**Regular Expressions**

A regular expression consists of a pattern and optional flags.  
  
There are two syntaxes to create a regular expression object.  
  
1. The long syntax:  
  
regexp = new RegExp("pattern", "flags");  
  
2 The short one, using slashes "/":  
  
regexp = /pattern/; // no flags  
regexp = /pattern/gmi; // with flags g,m and i (to be covered soon)  
  
Slashes "/" tell JavaScript that we are creating a regular expression. They play the same role as quotes for strings.

**Patterns and Flags**

 A regular expression consists of a pattern and optional flags.  
  
There are two syntaxes to create a regular expression object.  
  
1. The long syntax:  
  
regexp = new RegExp("pattern", "flags");  
  
2 The short one, using slashes "/":  
  
regexp = /pattern/; // no flags  
regexp = /pattern/gmi; // with flags g,m and i (to be covered soon)  
  
Slashes "/" tell JavaScript that we are creating a regular expression. They play the same role as quotes for strings.

**Usage**

 To search inside a string, we can use method search.  
  
Here’s an example:  
  
let str = "I love JavaScript!"; // will search here  
  
let regexp = /love/;  
  
alert( str.search(regexp) ); // 2  
  
Colors  
  
From here on the color scheme is:  
  
regexp – red  
string (where we search) – blue  
result – green  
  
When to use new RegExp?  
  
Normally we use the short syntax /.../. But it does not support variable insertions ${...}.  
  
On the other hand, new RegExp allows to construct a pattern dynamically from a string, so it’s more flexible.  
  
Here’s an example of a dynamically generated regexp:  
  
let tag = prompt("Which tag you want to search?", "h2");  
  
let regexp = new RegExp([Math Processing Error]);  
  
// finds <h2> by default  
  
alert( "<h1> <h2> <h3>".search(regexp));

 Regular expressions may have flags that affect the search.  
  
There are only 6 of them in JavaScript:  
  
i  
With this flag the search is case-insensitive: no difference between A and a .  
  
g  
With this flag the search looks for all matches, without it – only the first one  
  
m  
Multiline mode.  
  
s  
“Dotall” mode, allows . to match newlines.  
  
u  
Enables full unicode support. The flag enables correct processing of surrogate pairs.  
  
y  
Sticky mode

**Flags**

 Regular expressions may have flags that affect the search.  
  
There are only 6 of them in JavaScript:  
  
i  
With this flag the search is case-insensitive: no difference between A and a .  
  
g  
With this flag the search looks for all matches, without it – only the first one  
  
m  
Multiline mode.  
  
s  
“Dotall” mode, allows . to match newlines.  
  
u  
Enables full unicode support. The flag enables correct processing of surrogate pairs.  
  
y  
Sticky mode.

**Methods of RegExp and String**

 To search for all matches:  
  
Use regexp g flag and:  
  
Get a flat array of matches – str.match(reg)  
Get an array or matches with details – str.matchAll(reg).  
To search for the first match only:  
  
Get the full first match – str.match(reg) (without g flag).  
Get the string position of the first match – str.search(reg).  
Check if there’s a match – regexp.test(str).  
Find the match from the given position – regexp.exec(str) (set regexp.lastIndex to position).  
To replace all matches:  
  
Replace with another string or a function result – str.replace(reg, str|func)  
To split the string by a separator:  
  
str.split(str|reg)

**Character classes**

 A character class is a special notation that matches any symbol from a certain set.  
  
For the start, let’s explore a “digit” class. It’s written as \d. We put it in the pattern, that means “any single digit”.  
  
For instance, the let’s find the first digit in the phone number:  
  
let str = "+7(903)-123-45-67";  
  
let reg = /\d/;  
  
alert( str.match(reg) ); // 7  
  
Without the flag g, the regular expression only looks for the first match, that is the first digit \d.  
  
Let’s add the g flag to find all digits:  
  
let str = "+7(903)-123-45-67";  
  
let reg = /\d/g;  
  
alert( str.match(reg) ); // array of matches: 7,9,0,3,1,2,3,4,5,6,7  
  
alert( str.match(reg).join('') ); // 79035419441  
  
That was a character class for digits. There are other character classes as well.  
  
Most used are:  
  
\d (“d” is from “digit”)  
  
A digit: a character from 0 to 9.  
  
