

The JavaScript Language

INTRODUCTION,
CORE JAVASCRIPT



POLITECNICO
DI TORINO

Laura Farinetti - DAUIN



What and why JavaScript?

- JavaScript is a lightweight, interpreted programming language with object-oriented capabilities primarily used in web browsers for dynamic web pages and user interaction
 - JavaScript made its first appearance in Netscape 2.0 in 1995
 - Later standardized by ECMA (www.ecma.ch): ECMAScript
- JavaScript is one of the 3 languages all web developers must learn
 - HTML to define the content of web pages
 - CSS to specify the layout of web pages
 - JavaScript to program the behavior of web pages

What can JavaScript do for us?

- JavaScript can handle events (mouse click, page load, ...)
- JavaScript can change
 - HTML content
 - HTML attributes
 - HTML styles (CSS)
- JavaScript can validate form data
- JavaScript can manage media and graphics
- JavaScript can work with HTML5 (HTML5 APIs)

http://www.w3schools.com/js/js_intro.asp

Short history

- 1995
 - May: “Mocha” is invented in Netscape by Brendan Eich
 - September: renamed to LiveScript
 - December: renamed to Javascript (because Java was popular)
- 1996: JavaScript is taken to standardization in ECMA
 - From now on ECMAScript is the specification, Javascript is an implementation (ActionScript is another implementation)
- 1997: ECMA-262 (ECMAScript)
- 1998: ECMAScript 2
- 1999: ECMAScript 3

Short history

- 2005: Mozilla and Macromedia started work on ECMAScript 4 (feature rich and a very major leap from ECMAScript 3)
- Yahoo and Microsoft opposed the forming standard, and ECMAScript 3.1 was the compromise
- 2009: Opposing parties meet in Oslo and achieve an agreement, and ES3.1 is renamed to ES5
 - In the spirit of peace and agreement, the new Javascript long term agenda is named “Harmony”
- 2015: ES6 (part of the “Harmony” umbrella)
 - Starting with ES6 version names will be based on the year of release, so ES6 is ES2015 and ES7 should be ES2016

JavaScripts

- A JavaScript consists of JavaScript statements placed within the `<script>... </script>` HTML tags in a web page
- The `<script>` tag containing JavaScript code can be placed anywhere in a web page
 - In the head or the body section

```
<html>
<body>
<script language="javascript" type="text/javascript">
<!--
    document.write("Hello World!")
//-->
</script>
</body>
</html>
```

prova.html

Where to embed JavaScript code?

- In the head section
 - Scripts to be executed when they are called, or when an event is triggered, go in the head section
 - When you place a script in the head section, you will ensure that the script is loaded before anyone uses it
- In the body section
 - Scripts to be executed when the page loads go in the body section
 - When you place a script in the body section it generates the content of the page

What can JavaScript do?

- Generate dialog boxes
- Redirect a page
- Open new browser windows (pop-ups)
- Intercept mouse events
 - Clicks on links, buttons, ...
 - Mouse-overs, ...
- Read user input in forms
- Modify HTML pages
 - Add/remove content
 - Change images
 - Modify form controls

What you need to know...

- JS variables and expressions
- JS language constructs (if, while, ...)
- JS objects
 - The most important built-in objects
- Interaction with the user
 - Mouse, keyboard
- Interaction with the browser
 - Windows, pages
- Interaction with the page: the Document Object Model

Summary

- Core JavaScript
 - Lexical structure
 - Types, values, and variables
 - Expressions and operators
 - Statements
 - Objects
 - Arrays
 - Functions
 - Classes and modules
 - Pattern matching with regular expressions

Summary

- Client-Side JavaScript
 - JavaScript in web browsers
 - The window object
 - Scripting documents (DOM)
 - Scripting CSS
 - Handling events
 - Client-Side Storage
 - Scripted Media and Graphics
 - HTML5 APIs

JavaScript basics

- Syntax is similar to the C language
- Case-sensitive language
- Uses the Unicode character set
- Ignores spaces and line breaks
- Semi-colons (at the end of a line) can be omitted
- Comments

```
// 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.  
*/
```

