**STRINGS**

Now let's look at the work with string objects.  
  
You can **create a String object** in three ways:

* *myString = 'my string'*                // create using a string literal
* *myString = new String (object)* // create an object, now such syntax is not used
* *myString = String (object)*

Typically, such objects are created implicitly, by a string literal: *var myString = 'Some text;'*  
(a string literal is any sequence of characters enclosed in any quotes)  
In a string literal, you can use any characters as well as escape sequences if you need to specify a character that, for example, can not be typed on the keyboard. It looks like this: *var myString = '\u1234';* where 1234 is the number of the character in the unicode table.  
  
**An important point!** When you create an object with a literal (text in quotation marks), you get the type of the "string" object, and when you create with the keyword "new" you get the type "Object", this object can directly assign additional properties and methods.

**Properties of the "String" object**  
  
Although, of course, the word "properties" sounds too loud - the standard property of it is one - **length**. It is denoted by the word ".**length**" and returns the number of characters in the string.

var cat = "Cat";

console.log (cat.length);

In this example, the value of the string length is 3.  
  
The value of the length of the empty string, of course, is 0.

But the methods of the String object are more than enough. Let's look at them in groups:  
  
**Methods for accessing the symbols of the String object**:  
There are two ways to access the symbols of a String object.  
  
Basic is access through the **.charAt method**  
For example:

var cat = "Cat";

console.log (cat.charAt (1));

This code will output to the console a character with index 1 in the string - 'a' (remember that the numbering always starts with 0)

We also need to consider another method, related to the previous one.  
**charCodeAt** (index) - returns the character code at index  
The call to it looks similar to the previous method, but on the output we have not the character itself, but its unicode.  
  
The second way to access characters is to **access the string as an array** (we'll look at the arrays in the next lesson):

var cat = "Cat";

console.log (cat [1]);

One **important point to note**, in contrast to most non-scripting languages ​​in JavaScript, once a created string can not be changed, it can only be read. To change the line, you need to reassign it completely:

var cat = "Cat";

cat = cat.charAt (2) + cat.charAt (1) + cat.charAt (0);

The code in this example will turn the contents of the string "backwards".

**Merge and split strings**  
  
**concat**(string1, string2, ..., stringN) - concatenates strings  
**split**('separator') - splits strings into an array of substrings using the specified delimiter  
  
**Сoncat** is the union of strings.  
  
Example:

var a = "first line ";

var b = "second line ";

var c = "third line";

result = a.concat(b, c);

As a result, the value of the result variable becomes "first line second line third line"  
  
  
**Split**- splits a string into an array of values ​​by separator.

var myString = "cat,dog,bird";

var myArray = myString.split (',');

As a result, myArray array has three elements - ["cat", "dog", "bird"]  
The string is delimited by the comma, which, when divided into the final array, does not fall inside of array.  
If the delimiter is not found, or if we did not specify it, then there will be only one element in the result array - the whole line.  
  
The split method has one more optional parameter - the maximum number of elements in the array. All elements larger than the specified number will be discarded and will not appear in the final array.

var myString = "cat:dog:bird";

var myArray = myString.split (":", 2);

As a result, in myArray array there will be only two elements - ["cat", "dog"]

**Convert register**  
  
**toLowerCase**() - returns new string with all letters converted to lowercase  
**toUpperCase**() - returns new string with all letters converted to uppercase  
**toLocaleLowerCase** () - returns new string with all letters converted to lowercase  
**toLocaleUpperCase**() - returns new string with all letters converted to uppercase  
(The last two methods have some special features for languages ​​whose rules contradict the Unicode conventions.)  
  
Since the syntax of all these methods is the same, consider only one example.  
  
var catName = "Kitty";

console.log (catName.toUpperCase ());

As a result of executing this code, the contents of the line will be displayed in the console with capital letters - "KITTY"

**Return part of a string**  
  
**substring**(indexA [, indexB]) - returns part of the string starting at position indexA, but not including indexB  
**slice**(indexA [, indexB]) - returns part of the string starting at position indexA, but not including indexB  
**substr**(indexA [, length]) - returns a portion of the string starting at position indexA, in the number of length  
**replace**(regexp, newSubString | function) - returns a new string after the replacements specified in regexp, or the function that returns it. Regular expressions - RegExp, we will discuss in detail in the third module of the course.  
  
For example, we will see operation with the method **substring**(indexA [, indexB])  
  
var myString = "This is just a string!";

console.log (myString.substring (4, 17));

As a result, the text will appear in the console from 4 to 16 positions - "is just a st"

**Find a substring in a string**  
  
**indexOf** (searchValue [, fromIndex]) - searches for a searchValue substring starting from the fromIndex position  
**lastIndexOf**(searchValue [, fromIndex]) - searches for the last substring of searchValue starting from fromIndex  
**search**(regExp) - checks whether there is a match with the argument, the result is true or false  
**match**(regExp) - returns an array of all matches with an argument  
  
First, lets work with **indexOf**(searchValue [, fromIndex])  
var myString = "This is just a test string!";

console.log (myString.indexOf ("is", 0));

In this case, the console will display the position number with which the first found substring "is" - 2 begins.  
"Th**is** is just a test string!";  
  
  
However, if we specify the method to look for a substring not from the very beginning, but, for example, from the 3rd position:

console.log (myString.indexOf ("is", 3));  
Then the result will be "5" - the position with which the second occurrence of the substring "is" begins.  
"This **is** just a test string!";  
  
The **lastIndexOf** (searchValue [, fromIndex]) method works similarly, but returns the number from which the last occurrence of the searched substring begins. **fromIndex**denotes the place from which to start the search towards the beginning of the line, its default value is the length of the string. In the example:  
var myString = "This is just a test string!";

console.log (myString.lastIndexOf ("is"));

the number 5 - the beginning of the last occurrence of the substring "is" will be displayed in the console.  
  
