special type of **object**. Therefore typeof [1, 2, 3, 4] returns object.

typeof "John"                 // Returns string    
typeof 3.14                   // Returns number   
typeof false                  // Returns Boolean   
typeof [1,2,3,4]            // **Returns object**   
typeof {name:'John', age:34} // Returns object

## Undefined

* A variable without a value, has the value**undefined**. The typeof is also **undefined**.

var person = undefined;   // Value is undefined, type is undefined

## Empty Values

* An **empty value has nothing to do with undefined**.
* An empty string variable has **both a value and a type**.

var car = "";  // The value is "", the typeof is string

## Null

* null is "nothing". It is supposed to be something that doesn't exist.
* **Data type of null is an object**. null can be considered as **special object**.
* You can empty an object by setting it to null:

var person = null;  // Value is **null**, but type is still an **object**

* You can also empty **an object by setting it to undefined**:

var person = undefined; // **Value is undefined, type is undefined**

## Difference between Undefined and Null

* **null & undefined are same in value but different in definition or type.**

typeof undefined             // undefined   
typeof null                   // object   
null === undefined           // false   
**null == undefined**     **// true**

* Empty Array [] or empty object {} are not equal to null.
* Two objects (whether empty or not) are never equal.

{} == null or [] == null or {} == {}  or {v:'v'} == {v:'v'} or [] == []   or [1,2] == [1,2]    // false

## Boolean

* Everything Without a "Real" is False

**0**(zero), **""** (empty string), **undefined**, **null**, **false-** all represents Boolean data type false.

# Functions

* Function **parameters** are the **names** listed in the function definition.
* Function **arguments** are the real **values** received by the function when it is invoked.
* **toCelsius** refers to the function object, and **toCelsius()** refers to the function result

function toCelsius(f) {return (5/9) \* (f-32);}

# Objects

* access object properties/methods in two ways

*objectName.propertyName //or*

*objectName["propertyName"]*

*objectName.methodName()*

* Do Not Declare Strings, Numbers, and Booleans as Objects**! As they complicate your code and slow down execution speed. i.e.**

var x = new String(); //Avoid String, Number, and Boolean objects.

# Scope of variable

* If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.

function myFunction() {   carName = "Volvo"; }

* In HTML, the global scope is the **window object**: All global variables belong to the window object.
* The lifetime of a JavaScript variable starts when it is declared.
* Local variables are deleted when the function is completed.
* Global variables are deleted when you close the page.

## Strings

* var x = "John"; // typeof x will return string   
  var y = new String("John"); // typeof y will return object

// (x **==** y) is **true** because x and y have equal values

// (x **===** y) is **false** because x and y have different types

* With JavaScript, methods and properties are also available to primitive values, because JavaScript treats primitive values as objects when executing methods and properties.

## String Properties

|  |  |
| --- | --- |
| Property | Description |
| Constructor | Returns the function that created the String object's prototype |
| Length | Returns the length of a string |
| Prototype | Allows you to add properties and methods to an object |

## String Methods

|  |  |
| --- | --- |
| Method | Description |
| charAt(n) | Returns the character at the specified index (position) |
| charCodeAt(n) | Returns the Unicode of the character at the specified index |
| concat(objs) | Joins two or more strings, **and returns a copy** of the joined strings. Str1.concat(str2,str3, ...); |
| fromCharCode() | Converts Unicode values to characters |
| indexOf(str) | Returns the position of the **first found** **occurrence** of a specified value in a string. **return -1 if the text is not found** |
| lastIndexOf(str) | Returns the position of the **last found occurrence** of a specified value in a string. **return -1 if the text is not found** |
| localeCompare() | Compares two strings in the current locale |
| match() | Searches a string for a match against a regular expression, and returns the matches |
| replace(Old, new) | Searches a string for a value and returns a new string with the value replaced. can also take a regular expression as the search value |
| search(str) | Searches a string for a value and returns the position of the match |
| slice(start, end) | Extracts a part of a string and returns a new string. **Accept -ive indexes & one arg** |
| split(splitwith) | **Splits a string into an array of substrings.** txt.split(""); // Split in characters |
| substr(start, length) | Extracts a part of a string from a start position through a number of characters. **First** **arg can be negative but second can’t. 2nd can be omitted.** |
| substring(start, end) | Extracts a part of a string between two specified positions. **cannot accept negative indexes but accept one arg.** |
| toLocaleLowerCase() | Converts a string to lowercase letters, according to the host's locale |
| toLocaleUpperCase() | Converts a string to uppercase letters, according to the host's locale |
| toLowerCase() | Converts a string to lowercase letters |
| toString() | Returns the value of a String object |
| toUpperCase() | Converts a string to uppercase letters |
| trim() | Removes whitespace from both ends of a string |
| valueOf() | Returns the primitive value of a String object |