Literals

- Data values that appear directly in a program
- Examples

```
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
```

Types, values and variables

- JavaScript types can be divided into two categories
 - Primitive types: numbers, strings, Booleans and the special JavaScript values “null” and “undefined”
 - Object types: any JavaScript value that is not a primitive type
- An object (i.e., a member of the type object) is a collection of properties where each property has a name and a value
- JavaScript defines two special kind of objects
 - an “array”: an ordered collection of numbered values
 - a “function”: an object that has executable code associated

Types, values and variables

- In JavaScript all variables must be declared before their use with the “var” keyword
- 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
- JavaScript uses lexical scoping
 - Variables declared outside of a function are global variables and are visible everywhere in a JavaScript program
 - Variables declared inside a function have function scope and are visible only to code that appears inside that function

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
 - 64-bit floating-point format defined by the IEEE 754 standard

```
0
3
10000000
0xff // hexadecimal
0xCAFE911 // hexadecimal
3.14
2345.789
.33333333333333333333
6.02e23 // 6.02 × 1023
1.4738223E-32 // 1.4738223 × 10-32
```


Arithmetic in JavaScript

- Numeric operators: + - * / %
- Set of functions and constants defined as properties of the Math object

```
Math.pow(2,53)    // => 9007199254740992: 2 to the power 53
Math.round(.6)   // => 1.0: round to the nearest integer
Math.ceil(.6)    // => 1.0: round up to an integer
Math.floor(.6)   // => 0.0: round down to an integer
Math.abs(-5)     // => 5: absolute value
Math.max(x,y,z)  // Return the largest argument
Math.min(x,y,z)  // Return the smallest argument
Math.random()    // Pseudo-random number x where 0 <= x < 1.0
Math.PI          // π: circumference of a circle / diameter
Math.E           // e: The base of the natural logarithm
Math.sqrt(3)     // The square root of 3
Math.pow(3, 1/3) // The cube root of 3
Math.sin(0)      // Trigonometry: also Math.cos, Math.atan, etc.
Math.log(10)     // Natural logarithm of 10
Math.log(100)/Math.LN10 // Base 10 logarithm of 100
Math.log(512)/Math.LN2 // Base 2 logarithm of 512
```

Text

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

String literals

- Examples

```
"" // 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"
'You\'re right, it can\'t be a quote' // escape sequence
```

- In client-side JavaScript programming, JavaScript code may contain strings of HTML code, and HTML code may contain strings of JavaScript code

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

String operators, properties and methods

- Concatenation

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

- The only property is
 - `.length` (the number of characters in the string)
- Many general methods
 - `.charAt()`, `.concat()`, `.indexOf()`, `.localeCompare()`, `.match()`, `.replace()`, `.search()`, `.slice()`, `.split()`, `.substr()`, `.substring()`, `.toLowerCase()`, `.toUpperCase()`, `.toString()`, `.valueOf()`, ...
- Many methods specific for writing HTML

String methods for HTML formatting

- Methods that returns a copy of the string wrapped inside the appropriate HTML tag
 - Warning: not standard methods, may not work as expected in all browsers
- List of main methods
 - .big(), .small(),
.italic(), .bold(),
.fixed(), .sub(), .sup()
 - .fontcolor(c),
.fontsize(s)
 - .anchor("name"),
.link("url")

```
var str = "Hello World!";  
document.write(str);  
document.write("<br />");  
str = str.fontcolor("red");  
document.write(str + "<br/>");  
str = str.fontsize(7);  
document.write(str);
```

Main Javascript operators

- Numeric operators
+ - * / %
- Increment operators
++ --
- Assignment operators
= += -= *= /= %=
- String operator
+ (concatenation)
- Comparison operators
– == (same value) === (same value and same type)
– != > < >= <=
- Boolean and Logic operators
– && (logical “and”) || (logical “or”) ! (logical “not”)

Statements

- Conditionals (e.g. if, switch)
 - Make the JavaScript interpreter execute or skip other statements depending on the value of an expression
- Loops (e.g. while, for)
 - Execute other statements repetitively
- Jumps (e.g. break, return, throw)
 - Cause the interpreter to jump to another part of the program