The methods **search**(regExp) and **match** (regExp) here we indicated for completeness of the enumeration, their work will be considered in the next module, after learning the work with regular expressions.

**Type conversion**  
  
**fromCharCode** (num1, num2, ..., numN) - creates an elementary string from the values of Unicode characters  
For example, the expression *myString = String.fromCharCode (65, 66, 67);* assigns the variable myString the value "ABC".  
  
  
**toString**() - returns an elementary string instead of a String object  
**valueOf**() - returns an elementary string instead of a String object, equivalent to toString ()  
As a result of the following code, an elementary value will be output to the console.

myString = new String ("Hello world");

console.log (myString.valueOf ());

**ARRAYS**

First of all, let's consider in general - **what is an array?**  
  
**An array** is a set of values ​​accessed by their ordinal number. The value in the array is called the "**array element**" and refers to them by their ordinal number, called the "**index**".  
  
**Arrays in JavaScript are untyped -** allow elements of different types to be stored inside a single array. It can be not only elementary types - strings, numbers or symbols, but also objects and arrays, and even arrays of arrays.  
  
**Arrays in JavaScript start index counting from zero**, for indexes 32-bit values ​​are used.  
  
**Arrays in JavaScript are dynamic**, meaning they can grow or shrink in size. Announce the pre-fixed sizes when creating them, and also reallocate memory when resizing, as is done in some other languages ​​- it is not necessary.  
  
**Arrays in JavaScript are objects!**

**To create an array,** you can use three methods:  
  
**The usual way** is to create an array by calling the Array() constructor:

var myArray = new Array (); // declaration of the array

myArray [0] = "Ivanov"; // adding an element

myArray [1] = "Petrov";

myArray [2] = "Sidorov";

myArray [3] = "Kuznetsov";

You can also call a constructor with a numeric argument indicating the number of elements.

var myArray = new Array (10); // create an array of 10 elements

The same operation can be performed in a shorter way:

var myArray = new Array ("Ivanov", "Petrov", "Sidorov", "Kuznetsov");

And the third option is literal:

var myArray = ["Ivanov", "Petrov", "Sidorov", "Kuznetsov"];

**Reading and writing, adding and removing elements of an array.**  
  
To access the elements of the array, use the [ ] operator (square brackets).  
To the left of the parentheses there must be a reference to the array itself, inside the parentheses there must be an expression that returns a non-negative integer value.  
  
How to **write to the array** we already saw in the previous step:

myArray [0] = "Ivanov"; // write the 0th element to the array myArray

**Reading from array**is the same:

x = myArray [0]; // As a result, the value of the first cell of the array in the variable x is the string value "Ivanov".

**Adding the elements of an array.**  
  
As we have seen, the easiest way to **add an element to an array** is simply to assign a value to the new indexes. For example :

myArray [4] = "Petrov";

But we also have a special method - **push** ().  
For example:

myArray.push ("Sidorov");

Add a value to the end of the array, and the command

myArray.push ("Sidorov", 2);

adds two elements at once to the end of the array - the string "Sidorov" and the number "2". The value returned by the method will contain the length (number of elements) of the resulting array.  
  
**Add elements to the end of the array** can also be simply changing the property of the array length:

myArray.length = 5;

In this case, the number of elements in the array will increase, at the end of the array empty elements will be added.  
  
**Add elements to the beginning of the array** using the **unshift**() method. For example:

myArray.unshift ("Sergeev", "Dmitriev")

will add two elements with the values ​​"Sergeev" and "Dmitriev" to the beginning of our array, the remaining elements will be moved to positions with higher indices. The unshift () method also returns the length of the resulting array. Thus the expression

x = myArray.unshift ("Sergeev", "Dmitriev")

puts two new elements at the beginning of the array myArray and assigns the value of x to 7 (Because the previous command, we set the size of the array - 5)

**Delete the elements of the array.**  
  
You can **delete an element** of an array using the **delete**operator, as a normal property. This operator was considered in the lesson about objects. As a result of the action ***delete****myArray [2];* the value of the third element of the array will be set to **undefined**, but the number of elements in the array does not change, all other elements remain in their places.  
  
To **delete elements at the end** of the array, you can perfectly use the array length setting command - changing the **length**property, for example - *myArray.length = 4;*  
  
You can also **delete an element at the end** of the array using the **pop**() method (opposite to the push method), which reduces the length of the array by 1 and returns the value of the deleted element.  
  
Also there is a method **shift**() (opposite to the unshift method) which **deletes the element at the beginning of the array** and shifts all the elements 1 position to the beginning.

**Some other methods of the Array class.**  
  
**Array.join()** - turns all the elements of the array into strings, combines them and returns the resulting string.  
For example, if you look at the example from the previous step - an array of surnames, the command *myString = myArray.join();* will put the value "Ivanov, Petrov, Sidorov" in the string variable myString. By default, commas are used for the separation.  
In an optional argument, you can specify a substring to be used to separate the values. For example: the command

*myString = myArray.join("\_");*

will put in the string variable myString the value "Ivanov\_Petrov\_Sidov"

**Array.reverse()** - changes the order of the elements in the array to the inverse and returns the already rearranged array.  
For example, if you consider the previously declared array:

var myArray = new Array ("Ivanov", "Petrov", "Sidorov", "Kuznetsov");

then the command

console.log (myArray.reverse ());

will print to the console values ​​["Kuznetsov", "Sidorov", "Petrov", "Ivanov"].  
  