## Accessing a String as an Array is Unsafe

var str = "HELLO WORLD";   
str[0];                   // returns H

* This is unsafe and unpredictable:
* It does not work in all browsers (not in IE5, IE6, IE7)
* It makes strings look like arrays (but they are not)
* str[0] = "H" does not give an error (but does not work)

## Numbers

## JavaScript Numbers are Always 64-bit Floating Point

|  |  |  |
| --- | --- | --- |
| Value (aka Fraction/Mantissa) | Exponent | Sign |
| 52 bits (0 - 51) | 11 bits (52 - 62) | 1 bit (63) |

* **Integers** (numbers without a period or exponent notation) are **considered accurate up to 15 digits**.

var x = 999999999999999;   // x will be 999999999999999   
var y = 9999999999999999;  // y will be 10000000000000000

* **maximum number of decimals is 17**, but floating point arithmetic is not always 100% accurate:

var x = 0.2 + 0.1;         // x will be 0.30000000000000004

var x = (0.2 \* 10 + 0.1 \* 10) / 10;       // x will be 0.3 To solve the problem above

* The Number () function converts an empty string to 0, and a non numeric string to NaN.

## Infinity

* Infinity (or -Infinity) is the value JavaScript will return if you calculate a number outside the largest possible number.
* Division by 0 (zero) also generates Infinity

var x =  2 / 0;          // x will be Infinity   
var y = -2 / 0;          // y will be –Infinity

* Infinity is a number: **typeOf** Infinity returns number.

typeof Infinity;        // returns "number"

## NaN - Not a Number

* It’s **value is not a number**

var x = 100 / "Apple";  // x will be NaN (Not a Number)

var x = 100 / "10";     // x will be 10 **working**

* **Global** JavaScript function **isNaN()** to find out if a **value** is a number.

var x = 100 / "Apple";   
isNaN(x);               // returns true because x is Not a Number

* If you use NaN in a mathematical operation, the result will also be NaN:

var x = NaN,  y = 5;   
var z = x + y;         // z will be NaN

Or the **result might be a concatenation**:

var x = NaN,  y = "5";   
var z = x + y;         // z will be NaN5

* Numbers Can be Objects: **typeof** NaN

var x = 123; // **typeof**  x returns number   
var y = new Number(123); // **typeof**  y returns object

var z = new Number(123);

// (x **==** y) is **true** because x and y have equal values

// (x **===** y) is **false** because x and y have different types

// (**z** **==** y) is **false** because objects cannot be compared

## Number Properties

* Primitive values (like 3.14 or 2014), **cannot have properties and methods** (because they are not objects).
* But with JavaScript, methods **and properties are also available to primitive values**, because **JavaScript treats primitive values as objects when executing methods and properties**.

|  |  |
| --- | --- |
| Property | Description |
| MAX\_VALUE | Returns the largest number possible in JavaScript. 1.7 e+308 |
| MIN\_VALUE | Returns the smallest number possible in JavaScript. 5e-324 |
| NEGATIVE\_INFINITY | Represents negative infinity (returned on overflow) |
| NaN | Represents a "Not-a-Number" value |
| POSITIVE\_INFINITY | Represents infinity (returned on overflow) |

* Number properties belong to the JavaScript's number object wrapper called **Number**. These **properties can only be accessed as Number.MAX\_VALUE**.

var x = Number.MAX\_VALUE; // accessible and 1.7976931348623157e+308

var x = 6, y = x.MAX\_VALUE;    // y becomes **undefined**

var z = new number(6), y = x.MAX\_VALUE;    // y becomes **undefined**

## Global Methods

|  |  |
| --- | --- |
| Method | **Description** |
| Number() | Returns a number, converted from its argument. Spaces are **not allowed**. If the number cannot be converted, NaN is returned. Used on Date(), the Number() method returns the number of milliseconds since 1.1.1970. |
| parseFloat() | Parses its argument and returns a floating point number. Spaces are allowed. Only the first number is returned. If the number cannot be converted, NaN is returned. |
| parseInt() | Parses its argument and returns an integer. Spaces are allowed. Only the first number is returned. If the number cannot be converted, NaN (Not a Number) is returned. |