If statement

```
if (condition)
{
    ...code...
}
```

```
if (condition)
{
    ...code if true...
}
else
{
    ...code if false...
}
```

```
if (condition1)
{
    ...code if 1 true...
}
else if (condition2)
{
    ...code if 2 true...
}
else
{
    ...if both false...
}
```


Choice statement

```
switch(n)
{
  case 1:
    code block 1
    break

  case 2:
    code block 2
    break

  default:
    code to be executed if n is
    different from case 1 and 2
}
```

Loop statements


```
for ( var=startvalue; var<=endvalue; var=var+increment )  
{  
    code to be executed  
}
```

```
while ( condition_is_true )  
{  
    code to be executed  
}
```


```
do {  
    code to be executed  
} while ( condition_is_true )
```

Jump statements

```
while ( ... ) // or for
{
  code
  break
  code
}
```




```
while ( ... ) // or for
{
  code
  continue
  code
}
```



Objects

- An object is a composite value
 - It aggregates multiple values (primitive values or other objects) and allows to store and retrieve those values by name
 - Unordered collection of properties, each of which has a name and a value
- Property names are strings: objects map strings to values
 - Similar to fundamental data structure called “hash”, “hashtable”, “dictionary” or “associative array”
- However an object is more than a simple string-to-value map: it also inherits the properties of another object, known as its “prototype”
 - The methods of an object are typically inherited properties, and this “prototypal inheritance” is a key feature of JavaScript

Object example

Object	Properties	Methods
	<code>car.name = Fiat</code> <code>car.model = 500</code> <code>car.weight = 850kg</code> <code>car.color = white</code>	<code>car.start()</code> <code>car.drive()</code> <code>car.brake()</code> <code>car.stop()</code>

- All cars have the same properties, but the property values differ from car to car
- All cars have the same methods, but the methods are performed at different times

Objects

- JavaScript objects are dynamic: properties can usually be added and deleted
- Any value in JavaScript that is not a string, a number, true, false, null, or undefined is an object
- The most common operations to do with objects are create them and set, query, delete, test, and enumerate their properties
 - ES2015 added several advanced operations on objects

Object properties

- A property has a name and a value
 - The name may be any string, including the empty string, but no object may have two properties with the same name
- The value may be any JavaScript value, or (in ES2015) it may be a getter or a setter function
- In addition each property has associated values called property attributes
 - The writable attribute specifies whether the value of the property can be set
 - The enumerable attribute specifies whether the property name is returned by a for/in loop
 - The configurable attribute specifies whether the property can be deleted and whether its attributes can be altered

Object attributes

- In addition to its properties, every object has three associated object attributes:
 - An object's prototype is a reference to another object from which properties are inherited
 - An object's class is a string that categorizes the type of an object
 - An object's extensible flag specifies (in ES6) whether new properties may be added to the object

Terminology

- Three broad categories of JavaScript objects
 - A native object is an object or class of objects defined by the ECMAScript specification: arrays, functions, dates, and regular expressions (for example) are native objects
 - A host object is an object defined by the host environment (such as a web browser) within which the JavaScript interpreter is embedded: the HTML element objects that represent the structure of a web page in client-side JavaScript are host objects
 - A user-defined object is any object created by the execution of JavaScript code
- Two types of properties
 - A own property is a property defined directly on an object
 - An inherited property is a property defined by an object's prototype object

Creating objects

- Objects can be created with object literals, with the “new” keyword, and (in ES2015) with the `Object.create()` function
- The easiest way to create an object is to include an object literal in the JavaScript code

```
var empty = {}; // An object with no properties
var point = { x:0, y:0 }; // Two properties
var point2 = { x:point.x, y:point.y+1 }; // More complex values
var book = {
  "main title": "JavaScript", // Property names include spaces
  'sub-title': "The Definitive Guide", // and hyphens, so
                                     use string literals
  "for": "all audiences", // for is a reserved word, so quote
  author: { // The value of this property is
    firstname: "David", // itself an object. Note that
    surname: "Flanagan" // these property names are unquoted.
  }
};
```