**Array.sort()** - sorts the elements in the source array and returns the sorted array. If the method is used without an argument (sorting function), the result will be sorted alphabetically (in the order of the characters in Unicode), by converting everything that is possible to a string type. All that impossible convert to string - is placed at the end of the array. For example, using this method on an array from the previous example

var myArray = new Array ("Ivanov", "Petrov", "Sidorov", "Kuznetsov");

will print an array of this kind to the console: [Ivanov, Kuznetsov, Petrov, Sidorov].  
  
As an argument to the **sort**() method, you can pass a comparison function. If the result returned by the comparison function is less than 0, then the sorting will put a before b, and vice versa. For example, to display values ​​sorted in reverse alphabetical order, we can use this command:

console.log (myArray.sort (function (a, b) {return b - a;}));

However, it should be noted that such a comparison function will only work if the elements of our array consist of digits. If we want to change the direction of sorting for string elements, the comparison function should look slightly different, for example like this:

console.log (myArray.sort (function (a, b) {return b.localeCompare (a);}));

In this case, we used the **localeCompare**() method of the **String** object, which allows us to compare objects of type String or string literals.  
  
**Array.concat**() - returns (without changing the source) a new array with the addition of the elements passed to the method as an argument. For example,

console.log (myArray.concat("Smith"));

will print to the console values ​​["Ivanov", "Kuznetsov", "Petrov", "Sidorov", "Smith"].

**Array.slice**() - returns the subarray from the array, from the first to the second arguments (but not including it).

For example

console.log (myArray.slice (1,3));

will print to the console output values ​​["Kuznetsov", "Petrov"], i.e. elements with indices 1 and 2.

The next method will be considered a separate step because of its universality and convenience.  
  
**Array.splice**() is a universal function that allows you to remove elements from an array and insert new ones.  
  
The following data can be passed as an argument to this method:  
**1 argument** is the position of the element from which the method action begins  
**2 argument** - the number of elements to be deleted, starting from the starting position specified in the previous argument. (If the second value is not specified, all elements will be deleted, starting with the first argument).  
**3 argument** and subsequent (any number) are the elements of the array that will be added starting from the position specified in the first argument.  
  
An important point is that the method changes the original array, but returns an array of deleted elements. If no items are deleted, an empty value is returned.  
  
As an example, let's consider the work of this method on an array

var myArray = ["Ivanov", "Petrov", "Sidorov", "Kuznetsov"];

We obtain the following results:

// returns ["Sidorov", "Kuznetsov"], the original array will be equal to ["Ivanov", "Petrov"]

myArray.splice (2, 2);

// return ["Petrov", "Sidorov", "Kuznetsov"], the original array will turn into ["Ivanov"]

myArray.splice (1);

// returns an empty value [], the element ["Ivanov", "Petrov", "Smith", "Sidorov", "Kuznetsov"] will be added to myArray

myArray.splice (2, 0, "Smith");

**DATE AND TIME**

To work with dates and time in JavaScript, use the **Date objec**t.  
  
The **Date object** is created using the **Date()** constructor. There are several ways to create an object of this type:

var x = new Date () // The object will get the current date and time

var x = new Date (milliseconds) // The number of milliseconds counting from January 1, 1970

var x = new Date (stringData)

var x = new Date (year, month, day [, hours, minutes, seconds, milliseconds])

Note that the parameters specified in square brackets are optional.  
  
**year** - format YYYY  
**month** - from 0 to 11  
**day**-from 1 to 31  
**hours**- from 0 to 23  
**minutes**-from 0 to 59  
**seconds**- from 0 to 59  
**milliseconds** - from 0 to 999  
  
Let's look at some examples of creating a Date object and show in the comments - which will be displayed on the page when using the **document.write (myDate)** command:  
(the document.write () command has not been considered in detail, at this stage it is sufficient to understand that it outputs what is passed to it in an HTML document.)

var myDate = new Date () // Mon May 15 2017 19:20:25 GMT + 0300 (RTZ 2 (winter))

var myDate = new Date ("December 14, 1975 12:10:00") // Sun Dec 14 1975 12:10:00 GMT + 0300 (RTZ 2 (winter))

var myDate = new Date (1989, 6, 14) // Fri Jul 14 1989 00:00:00 GMT + 0400 (RTZ 2 (summer))

var myDate = new Date (1998, 6, 14, 11, 20, 00) // Tue Jul 14 1998 11:20:00 GMT + 0400 (RTZ 2 (summer))

The date and time values ​​can be easily managed using the methods available for the Date object. (These methods will be discussed in more detail in the next steps.)  
  
In the following example, we will set a date in the Date object:

var myDate = new Date (); // Declare a Date variable

myDate.setFullYear (2017, 4, 22); // Assign to it the date value - May 22, 2017

And now we'll try to move the date 10 days forward:

var myDate = new Date (); // Declare a Date variable

myDate.setFullYear (2017, 4, 22); // Assign to it the date value - May 22, 2017

myDate.setDate (myDate.getDate () + 10); // Set the new date value, get the previous value and add 10.

Pay attention - if the days increase will change the month and year, this will be done automatically, by the method setDate ().