\s (“s” is from “space”)  
  
A space symbol: that includes spaces, tabs, newlines.

**Word boundary \b**

 A word boundary \b – is a special character class.  
  
It does not denote a character, but rather a boundary between characters.  
  
For instance, \bJava\b matches Java in the string Hello, Java!, but not in the script Hello, JavaScript!.  
  
alert( "Hello, Java!".match(/\bJava\b/) ); // Java  
  
alert( "Hello, JavaScript!".match(/\bJava\b/) ); // null  
  
The boundary has “zero width” in a sense that usually a character class means a character in the result (like a wordly character or a digit), but not in this case.  
  
The boundary is a test.  
  
When regular expression engine is doing the search, it’s moving along the string in an attempt to find the match. At each string position it tries to find the pattern.

**Inverse Classes**

 For every character class there exists an “inverse class”, denoted with the same letter, but uppercased.  
  
The “reverse” means that it matches all other characters, for instance:  
  
\D  
Non-digit: any character except \d, for instance a letter.  
  
\S  
Non-space: any character except \s, for instance a letter.  
  
\W  
Non-wordly character: anything but \w.  
  
\B  
Non-boundary: a test reverse to \b.  
  
In the beginning of the chapter we saw how to get all digits from the phone +7(903)-123-45-67.  
  
One way was to match all digits and join them:  
  
let str = "+7(903)-123-45-67";  
  
alert( str.match(/\d/g).join('') ); // 79031234567  
  
An alternative, shorter way is to find non-digits \D and remove them from the string:  
  
let str = "+7(903)-123-45-67";  
  
alert( str.replace(/\D/g, "") ); // 79031234567

**Spaces are regular characters**

 Usually we pay little attention to spaces. For us strings 1-5 and 1 - 5 are nearly identical.  
But if a regexp doesn’t take spaces into account, it may fail to work.  
Let’s try to find digits separated by a dash:  
alert( "1 - 5".match(/\d-\d/) ); // null, no match!  
Here we fix it by adding spaces into the regexp \d - \d:  
alert( "1 - 5".match(/\d - \d/) ); // 1 - 5, now it works  
A space is a character. Equal in importance with any other character.  
Of course, spaces in a regexp are needed only if we look for them. Extra spaces (just like any other extra characters) may prevent a match:  
alert( "1-5".match(/\d - \d/) ); // null, because the string 1-5 has no spaces  
In other words, in a regular expression all characters matter, spaces too.

**A dot is any character**

 The dot "." is a special character class that matches “any character except a newline”.  
  
For instance:  
  
alert( "Z".match(/./) ); // Z  
  
Or in the middle of a regexp:  
  
let reg = /CS.4/;  
  
alert( "CSS4".match(reg) ); // CSS4  
  
alert( "CS-4".match(reg) ); // CS-4  
  
alert( "CS 4".match(reg) ); // CS 4 (space is also a character)  
  
Please note that the dot means “any character”, but not the “absense of a character”. There must be a character to match it:  
  
alert( "CS4".match(/CS.4/) ); // null, no match because there's no character for the dot

**The dotall “s” flag**

 Usually a dot doesn’t match a newline character.  
  
For instance, A.B matches A, and then B with any character between them, except a newline.  
  
This doesn’t match:  
  
alert( "A\nB".match(/A.B/) ); // null (no match)  
  
// a space character would match, or a letter, but not \n  
  
Sometimes it’s inconvenient, we really want “any character”, newline included.  
  
That’s what s flag does. If a regexp has it, then the dot "." match literally any character:  
  
alert( "A\nB".match(/A.B/s) ); // A\nB (match!)

**Escaping, Special character**

 As we’ve seen, a backslash "\" is used to denote character classes. So it’s a special character in regexps (just like in a regular string).  
  
There are other special characters as well, that have special meaning in a regexp. They are used to do more powerful searches. Here’s a full list of them: [ \ ^ $ . | ? \* + ( ).  
  
Don’t try to remember the list – soon we’ll deal with each of them separately and you’ll know them by heart automatically.  
  
Escaping  
Let’s say we want to find a dot literally. Not “any character”, but just a dot.  
  
To use a special character as a regular one, prepend it with a backslash: \..  
  
That’s also called “escaping a character”.