Creating objects

- The “new” operator creates and initializes a new object
 - The new keyword must be followed by a function invocation
 - A function used in this way is called a constructor and serves to initialize a newly created object
 - Core JavaScript includes built-in constructors for native types

```
var o = new Object(); // Create an empty object: same as {}
var a = new Array(); // Create an empty array: same as []
var d = new Date(); // Create a Date object representing
                    // the current time
var r = new RegExp("js"); // Create a RegExp object for pattern
                          // matching
```

Querying and setting properties

- To obtain the value of a property: dot (.) or square bracket ([]) operators on the right side of the assignment expression

```
var author = book.author;    // Get the "author" property of the book.
var name = author.surname   // Get the "surname" property of the
author.
var title = book["main title"] // Get the "main title" property of
the book.
```

- To create or set a property: dot (.) or square bracket ([]) operators on the left side of the assignment expression

```
book.edition = 6;           // Create an "edition" property of book.
book["main title"] = "ECMAScript"; // Set the "main title" property.
```

Examples

```
var person = new Object();  
person.name = "Nicholas";  
person.age = 29;
```

```
var person = {  
  name : "Nicholas",  
  age : 29  
};
```

```
var person = {}; //same as new Object()  
person.name = "Nicholas";  
person.age = 29;
```

```
alert(person["name"]); // "Nicholas"  
alert(person.name); // "Nicholas"
```

Operations on properties

- Deleting Properties

- The delete operator removes a property from an object

```
delete book.author; // The book object now has no author property.
delete book["main title"]; // Now it doesn't have "main title", either.
```

- Testing properties

- Check whether an object has a property with a given name: in operator, hasOwnProperty() method, or simply by querying the property

```
var o = { x: 1 }
"x" in o; // true: o has an own property "x"
"y" in o; // false: o doesn't have a property "y"

o.hasOwnProperty("x"); // true: o has an own property x
o.hasOwnProperty("y"); // false: o doesn't have a property y

o.x !== undefined; // true: o has a property x
o.y !== undefined; // false: o doesn't have a property y
```

Arrays

- An array is an ordered collection of values
 - Each value is called an element, and each element has a numeric position in the array, known as its index
- JavaScript arrays are untyped: an array element may be of any type, and different elements of the same array may be of different types
- Creating arrays
 - With array literals
 - With the `Array()` constructor
- Reading and Writing Array Elements
 - Access to an element of an array: `[]` operator

Examples

```
var empty = []; // An array with no elements
var primes = [2, 3, 5, 7, 11]; // An array with 5 numeric elements
var misc = [ 1.1, true, "a" ]; // 3 elements of various types

var base = 1024; // The values in an array literal need
var table = [base, base+1, base+2, base+3]; // not be constants

var b = [[1,{x:1, y:2}], [2, {x:3, y:4}]]; // can contain object literals
// or other array literals

var count = [1,,3]; // An array with 3 elements, the middle one undefined.

var a = new Array(); // An array with no elements
var a = new Array(10);
var a = new Array(5, 4, 3, 2, 1, "testing, testing");

var a = ["world"]; // Start with a one-element array
var value = a[0]; // Read element 0
a[1] = 3.14; // Write element 1
i = 2;
a[i] = 3; // Write element 2
a[i + 1] = "hello"; // Write element 3
a[a[i]] = a[0]; // Read elements 0 and 2, write element 3
a.length // => 4
```


Array methods

- See references
 - join()
 - reverse()
 - sort()
 - concat()
 - slice()
 - splice()
 - push() and pop()
 - unshift() and shift()
 - toString()
- Several more in ES2015
 - E.g., forEach()

Functions

- A function is a block of JavaScript code that is defined once but may be executed, or invoked, any number of times
- JavaScript functions are parameterized
 - A function definition may include a list of identifiers, known as parameters, that work as local variables for the body of the function
 - Function invocations provide values, or arguments, for the function's parameters
- Functions often use their argument values to compute a return value that becomes the value of the function-invocation expression
- In addition to the arguments, each invocation has another value—the invocation context—that is the value of the `this` keyword
- If a function is assigned to the property of an object, it is known as a method of that object