Objects of type Date **can be compared** in the same way as others.  
  
In the example below, we compare the current date with the date of January 1, 2018 and if it coincides - print the greeting.

var currentDate = new Date (); // Declare a variable for the current date

var nextNewYear = new Date (); // Declare a variable for the New Year's date

nextNewYear.setFullYear (2018, 0, 1); // Record the date value for the New Year - January 1, 2018

if (currentDate == nextNewYear) {

  alert ("Congratulations on the New, 2018th Year! Ur! !!!");

}

Let's look at **the basic methods of the Date object**.  
  
**Retrieving data** from a Date object.  
  
**getDay**() - returns the day of the week from 0 to 6, 0 - Sunday, 1 - Monday, etc.  
**getTimeZoneOffset**() - returns the offset of the time zone relative to UTC, in minutes with the opposite sign.  
**getYear**() - returns the value of the year minus 1900, to use is not recommended.  
**getFullYear**() - returns the value of the year.  
**getMonth**() - returns the month, from 0 to 11  
**getDate**() - returns the day of the month from 1 to 31  
**getHours**() - returns the hour, from 0 to 23  
**getMinutes**() - returns the number of minutes, from 0 to 59  
**getSeconds**() - returns the number of seconds from 0 to 59  
**getMilliseconds**() - returns the number of milliseconds, from 0 to 999  
**getTime**() - returns the number of milliseconds that have elapsed since midnight on January 1, 1970 GMT.

**Write data** to a Date object.  
  
**setYear**() - sets the value of the year minus 1900, not recommended for usage.  
**setFullYear**() - sets the value of the year.  
**setMonth**() - sets the month, from 0 to 11  
**setDate**() - sets the day of the month from 1 to 31  
**setHours**() - sets the hour, from 0 to 23  
**setMinutes**() - sets the number of minutes from 0 to 59  
**setSeconds**() - sets the number of seconds from 0 to 59  
**setMilliseconds**() - sets the number of milliseconds, from 0 to 999  
**setTime**() - sets the number of milliseconds that have elapsed since midnight on January 1, 1970 GMT.

**Converting Data** of a Date Object  
  
**Date.parse**() - converts a string with a date, for example "Jul 05, 2017" and returns the number of milliseconds that passed from midnight on January 1, 1970. If the string could not be converted, it returns NaN.

var myDate = new Date ();

document.write (myDate.setTime (Date.parse ("22 May 2017 11:11"))); // Display the "1495613460000"

document.write (myDate); // Display the following: Mon May 22 2017 11:11:00 GMT + 0300 (RTZ 2 (winter))

**toLocaleString**() - Returns an object of type String containing a date in long format, for example "10 January 2017 12:26:01" in accordance with the regional settings of the operating system in which the script is run. Outwardly, the difference will be, for example, between countries where the 12-hour time designation is common, in comparison with the 24-hour time.

**toLocaleTimeString**() - converts time data to a string, using the formatting settings of the operating system in which the script is executed.

**toLocaleDateString**() - performs a transformation similar to the previous one, but with a date.

**MATHEMATICAL OPERATIONS**

In JavaScript, **mathematical operations** are represented by methods of the standard **Math object**.  
  
The **Math**object has **methods**, each of which performs a certain mathematical function (for example, square root, exponentiation, sine, etc.) and **properties**that are actually constants - the number Pi, the number E, the natural logarithm 10 and others .  
  
Unlike other global objects, **Math is not a constructor** - properties and methods of the **Math**object are **static**. This means that we can access its methods and properties directly, without creating an instance of the object. For example, you can refer to the constant pi as **Math.PI** or call the sqrt (square root) function as **Math.sqrt(x**).

First, let's look at the **constants of the Math object**.  
  
**Math.E** is the number e, the base of the natural logarithm, the constant of Euler (Neper), about 2.718 ...  
**Math.PI** is the Pi number, approximately equal, as is known, 3.1415926 ...  
**Math.SQRT2** is the square root of 2, the approximate value is 1.414  
**Math.SQRT1\_2** is the square root of 1/2, the approximate value is 0.707  
**Math.LN2** is the natural logarithm 2, the approximate value is 0.693  
**Math.LN10** - natural logarithm 10, the approximate value of 2.302  
**Math.LOG2E** is the logarithm E of base 2, an approximate value of 1.442  
**Math.LOG10E** is the logarithm E of base 10, an approximate value of 0.434  
  
As an example, consider a function that receives the radius of a circle and returns its length:

function circumference (radius) {

  return radius \* Math.PI \* 2;

}

In this example, we use access to the constant Math.PI to get the value of the Pi value.

Now let's look at the **methods of the Math object**.  
  
We divide them into groups and first consider **trigonometric functions**:  
  
  
**Math.sin**(x) - returns the sine of the argument (in radians), from -1 to 1.  
**Math.cos**(x) - returns the cosine of the argument (in radians), from -1 to 1.  
**Math.tan**(x) - returns the numerical value for the tangent of the angle in radians.  
**Math.asin**(x) - returns the value (in radians) of the arcsine for the argument, which is specified from -1 to 1  
**Math.acos**(x) - returns the value (in radians) of the arccosine for the argument, which is specified from -1 to 1  
**Math.atan**(x) - returns the value of the arc tangent (from the range -π2π2 до π2π2) for the argument  
**Math.atan2**(x, y) - function is called the arc tangent of two variables. It returns a numeric value between -ππ and ππ, and represents the angle between the positive X-axis and the x, y point.