Number(false); // returns 0

Number(new Date()); // returns 1404568027739

Number("10 20 30");  // returns NaN

parseInt("10");         // returns 10   
parseInt("10.33");      // returns 10   
parseInt("10 20 30");   // returns 10   
parseInt("10 years");   // returns 10   
parseInt("years 10");   // returns NaN

parseFloat("10");        // returns 10   
parseFloat("10.33");     // returns 10.33   
parseFloat("10 20 30");  // returns 10   
parseFloat("10 years");  // returns 10   
parseFloat("years 10");  // returns NaN

## Number Methods

* All number methods **return a new value**. They **do not change the original variable**.

|  |  |
| --- | --- |
| Method | Description |
| toString(base) | Returns a number as a string. **Default base** is **10**. |
| toExponential() | Returns a string, with a number rounded and written using exponential notation. Parameter = number of characters behind the decimal point & it’s optional. |
| toFixed() | Returns a string, with a **number rounded** and Parameter = number of characters behind the decimal point & its **default value** is **0**. |
| toPrecision() | Returns a string, with a number written with a **specified length.** Parameter = length of numbers & it’s optional. |
| valueOf() | Returns a number as a number |

## Math Object

* Unlike Number (which is function) it is an Object.

## Math Constants

|  |  |
| --- | --- |
| Property | Description |
| E | Returns Euler's number (approx. 2.718) |
| LN2 | Returns the natural logarithm of 2 (approx. 0.693) |
| LN10 | Returns the natural logarithm of 10 (approx. 2.302) |
| LOG2E | Returns the base-2 logarithm of E (approx. 1.442) |
| LOG10E | Returns the base-10 logarithm of E (approx. 0.434) |
| PI | Returns PI (approx. 3.14) |
| SQRT1\_2 | Returns the square root of 1/2 (approx. 0.707) |
| SQRT2 | Returns the square root of 2 (approx. 1.414) |

## Math Object Methods

|  |  |
| --- | --- |
| Method | Description |
| abs(x) | Returns the absolute value of x |
| acos(x) | Returns the arccosine of x, in radians. x = -1 to 1, the method will return NaN |
| asin(x) | Returns the arcsine of x, in radians. x = -1 to 1, the method will return NaN |
| atan(x) | Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians or NaN if the value is empty. |
| atan2(y,x) | Returns the arctangent of the quotient of its arguments. I.e. Angle between (x,y) coordinate and positive x-axis. |
| ceil(x) | Returns x, rounded upwards to the nearest integer |
| cos(x) | Returns the cosine of x (x is in radians) |
| exp(x) | Returns the value of Ex |
| floor(x) | Returns x, rounded downwards to the nearest integer |
| log(x) | Returns the natural logarithm (base E) of x |
| max(x,y,z,...,n) | Returns the number with the highest value |
| min(x,y,z,...,n) | Returns the number with the lowest value |
| pow(x,y) | Returns the value of x to the power of y |
| random() | Returns a random number between 0 and 1 |
| round(x) | Rounds x to the nearest integer |
| sin(x) | Returns the sine of x (x is in radians) |
| sqrt(x) | Returns the square root of x |
| tan(x) | Returns the tangent of an angle |

## Arrays

* Arrays are a special type of objects. The **typeof** operator in JavaScript returns "object" for arrays.
* Because of this, you can have variables of different types in the same Array.
* Don’t use **new** Array(). **Use [] instead** for simplicity, readability and execution speed.

## Array Vs Object

* In JavaScript, **arrays** use **numbered indexes**, whereas **objects** use **named indexes**.
* How do I know if a variable is an array? As **typeof** both object and array is Object. To solve the problem one can create its own **isArray()** function.

function **isArray**(myArray) {   
    return myArray.constructor.toString().indexOf("Array") > -1;   
}

The function above always returns true if the argument is an array. Above function will search index of Array in constructor string function Array() { [native code] }

Or more precisely: **it returns true** if the object prototype of the argument is **"[object array]"**.