Defining functions

```
// Print the name and value of each property of o. Return undefined.
function printprops(o) {
  for(var p in o)
    console.log(p + ": " + o[p] + "\n");
}

// Compute the distance between Cartesian points (x1,y1) and (x2,y2).
function distance(x1, y1, x2, y2) {
  var dx = x2 - x1;
  var dy = y2 - y1;
  return Math.sqrt(dx*dx + dy*dy);
}

// A recursive function (one that calls itself) that computes factorials
// Recall that x! is the product of x and all positive integers less than it.
function factorial(x) {
  if (x <= 1) return 1;
  return x * factorial(x-1);
}

// This function expression defines a function that squares its argument.
// Note that we assign it to a variable
var square = function(x) { return x*x; }

// Function expressions can also be used as arguments to other functions:
data.sort(function(a,b) { return a-b; });
```

Invoking functions

- JavaScript functions can be invoked in four ways

- As functions

```
printprops({x:1});  
var total = distance(0,0,2,1) + distance(2,1,3,5);  
var probability = factorial(5)/factorial(13);
```

- As methods

```
var calculator = { // An object literal  
  operand1: 1,  
  operand2: 1,  
  add: function() {  
    // Note the use of the this keyword to refer to this object.  
    this.result = this.operand1 + this.operand2;  
  }  
};  
calculator.add(); // A method invocation to compute 1+1.  
calculator.result // => 2
```

- As constructors
- Indirectly through their call() and apply() methods

Classes

- It is often useful to define a class of objects that share certain properties
- Members, or instances, of the class have their own properties to hold or define their state, but they also have properties (typically methods) that define their behavior
- This behavior is defined by the class and is shared by all instances
- Example
 - A class named Complex represents and performs arithmetic on complex numbers
 - A Complex instance have properties to hold the real and imaginary parts of the complex number (state)
 - The Complex class defines methods to perform addition and multiplication (behavior)
- In JavaScript, a class is a set of objects that inherit properties from the same prototype object

Example

```
/*
 * Complex.js:
 * This file defines a Complex class to represent complex numbers.
 * Recall that a complex number is the sum of a real number and an
 * imaginary number and that the imaginary number i is the square root of -1.
 */
/*
 * This constructor function defines the instance fields r and i on every
 * instance it creates. These fields hold the real and imaginary parts of
 * the complex number: they are the state of the object.
 */
function Complex(real, imaginary) {
  if (isNaN(real) || isNaN(imaginary)) // Ensure that both args are numbers.
    throw new TypeError(); // Throw an error if they are not.
  this.r = real; // The real part of the complex number.
  this.i = imaginary; // The imaginary part of the number.
}
/*
 * The instance methods of a class are defined as function-valued properties
 * of the prototype object. The methods defined here are inherited by all
 * instances and provide the shared behavior of the class. Note that JavaScript
 * instance methods must use the this keyword to access the instance fields.
 */
// Add a complex number to this one and return the sum in a new object.
Complex.prototype.add = function(that) {
  return new Complex(this.r + that.r, this.i + that.i);
};
```

Example

```
// Multiply this complex number by another and return the product.
Complex.prototype.mul = function(that) {
    return new Complex(this.r * that.r - this.i * that.i, this.r * that.i + this.i * that.r);
};
// Return the real magnitude of a complex number. This is defined
// as its distance from the origin (0,0) of the complex plane.
Complex.prototype.mag = function() {
    return Math.sqrt(this.r*this.r + this.i*this.i);
};
// Return a complex number that is the negative of this one.
Complex.prototype.neg = function() { return new Complex(-this.r, -this.i); };
// Convert a Complex object to a string in a useful way.
Complex.prototype.toString = function() {
    return "{" + this.r + "," + this.i + "}";
};
// Test whether this Complex object has the same value as another.
Complex.prototype.equals = function(that) {
    return that != null && // must be defined and non-null
        that.constructor === Complex && // and an instance of Complex
        this.r === that.r && this.i === that.i; // and have the same values.
};
```