**Conversion and comparison functions:**  
  
**Math.min** ([Value1 [, value2 [, ...]]]) - Returns the minimum value of the arguments.  
**Math.max** ([Value1 [, value2 [, ...]]]) - Returns the maximum value of the arguments.  
**Math.floor** (x) - Returns the largest integer less than or equal to the argument  
**Math.ceil** (x) - returns the smallest integer greater than or equal to the argument  
**Math.abs** (x) - returns the absolute value of a number, it is also called a "module"  
**Math.round** (x) - rounds the number according to the rules of mathematics

**Функции преобразования и сравнения:**

**Math.min ([Значение1[,значение2[, ...]]])** - возвращает минимальное значение из аргументов.  
**Math.max ([Значение1[,значение2[, ...]]])** - возвращает максимальное значение из аргументов.    
**Math.floor (x)** - округление до ближайшего целого в меньшую сторону  
**Math.ceil (x)** - округление до ближайшего целого в большую сторону  
**Math.abs(x)** - возвращает абсолютное значение числа, его еще называют "модуль"  
**Math.round(x)** - округляет число по правилам математики

**Обратите внимание!** Запись аргументов в квадратных скобках означает, что аргументы - необязательные. Это означает, что запись Math.min([Значение1[,значение2[, ...]]]). Может использоваться как Math.min() или Math.min(Значение1), или Math.min(Значение1, значение2) и т.п.

**Calculation functions**:  
  
**Math.sqrt**(x) - returns the square root of the argument  
**Math.pow**(base, exponent) - raises the number "base" to the power of "exponent"  
**Math.log**(x) - calculates the natural (on the basis of e) logarithm of the number  
**Math.exp**(x) - calculates the exponent - the value of the number e in the power of the argument "x"  
**Math.random**() - returns a random number from 0 (inclusive) to 1

ets start this topic with the concept of "**Exception**". Let's consider some terms:  
  
**Exception**is an event that signals an abnormal situation or error.  
**To excite, (create, throw) an exception** - to signal about any mistake or an exceptional situation.  
**To catch an exception** is to take action to handle the exception and restore the normal functionality of the code.  
  
The exception is raised by the "**throw**" operator, the interception - by the "**catch**" command (more precisely, the "try-catch-finally" statement).

**EXCEPTIONS**

Lets start this topic with the concept of "**Exception**". Let's consider some terms:  
  
**Exception**is an event that signals an abnormal situation or error.  
**To excite, (create, throw) an exception** - to signal about any mistake or an exceptional situation.  
**To catch an exception** is to take action to handle the exception and restore the normal functionality of the code.  
  
The exception is raised by the "**throw**" operator, the interception - by the "**catch**" command (more precisely, the "try-catch-finally" statement).

The syntax of the **throw**statement is as follows:

throw expression;

"*Expression*" (or the result of its calculation) can represent almost any type of data: a string, a number, a Boolean value, an object. For example, a number that represents the error code, a string containing the error text.  
  
For example, the following example demonstrates checking the input parameter in a recursive function when evaluating:

function testFactorial (inputData) {

  if (inputData <0) // Check if the number is positive

throw "The number must not be less than zero"; // If negative, throw an exception

  return (inputData - 1)? (inputData \* testFactorial (inputData - 1)): inputData;

}

In this example, if a number less than 0 is applied to the input of the function, an error will be generated with the text "The number should not be less than zero" and the function will exit.  
If the input is correct, the factorial of the number fed to the input will be calculated.

Once an exception is created, the JavaScript interpreter breaks the normal execution of the code and begins to search for an exception handler - **try / catch / finally** constructs.  
  
The syntax of this construct is as follows:

try {

  // code that you need to "try"

  // in this code an exception can be thrown

} catch (exception\_variable) {

  // at this point the code is written, which is executed only in case of detection

  // exceptions in the previous block "try"

  // if an exception occurs, the exception\_variable variable will be passed

  // code of the error that occurred, for example, the argument to the throw statement

} finally {

  // The code in this block will always be executed, regardless of the result of the end of the try block:

  // and when completed without error, and when completed with an error, and when completed on any branch statement

  // (break, continue, return)

}

The **catch**and **finally**blocks are not strictly mandatory, but at least one of them must be present in the construction.

Let's look at an example using the testFactorial() function, created during this lesson.

// Declaration of the function

function testFactorial (inputData) {

  if (inputData <0) // Check if the number is positive

    throw "The number must not be less than zero"; // If negative, throw an exception

  return (inputData - 1)? (inputData \* testFactorial (inputData - 1)): inputData;

}

// Initiate the variable for the input parameter, set it manually in this example

var myNumber = -5;

// start of error handling design

try {

  document.write (testFactorial (myNumber)); // attempt to call a function

} catch (ex) {

  document.write (ex); // if an exception occurs in the function, the text will be displayed,

                      // which we used in the parameter of the throw statement

}

In this example, we did not use the finally block, it does not need us here. If you remember, at least one of the finally and catch blocks can be used in the construction, both are optional.

A very important point is that the **try / catch / finally** constructs can be nested multiple times. In such a situation, we often need to check - what kind of exception was catched, and if this is not the exception that we were waiting for and ready to process, "not ours," then we need to send it on. We cannot use option "do not intercept", but we can intercept, check and if it's not what we need - to generate new and send on.  
  