* Ticks

var fruits = ["Banana", "Orange", "Apple", "Mango"];   
fruits[fruits.length] = "Lemon";     // adds a new element (Lemon) to fruits

fruits[10] = "Grapes";                // adds a new element (Grapes) to fruits

fruits[8]; // return **undefined**

* Example: why not use new Array. complicates your code and produces nasty side effects

var points = new Array(40, 100);  // Creates an array with two elements (40 and 100)

var points = new Array(40);       // Creates an array with 40 undefined elements !!!!!

|  |  |
| --- | --- |
| Method | Description |
| concat(*a1,a2,...*) | Joins two or more arrays, and returns a copy of the joined arrays |
| indexOf(*item, start*) | Search the array for an element and returns its position.*Item-*required*, start*- Optional. Default value **0**. Negative values to start counting from the end, and search to the end. *Return* -1 id not found. |
| join(*separator*) | Joins all elements of an array into a string. Default *separator* is **comma**. |
| lastIndexOf(*item, start*) | Search the array for an element, starting at the end, and returns its position. *Item-*required*, start*- Optional. Default value **0**. Negative values to start counting from the end, and search to the beginning. *Return* -1 id not found |
| pop() | **Removes the last element** of an array, and **returns that element** |
| push(*i1,i2,i3,...*) | **Adds new elements** to the **end** of an array, and **returns the new length** |
| reverse() | Reverses the order of the elements in an array. |
| shift() | **Removes the first element** of an array, and **returns that element** |
| slice(start, end) | Selects a part of an array, and returns the new array. Start*-required and can be negative value*, end-optional & can be negative value. Item indexed at end will not get sliced. Note: start counting from 1 from end rather than 0. |
| sort(compareFunction) | Sorts the elements of an array. **compareFunction** Optional. A function that defines an alternative sort order, like:  function(a, b){return a-b} // ascending  When the sort() method compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.  **Default compareFunction compares only strings.** |
| splice(*index,howmany,i1,i2,...*) | Adds/Removes elements from an array. *index****-*** required & can be negative. *howmany* - Required. The number of items to be removed. If set to 0, no items will be removed. *i1,i2,...* Optional. The new item(s) to be added to the array. *Return* A new Array, containing the removed items (if any). |
| toString() | Converts an array to a string, and returns the result. **Note**: The returned string will separate the elements with commas. |
| unshift(*i1,i2,i3,...*) | **Adds new elements** to the **beginning** of an array, and **returns the new length.** |
| valueOf() | Returns the primitive value of an array, separated by commas.  var fruits = ["Banana", "Orange", "Apple", "Mango"];  fruits.toString() == fruits.valueOf();           // return true  fruits.toString() === fruits.valueOf();        // return false  // typeof’s fruits.toString= String & fruits.valueOf = Object |

## Deleting Elements

* Using **delete** on array elements leaves **undefined holes** in the array.

var fruits = ["Banana", "Orange", "Apple", "Mango"];   
delete fruits[0];           // Changes the first element in fruits to **undefined**

## Miscellaneous

## Bitwise Operators

* Bit operators work on 32-bit numbers.
* Any numeric operand in the operation is converted into a 32-bit number.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Operator | Description | Example | Same as | Result | Decimal |
| & | AND | x = 5 & 1 | 0101 & 0001 | 0001 | 1 |
| | | OR | x = 5 | 1 | 0101 | 0001 | 0101 | 5 |
| ~ | NOT | x = ~ 5 | ~0101 | 1010 | 10 |
| ^ | XOR | x = 5 ^ 1 | 0101 ^ 0001 | 0100 | 4 |
| << | Left shift | x = 5 << 1 | 0101 << 1 | 1010 | 10 |
| >> | Right shift | x = 5 >> 1 | 0101 >> 1 | 0010 | 2 |

The examples above uses **4 bits unsigned** examples. But JavaScript uses 32-bit signed numbers.   
Because of this, in JavaScript, ~ 5 will not return 10. It will return -6.   
**~00000000000000000000000000000101 will return 11111111111111111111111111111010**

## 4 Different Kinds of Loops

* **for** - loops through a block of code a number of times

for (i = 0; i < 5; i++){ code block to be executed }

* **for/in** - loops through the properties of an object

for (x in person) { code block to be executed }

* **while** - loops through a block of code while a specified condition is true

while (*condition*) { code block to be executed }

* **do/while** - also loops through a block of code while a specified condition is true

do {   
    code block to be executed   
}while (*condition*);

## The Unary + Operator

* The **unary + operator** can be used to convert a variable to a number:

var y = "5";      // y is a string   
var x = + y;      // x is a number

If the variable cannot be converted, it will still become a number, but with the value NaN:

var y = "John";   // y is a string   
var x = + y;      // x is a number (NaN)

* When JavaScript tries to operate on a "wrong" data type, it will try to convert the value to a "right" type.