Example

```
/*
 * Class fields (such as constants) and class methods are defined as
 * properties of the constructor. Note that class methods do not
 * generally use the this keyword: they operate only on their arguments.
 */
// Here are some class fields that hold useful predefined complex numbers.
// Their names are uppercase to indicate that they are constants.
// (In ECMAScript 5, we could actually make these properties read-only.)
Complex.ZERO = new Complex(0,0);
Complex.ONE = new Complex(1,0);
Complex.I = new Complex(0,1);
// This class method parses a string in the format returned by the toString
// instance method and returns a Complex object or throws a TypeError.
Complex.parse = function(s) {
  try { // Assume that the parsing will succeed
    var m = Complex._format.exec(s); // Regular expression magic
    return new Complex(parseFloat(m[1]), parseFloat(m[2]));
  } catch (x) { // And throw an exception if it fails
    throw new TypeError("Can't parse '" + s + "' as a complex number.");
  }
};
```


Example

```
// A "private" class field used in Complex.parse() above.  
// The underscore in its name indicates that it is intended for internal  
// use and should not be considered part of the public API of this class.  
Complex._format = /^\\{([^\,]+), ([^}]+)\\}$/;
```

```
var c = new Complex(2,3);      // Create a new object with the constructor  
var d = new Complex(c.i,c.r); // Use instance properties of c  
c.add(d).toString();         // => "{5,5}": use instance methods  
// A more complex expression that uses a class method and field  
Complex.parse(c.toString()). // Convert c to a string and back again,  
add(c.neg()).                // add its negative to it,  
equals(Complex.ZERO)        // and it will always equal zero
```

Regular expressions

- In JavaScript, regular expressions are represented by RegExp objects
- RegExp objects may be created with the RegExp() constructor but they are more often created using a special literal syntax

```
var pattern = /s$/;    // matches any string that ends with the
                        letter "s."
var pattern = new RegExp("s$");
```

- Regular-expression pattern specifications consist of a series of characters
 - Most characters, including all alphanumeric characters, simply describe characters to be matched literally
 - Other characters in regular expressions are not matched literally but have special significance

Regular expression basic syntax

Character	Description	Example
Any character except <code>[\^\$. ?*+()]</code>	All characters except the listed special characters match a single instance of themselves. { and } are literal characters, unless they're part of a valid regular expression token (e.g. the {n} quantifier).	<code>a</code> matches <code>a</code>
<code>\</code> (backslash) followed by any of <code>[\^\$. ?*+(){}]</code>	A backslash escapes special characters to suppress their special meaning.	<code>\+</code> matches <code>+</code>
<code>\Q... \E</code>	Matches the characters between <code>\Q</code> and <code>\E</code> literally, suppressing the meaning of special characters.	<code>\Q+ -*/\E</code> matches <code>+ -*/</code>
<code>\xFF</code> where FF are 2 hexadecimal digits	Matches the character with the specified ASCII/ANSI value, which depends on the code page used. Can be used in character classes.	<code>\xA9</code> matches <code>©</code> when using the Latin-1 code page.
<code>\n</code> , <code>\r</code> and <code>\t</code>	Match an LF character, CR character and a tab character respectively. Can be used in character classes.	<code>\r\n</code> matches a DOS/Windows CRLF line break.
<code>\a</code> , <code>\e</code> , <code>\f</code> and <code>\v</code>	Match a bell character (<code>\x07</code>), escape character (<code>\x1B</code>), form feed (<code>\x0C</code>) and vertical tab (<code>\x0B</code>) respectively. Can be used in character classes.	
<code>\cA</code> through <code>\cZ</code>	Match an ASCII character Control+A through Control+Z, equivalent to <code>\x01</code> through <code>\x1A</code> . Can be used in character classes.	<code>\cM\cJ</code> matches a DOS/Windows CRLF line break.