Let's see how we would implement the previous example with this refinement:

// Declaration of the function

function testFactorial (inputData) {

  if (inputData <0) // Check if the number is positive

    throw "The number must not be less than zero"; // If negative, throw an exception

  return (inputData - 1)? (inputData \* testFactorial (inputData - 1)): inputData;

}

// Initiate the variable for the input parameter, set it manually in this example

var myNumber = -5;

// start of error handling design

try {

  document.write (testFactorial (myNumber)); // attempt to call a function

} catch (ex) {

  if (ex != "The number should not be less than zero") // Check - if the exception is not ours

    throw (ex); // then "run" it on

  document.write (ex); // output the error if the exception is "our"

}

In this example, before we display the error on the screen, we check whether it is the error that was generated by the throw statement in our function. If not, then generate it anew - as if we send a walk further, in search of its catch.

Also we need to consider the standard Error object, which is used when using exceptions.  
  
In general, almost any object can be thrown by the throw command, however, when standard errors occur, the object of the **Error**class or its subclasses is thrown out.  
  
The syntax that we will look at is as follows:

new Error (message);

This command creates a new instance of the **Error** object, and the text "*message*" is written to the property "message" of the object.  
  
The simplest example of calling the Error constructor is as follows.

try {

  throw new Error ('Something went wrong!');

} catch (e) {

  console.log (e.name + ':' + e.message);

}

In this example, we created an **Error**object with the name *Error*and the text of the error message "*Something went wrong!*".

**REGULAR EXPRESSIONS**

What are **regular expressions**?  
  
Regular expressions (**RegExp - regular expressions**) is an object that can describe a pattern of characters.  
  
For example, if you are looking for a substring in a string, you can describe a pattern of what you are looking for.  
A simple template can consist of even one character, large and complex templates can be used for parsing, checking the format of entered data, replacing and other various purposes.  
  
  
The content of the regular expression pattern consists of a sequence of characters. Most of them (all letters and numbers) describe themselves - they directly indicate their presence. For example, a regular expression

/hello/

matches all the strings that have the word "hello". Other symbols do not designate themselves, but have some modifying value.  
  
For example, expression

/hello$/

will correspond to the strings that FINISH on the word "hello". This provides the "$" metacharacter, indicating the end of the string.

**Creating RegExp objects**  
  
RegExp objects can be created using the **RegExp() constructor**, or by **using literals**. But if, for example, string literals are specified in the form of characters enclosed in quotation marks, the regular expression literals are specified by a pair of slash characters "/". For example, like this:

var myPattern = /q$/; // Create a regular expression with a literal

In this example, we created a new object of type RegExp with the literal and assigned it to the variable myPattern. This pattern matches any string ending with the character q.  
  
To create the same object using the constructor, we need to write the following expression:

var myPattern = new RegExp ("q$"); // Create a regular expression using the constructor

Now let's look at what you can write regular expressions from.  
  
The first group is certainly **symbols**. As we have already said, all alphabetic and numeric symbols denote themselves. You can also enter some non-alphabetic characters using sequences that begin with a backslash.

**Numbers and letters** correspond to themselves  
**\0** - NUL (Corresponds \ u0000 to Unicode)  
**\t** - Tab (\ u0009)  
**\n** - Line feed (\ u000A)  
**\v** - Vertical tab (\ u000B)  
**\f** - Translate page (\ u000C)  
**\r** - Carriage return (\ u000D)  
**\xnn** - The character from the Latin set, given by the hexadecimal number nn  
**\unnnn - The Unicode character specified by the hexadecimal number nnnn**  
**\cX** - Control character "X", for example \ cJ is equivalent to \ n

The following symbols are also used in regular expressions:

^ $ . \* + ? = ! : | \ / ( ) [ ] { }

In the next steps, we'll take a closer look at their meaning and application in combination with other symbols, but now we need to remember that to determine the meaning of these symbols literally, i.e. "themselves", it is necessary to put a backslash symbol before them.

For example, if you want to write a regular expression that will contain a backslash character, you must put this character in the expression, preceded by the same backslash character. As a result, this regular expression will look like this:

/\\/

Individual symbols can be combined into **classes**. This is indicated by a set of characters enclosed in square brackets. For example, a regular expression

/[0123456789]/

corresponds to any number.

Alternatively, you can specify the "^" sign before the character set, to define a regular expression that matches any character, EXCEPT those specified in parentheses - a class with negation. For example, expression

/[^0123456789]/

will match any character EXCEPT digits.

Also in classes, you can specify a range using the hyphen "-" to avoid listing all the characters. For example, all the numbers can be expressed by the following expression:

/[0-9]/

Some classes from character sets are so often used that they have been assigned special notation:

**[...]** - any of the characters indicated in parentheses  
**[^ ...]** - any other than the characters indicated in parentheses  
**. (dot)** - any character except line feed or other line separator

**\ w**- is equivalent to [a-zA-Z0-9\_] (Any ASCII text character)  
**\W** - is equivalent to [^ a-zA-Z0-9\_] (Any character other than ASCII text characters)  
**\s** - any whitespace character from Unicode  
**\S** - any non-whitespace character from Unicode  
**\d** - is equivalent to [0-9] (any ASCII digits)  
**\D** - is equivalent to [^ 0-9] (all characters except ASCII digits)  
**[\b]** - the designation of the symbol "slaughter"

Sequences of such control characters can also be combined into a class, for example, a regular expression

**/[\w\d]/**

matches any ASCII character with letters or numbers.

The templates described above can be used not only to describe single character combinations, but also for arbitrarily multiple repetitions. This is called "**quantification**".