5 + null    // returns 5         because **null is converted to 0**   
"5" + null  // returns "5null"   because **null is converted to "null"**   
"5" + 2     // returns 52        because **2 is converted to "2"**   
"5" - 2     // returns 3         because **"5" is converted to 5**   
"5" \* "2"   // returns 10        because "**5" and "2" are converted to 5 and 2**

Think operator on string then on Number then on other type for automatic conversions

<http://www.w3schools.com/js/js_type_conversion.asp>

## Regular Expression

* It is a **sequence of characters** that forms a **search pattern**.

var patt = /pattern/modifiers; // syntax Or

var patt2 = new RegExp("regexp","g") // syntax

* **typeof** Regular Expression is **Object**
* It will **return null** if could not find the pattern in string.

## Regular Expression Modifiers

|  |  |
| --- | --- |
| Modifier | Description |
| i | Perform case-insensitive **match**ing |
| g | Perform a global **match** (find all matches rather than stopping after the first match) |
| m | Perform multiline **match**ing. The m modifier treat **beginning (^)** and **end ($)** characters to match the beginning or end of **each line** of a string (**delimited by \n or \r**), rather than just the beginning or end of the string.  "\nIs th\nis it is?".match(/^is/m) |

## Regular Expression Patterns

* **Brackets** are used to find a range of characters:

|  |  |
| --- | --- |
| Expression | Description |
| [abc] | Find any of the characters between the brackets |
| [^abc] | Find any character **NOT** between the brackets |
| [0-9] | Find any of the digits between the brackets |
|  | Find any digit **NOT** between the brackets |
| (x|y) | Find any of the alternatives separated with | |

* **Metacharacters**arecharacters with a special meaning:

|  |  |
| --- | --- |
| Metacharacter | Description |
| . | Find a single character, except newline or line terminator. **/h.t/** |
| \w | Find a word character. A word character is a character from a-z, A-Z, 0-9, including the \_ (underscore) character. **/\w/** |
| \W | Find a non-word character. **/\W/** |
| \d | Find a digit. **/\d/** |
| \D | Find a non-digit character **/\D/** |
| \s | Find a whitespace character. **/\s/** |
| \S | Find a non-whitespace character. **/\S/** |
| \b | Find a match at the beginning/end of a word (say W3). **/\b**W3**/** |
| \B | Find a match not at the beginning/end of a word (say W3). **/\B**W3**/** |
| \0 | Find a NUL character. **/\0/** Note: strings does’t end with NULL character in JS |
| \n | Find a new line character. **/\n/** |
| \f | Find a form feed character. **/\f/** |
| \r | Find a carriage return character. **/\r/** |
| \t | Find a tab character. **/\t/** |
| \v | Find a vertical tab character. **/\v/** |
| \xxx | Find the character specified by an octal number xxx. **/\127/g** |
| \xdd | Find the character specified by a hexadecimal number dd. **/\x57/g** |
| \uxxxx | Find the Unicode character specified by a hexadecimal number xxxx. **/\u0057/g** |

* **Quantifiers**define quantities:

|  |  |
| --- | --- |
| Quantifier | Description |
| n+ | Matches any string that contains at least one n |
| n\* | Matches any string that contains zero or more occurrences of n |
| n? | Matches any string that contains zero or one occurrences of n |
| n{X} | Matches any string that contains a sequence of *X* *n*'s **/\d{4}/g** |
| n{X,Y} | Matches any string that contains a sequence of X to Y *n*'s  **/\d{3,4}/g** |
| n{X,} | Matches any string that contains a sequence of at least X *n*'s **/\d{3,}/g** |
| n$ | Matches any string with n at the end of it **/is$/gi** |
| ^n | Matches any string with n at the beginning of it **/^is/gi** |
| ?=n | Matches any string that is followed by a specific string n  **/is(?= all)/g** |
| ?!n | Matches any string that is not followed by a specific string n  **/is(?! all)/gi** |

## RegExp Object Properties

|  |  |
| --- | --- |
| Property | Description |
| constructor | Returns the function that created the RegExp object's prototype |
| global | Checks whether the "g" modifier is set |
| ignoreCase | Checks whether the "i" modifier is set |
| lastIndex | Specifies the index at which to start the next match.   * **Note**: This property only works if the "g" modifier is set. * This property returns an integer that specifies the character position immediately after the last match found by exec( ) or test( ) methods. * **Note**: exec( ) and test( ) reset lastIndex to 0 if they do not get a match. |
| multiline | Checks whether the "m" modifier is set |
| source | Returns the text of the RegExp pattern, but not its modifiers |