Regular expression basic syntax

Character	Description	Example
[(opening square bracket)	Starts a character class. A character class matches a single character out of all the possibilities offered by the character class. Inside a character class, different rules apply. The rules in this section are only valid inside character classes. The rules outside this section are not valid in character classes, except for a few character escapes that are indicated with "can be used inside character classes".	
Any character except <code>^-\</code> \ add that character to the possible matches for the character class.	All characters except the listed special characters.	<code>[abc]</code> matches <code>a</code> , <code>b</code> or <code>c</code>
<code>\</code> (backslash) followed by any of <code>^-\</code> \	A backslash escapes special characters to suppress their special meaning.	<code>[\\^\\]</code> matches <code>\</code> or <code>]</code>
- (hyphen) except immediately after the opening <code>[</code>	Specifies a range of characters. (Specifies a hyphen if placed immediately after the opening <code>[</code>)	<code>[a-zA-Z0-9]</code> matches any letter or digit
<code>^</code> (caret) immediately after the opening <code>[</code>	Negates the character class, causing it to match a single character <i>not</i> listed in the character class. (Specifies a caret if placed anywhere except after the opening <code>[</code>)	<code>[^a-d]</code> matches <code>x</code> (any character except a, b, c or d)

Regular expression basic syntax

Character	Description	Example
<code>\d, \w and \s</code>	Shorthand character classes matching digits, word characters (letters, digits, and underscores), and whitespace (spaces, tabs, and line breaks). Can be used inside and outside character classes.	<code>[\d\s]</code> matches a character that is a digit or whitespace
<code>\D, \W and \S</code>	Negated versions of the above. Should be used only outside character classes. (Can be used inside, but that is confusing.)	<code>\D</code> matches a character that is not a digit
<code>[\b]</code>	Inside a character class, <code>\b</code> is a backspace character.	<code>[\b\t]</code> matches a backspace or tab character

Character	Description	Example
<code>.</code> (dot)	Matches any single character except line break characters <code>\r</code> and <code>\n</code> . Most regex flavors have an option to make the dot match line break characters too.	<code>.</code> matches <code>x</code> or (almost) any other character

Regular expression basic syntax

Character	Description	Example
<code>^</code> (caret)	Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the caret match after line breaks (i.e. at the start of a line in a file) as well.	<code>^.</code> matches <code>a</code> in <code>abc\ndef</code> . Also matches <code>d</code> in "multi-line" mode.
<code>\$</code> (dollar)	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the dollar match before line breaks (i.e. at the end of a line in a file) as well. Also matches before the very last line break if the string ends with a line break.	<code>.\$</code> matches <code>f</code> in <code>abc\ndef</code> . Also matches <code>c</code> in "multi-line" mode.
<code>\A</code>	Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Never matches after line breaks.	<code>\A.</code> matches <code>a</code> in <code>abc</code>
<code>\Z</code>	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks, except for the very last line break if the string ends with a line break.	<code>.\Z</code> matches <code>f</code> in <code>abc\ndef</code>
<code>\z</code>	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks.	<code>.\z</code> matches <code>f</code> in <code>abc\ndef</code>

Regular expression basic syntax

Character	Description	Example
<code>\b</code>	Matches at the position between a word character (anything matched by <code>\w</code>) and a non-word character (anything matched by <code>[^\w]</code> or <code>\W</code>) as well as at the start and/or end of the string if the first and/or last characters in the string are word characters.	<code>.\b</code> matches <code>c</code> in <code>abc</code>
<code>\B</code>	Matches at the position between two word characters (i.e. the position between <code>\w\w</code>) as well as at the position between two non-word characters (i.e. <code>\W\W</code>).	<code>\B.\B</code> matches <code>b</code> in <code>abc</code>

Character	Description	Example
<code> </code> (pipe)	Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of options.	<code>abc def xyz</code> matches <code>abc</code> , <code>def</code> or <code>xyz</code>
<code> </code> (pipe)	The pipe has the lowest precedence of all operators. Use grouping to alternate only part of the regular expression.	<code>abc(def xyz)</code> matches <code>abcdef</code> or <code>abcxyz</code>