For quantification in regular expressions, there is a set of special combinations enclosed in braces. This combination in curly brackets should follow immediately the described pattern. For example, a combination

**/\d{4}/**

corresponds to a number consisting of 4 digits.Let's look at some control combinations for repetitions:

**{n}** - exactly n instances of the template

**{n,}** - denotes n or more instances of the template  
**{n, m}** - means at least n and not more than m instances of the template  
**?** - denotes zero or one template instance (equivalent to the expression {0,1})

**+** - denotes 1 or more instances of a template (is equivalent to the expression {1,})  
**\*** - denotes zero or more instances of the template (equivalent to the expression {0,})  
  
The repetition symbols indicated in the table correspond to the maximum possible number of matches. For example, expression

/х{1,}/

applied to the line "xxx" will correspond to the maximum number of correspondences, i.e. all three letters "x", found on the line. This is called "greedy" repetition. ("greedy" quantification).

If we want to limit the search to the first occurrence, we can use the so-called "non-greedy", or "lazy" quantification. To do this, after the control combination of repetitions, the symbol "?" Is put. Thus the expression

/х{1,}?/

will correspond only to the first match, i.e. only the first letter "x" in the row.

**An important point! The sign of "laziness" acts only on that quantifier (subpattern) in the template, after which it stands, all other quantifiers remain "greedy."**

The syntax of regular expressions contains a special character for determining the **alternative**, i.e. you can specify more than one variant of the template, the compliance of which will be checked. To separate alternatives, use the symbol "|" - vertical line.  
  
For example, expression

/ma|pa|da/

will match either "ma" or "pa" or "da".

Alternatives of course can also be combined with classes and with repetitions. The template below corresponds to either two digits or two lowercase letters or two uppercase letters:

/\d{2}|[a-z]{2}|[A-Z]{2}/

It should be noted that alternatives are processed from left to right until the first match. After finding the first match, the remaining alternatives will be ignored. In practice, this means that, for example, the pattern /1|12|123/ applied to the string "123" will match the first character, although in alternatives there is a much more complete match.

et's consider one more possibility, which we can use in regular expressions - **grouping**.  
**Grouping** is indicated by enclosing the subpattern in **parentheses** (). In this case, the elements used in conjunction with special symbols, for example |, +, \*,? and others, will be treated as one.  
  
For example, template

/regular(expression)?/

will match the word "regular" followed by the optional word "expression". Another possibility of regular expressions is the indication of the boundary of correspondence. For this, **anchor expressions** are used.  
  
Often we need to find a word that is on a separate line, or at the beginning of the line. Alternatively, you might need to find a single word, but just specify a template in which the word will be surrounded by spaces, we can not - the words will not get the words from which the lines begin or end. Also, the word boundary can be determined by any punctuation mark in the text and it is rather tedious to list all possible combinations in the template. Let's look at what special characters are used to define boundaries:  
  
**^** - corresponds to the beginning of the line with a multi-line search or the beginning of a string expression  
**$** - corresponds to the end of the line with a multi-line search or the end of a string expression  
**\b** - corresponds to the word boundary, i.e. The position between the text (aA-zZ) and non-text character, or between the text character   and the beginning or end of the line.  
**\B** - Corresponds to a position that is not a word boundary.  
**(?=p**) - Positive forward checking for subsequent symbols - makes sure that subsequent characters match the pattern "p" but does not   include them in the search result.  
**(?!p)** - Negative early check for subsequent symbols - requires that subsequent characters NOT match the pattern "p".

Well, the last element of the syntax of regular expressions is **flags**. **Flags**specify global rules for the entire template and are not specified within the slash characters in which the template is enclosed, but AFTER them.  
  
JavaScript supports three flags:  
**i** - indicates that the pattern search must be case insensitive  
**g** - indicates that the search should be global, i.e. all matches in the string must be found  
**m** - indicates that the search should be performed in multi-line mode.

In this lesson, we first look at the **methods of the String** class that allow you to use **regular expressions**:

**search(**regexp**)** **replace(**regexp, newString**)** **match(**regexp**)** **split(**divider**)**

The first and simplest is the **search()** method. As an argument, we pass it a regular expression, and it returns the position number with which the match was found, or "-1" if no match was found.

var myString = "This is just test string";

result = myString.search(/is/);

In the above example, the variable result is the number 2 (the count of positions starts at 0).

**Two important points:**

1. The search () method does not support global search, and the g flag in the regular expression will be ignored.  
2. If the argument is not a regular expression, it will be converted to it by passing it to the RegExp constructor. (RegExp objects will be discussed later).

The following method, with which we get acquainted, is **replace()**  
  
With it, you can perform a **replace** operation.  
As arguments, it takes a regular expression and a replacement string.

var myString = "This is just test string";

result = myString.replace(/is/,"as");

This example will replace the first match found with the pattern ("is") with the substring "as", resulting in the result variable in the result string *"Thas is just test string"*.

It should be noted that this method supports global search and when using the flag "g" it will change all the matches found.

var myString = "This is just test string";

result = myString.replace(/is/g,"us");

In this example, the variable result is a string *"Thus us just test string"*.  
  
It should also be noted that as a second argument to the replace() method, a **function**can be used, in which case we will be able to dynamically change the replacement string.

If the first argument is**not a regular expression**, it will be the same as the search() method is converted to a regular expression using the RegExp constructor. And if you forget to specify the second argument at all, then all the found matches will be replaced by undefined.