## RegExp Object Methods

|  |  |
| --- | --- |
| Method | Description |
| compile() | Deprecated in version 1.5. Compiles a regular expression |
| exec(str) | Tests for a **match** in a string. Returns the first match |
| Test(str) | Tests for a **match** in a string. Returns true or false |
| toString() | Returns the string value of the regular expression |

## String functions that can be used with Regular Expression patterns

|  |  |
| --- | --- |
| Method | Description |
| search (patt) | Search RegExp in string. Returns **index of first match** |
| replace (patt, str) | Replace for **matches** in a string. Returns **new String**. |
| match (patt) | Find matches in string. Return **Array of matches** |

## Errors - Throw and Try to Catch

**try** {   
  *Block of code to try for errors*

**throw** *statements lets you create custom errors*  
}  
**catch**(err) {  
*Block of code to handle errors*  
}   
**finally** {  
    *Block of code to be executed regardless of the try / catch result*  
}

## Debugging

* Searching for errors in programming code is called code debugging.

## console.log() Method to display JavaScript values in the debugger window

* The **debugger** keyword stops the execution of JavaScript, and calls (if available) the debugging function.

var x = 15 \* 5;  
debugger;  
document.getElementbyId("demo").innerHTML = x;

## Hoisting

* Hoisting is JavaScript's default behaviour of moving all declarations to the top of the current scope (to the top of the current script or the current function).
* Hoisting applies **to variable declarations** and **to function declarations**.

x = 5; // Assign 5 to x  
  
elem = document.getElementById("demo"); // Find an element   
elem.innerHTML = x;                     // Display x in the element  
  
var x; // Declare x

* JavaScript only hoists declarations, not initializations.

var  x = 5; // Initialize x  
  
elem = document.getElementById("demo"); // Find an element   
elem.innerHTML = x + " " + y;           // Display x and y; **y is undefined up to here**  
  
var y = 7; // Initialize y

## Use Strict

* **"use strict";**  Defines that JavaScript code should be executed in "strict mode".
* It is not a statement, but a **literal expression**, ignored by earlier versions of JavaScript.
* The purpose of "use strict" is to indicate that the code should be executed in "strict mode".
* With strict mode, you cannot, for example, use undeclared variables.
* Strict mode is declared by adding "use strict"; to the beginning of a JavaScript file, or a JavaScript function.
* Declared at the beginning of a JavaScript file, it has global scope (all code will execute in strict mode):

"use strict";  
x = 3.14;       // This will cause an error (if x has not been declared)

* Declared inside a function, it has local scope (only the code inside the function is in strict mode):

x = 3.14;       // This will **not** cause an error.   
myFunction();  
  
function myFunction() {  
   "use strict";  
    y = 3.14;   // This will cause an error  
}

## Why Strict Mode?

* Strict mode makes it **easier to write "secure"** JavaScript.
* Strict mode **changes previously accepted "bad syntax" into real errors**.
* As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, **making it impossible to accidentally create** a global variable.
* In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.
* In strict mode, **any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error**.

[**http://www.w3schools.com/js/js\_strict.asp**](http://www.w3schools.com/js/js_strict.asp)

## It is a common mistake to forget that switch statements use strict comparison:

var x = 10;  
switch(x) {  
    case 10: alert("Hello); // This case switch will display an alert  
}  
switch(x) {  
    case "10": alert("Hi"); //This case switch will **NOT** display an alert  
}

## JSON

JSON is a format for storing and transporting data.

JSON is often used when data is sent from a server to a web page.

## What is JSON?

* JSON stands for **J**ava**S**cript **O**bject **N**otation
* JSON is lightweight data interchange format
* JSON is language independent **\***
* JSON is "self-describing" and easy to understand

## Much Like XML Because

* Both JSON and XML is "self describing" (human readable)
* Both JSON and XML is hierarchical (values within values)
* Both JSON and XML can be parsed and used by lots of programming languages
* Both JSON and XML can be fetched with an XMLHttpRequest

## Much Unlike XML Because

* JSON **doesn't use end tag**
* JSON is **shorter**
* JSON is **quicker** to read and write
* JSON can use **arrays**

## Why JSON?

* For AJAX applications, JSON is faster and easier than **XML**:
* Using XML

1. Fetch an XML document
2. Use the XML DOM to loop through the document
3. Extract values and store in variables

* Using JSON

1. Fetch a JSON string
2. JSON.Parse() the JSON string

* The biggest difference is:

XML has to be parsed with an XML parser, JSON can be parsed by a **standard JavaScript function**.