Regular expression basic syntax

Character	Description	Example
? (question mark)	Makes the preceding item optional. Greedy, so the optional item is included in the match if possible.	<code>abc?</code> matches <code>ab</code> or <code>abc</code>
??	Makes the preceding item optional. Lazy, so the optional item is excluded in the match if possible. This construct is often excluded from documentation because of its limited use.	<code>abc??</code> matches <code>ab</code> or <code>abc</code>
* (star)	Repeats the previous item zero or more times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is not matched at all.	<code>".*"</code> matches <code>"def" "ghi"</code> in <code>abc "def" "ghi" jkl</code>
? (lazy star)	Repeats the previous item zero or more times. Lazy, so the engine first attempts to skip the previous item, before trying permutations with ever increasing matches of the preceding item.	<code>".?"</code> matches <code>"def"</code> in <code>abc "def" "ghi" jkl</code>
+ (plus)	Repeats the previous item once or more. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only once.	<code>".+"</code> matches <code>"def" "ghi"</code> in <code>abc "def" "ghi" jkl</code>

Regular expression basic syntax

Character	Description	Example
+? (lazy plus)	Repeats the previous item once or more. Lazy, so the engine first matches the previous item only once, before trying permutations with ever increasing matches of the preceding item.	<code>".+?"</code> matches <code>"def"</code> in <code>abc "def" "ghi" jkl</code>
{n} where n is an integer >= 1	Repeats the previous item exactly n times.	<code>a{3}</code> matches <code>aaa</code>
{n,m} where n >= 0 and m >= n	Repeats the previous item between n and m times. Greedy, so repeating m times is tried before reducing the repetition to n times.	<code>a{2,4}</code> matches <code>aaaa</code> , <code>aaa</code> or <code>aa</code>
{n,m}? where n >= 0 and m >= n	Repeats the previous item between n and m times. Lazy, so repeating n times is tried before increasing the repetition to m times.	<code>a{2,4}?</code> matches <code>aa</code> , <code>aaa</code> or <code>aaaa</code>
{n,} where n >= 0	Repeats the previous item at least n times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only n times.	<code>a{2,}</code> matches <code>aaaaa</code> in <code>aaaaa</code>
{n,}? where n >= 0	Repeats the previous item n or more times. Lazy, so the engine first matches the previous item n times, before trying permutations with ever increasing matches of the preceding item.	<code>a{2,}?</code> matches <code>aa</code> in <code>aaaaa</code>

Pattern matching

- RegExp objects have methods for performing pattern matching and search-and-replace operations

```
/^HTML/           // Matches the letters H T M L at the start of a string
/[1-9][0-9]*/    // Matches a non-zero digit, followed by any # of digits
/\bjavascript\b/i // Matches "javascript" as a word, case-insensitive

var text = "testing: 1, 2, 3"; // Sample text
var pattern = /\d+/g          // Matches all instances of one or more digits
pattern.test(text)           // => true: a match exists
text.search(pattern)         // => 9: position of first match
text.match(pattern)          // => ["1", "2", "3"]: array of all matches
text.replace(pattern, "#");   // => "testing: #, #, #"
"123,456,789".split(",");    // split on ","
                             //Returns ["123","456","789"]
```

References

- D. Flanagan, "Javascript: the Definitive Guide. Sixth Edition", O'Reilly
- N. C. Zakas, "Professional JavaScript for Web Developers. Third Edition", John Wiley & Sons
- ECMAScript® 2015 Language Specification
 - <http://www.ecma-international.org/ecma-262/6.0/>
- Alex MacCaw, "Javascript Web Applications"
- Stoyan Stefanow, "Javascript patterns"
- References
 - Mozilla Developer Network <http://developer.mozilla.org>
 - JSbooks <https://jsbooks.revolunet.com>
 - <https://developers.google.com/chrome-developer-tools/>
 - [Slides embedded references]

License

- This work is licensed under the Creative Commons “Attribution-NonCommercial-ShareAlike Unported (CC BY-NC-SA 3,0)” License.
- You are free:
 - to Share - to copy, distribute and transmit the work
 - to Remix - to adapt the work
- Under the following conditions:
 - Attribution - You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
 - Noncommercial - You may not use this work for commercial purposes.
 - Share Alike - If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.
- To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/>