The next method that interests us is **match().**  
  
It takes as an argument a regular expression (or converts an argument to it like the previous methods), and returns the **array**of all matches found as the result.

var myString = "Grandfather in the village had 12 apple trees, 5 currant bushes, 10 hens and 33 cows";

result = myString.match(/\d{2}/g);

In this example, we wrote out the notation of the digit "\ d" in the regular expression, indicated that we were looking for its double repetition "{2}" and included the global search flag "g". As a result, the result variable will be an array [12, 10, 33].

However, if the global search flag is not specified, only the first match will be added to the array, it will be written by the zero element. The remaining elements of the array are substrings corresponding to all subexpressions, if any.

And the last method of the String object that allows you to work with regular expressions is **split()**  
  
It splits the string into an array of substrings, using as its delimiter the contents of the argument, which may also be a regular expression. For example, if you use two slashes as a separator, you can separate the protocol from the actual website name in the web address:

var myString = "http://www.example.com/download/pictures";

result = myString.split(/\/{2}/g);

The result of this expression is an array of two elements: *"http:"* и  *"www.example.com/download/pictures";*

In the same way, for example, you can split an array of numbers into an array:

var myString = "1234567890987654321"

result = myString.split(/\B/g);

Using as a separator the definition of a symbol that is not a word boundary (a strange solution, but why not), we get an array in the result variable  [1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 9, 8, 7, 6, 5, 4, 3, 2, 1].

In the second part of the lesson, we will more closely consider the **RegExp object** - its properties, methods and constructor.  
  
Let's start with the designer. As you remember, the **constructor** is a method that takes an argument as an input, and creates an object of the required type at the output.  
  
The **RegExp constructor** can accept a string on the input, the contents of which will be converted to a regular expression. The line must contain the contents of the regular expression, i.e. the text that is usually between two slashes.

var myPattern = new RegExp("q$"); // Create a template that finds the letter "q" at the end of the line

Also, you can pass a second, optional argument to the constructor, in which you can specify flags. For example:

var myPattern = new RegExp("q$","g"); // Create the same template, but add the global search flag

**An important point!** If the text contains a backslash character "**\**", it must be preceded by the same character (ie, it will be "**\\**"), since, as we recall, the backslash is used in regular expressions to specify control characters.

The **RegExp object** in JavaScript has the following **properties**:  
  
**source** - actual text of the regular expression  
**ignoreCase**- A Boolean value indicating the presence of the "i" flag, is read-only  
**global** - Boolean meaning that the flag "g" is present, read-only  
**multiline**- Boolean meaning that the flag "m" is present, read-only  
**lastIndex** - the counter, indicating from which position in the line to begin the search

There are only two**methods for the RegExp** object:  
  
**exec(text)**- performing a search on a string specified as a parameter returns an array of matches found.  
**test(text)** - check the matching of the regular expression, return true \ false.  
  
The **exec ()** method executes a regular expression with respect to the argument string, the result of its operation is an array into which the matches come. If no match is found, the result is null. And if there is a match, it falls into the array with a zero element, with the lastIndex property of the object shifted to the position immediately following the found substring. Let's look at an example:

var myString = "This is just a test text"; // Specify a string to search

var myPattern = /te|is/g; // Create a template - either "te" or "is"

result = myPattern.exec(myString);

//result will be equal to "is" - the first match of the pattern, the lastIndex property will be 4

result = myPattern.exec(myString); //result == "is" - second pattern match, lastIndex == 7

result = myPattern.exec(myString); //result == "te" - third pattern match, lastIndex == 17

result = myPattern.exec(myString); //result == "te" - fourth pattern match, lastIndex == 22

In this example, we call the exec () method four times in a row, each time it shifts the search start pointer to the position that follows the match found and assigns the result itself to the match found.

The **test() method** executes a regular expression with respect to the argument string, the result of its operation is a logical value - true if there is a match, and false if not. The lastIndex property of the object as well as the exec () method will shift to the position immediately following the found substring.  
  
**An important point!** If no match is found, lastIndex will be shifted to position 0 and the search can begin again.  
  
Let's look at the same example:

var myString = "This is just a test text"; // Specify a string to search

var myPattern = /te|is/g; // Create a template - either "te" or "is"

result = myPattern.test(myString);

// result will be true, because the first match is found, the lastIndex property will be set to 4

result = myPattern.test(myString); // result == true, lastIndex == 7

result = myPattern.test(myString); // result == true, lastIndex == 17

result = myPattern.test(myString); // result == true, lastIndex == 22

result = myPattern.test(myString); // result == false, lastIndex == 0

In this example, we call the**test()** method four times in a row, each time it shifts the search start pointer to the position following the match found and assigns the result variable to the boolean value - true if a match is found and false - if not. In the last run, a match was not found, so the lastIndex pointer is set to 0.

**CLOSURES**

In this topic, we will discuss the basics of such a concept as **closures**.  
  
In general - **closure** is a function that was declared inside another function.  
There is one more condition - this function must have access to the variables of the function within which it was declared.  
  
Thus, such a function (**closure**) **has access to data inside itself and inside the parent function**.  
Also, the internal function can access not only the variables, but also the input parameters of its external function.

Let's look at example.

function greetPirate(pirateName) { // Declaring a Parent Function

var greeting = "Hello ";

  function checkCaptain() { // Closure declaration

if (pirateName != "Jack Sparrow")

      return greeting + pirateName;

    else

      return greeting + "CAPTAIN " + pirateName + "!";

}

  return checkCaptain();

}

In this example, we wrote a function that gives a greeting to the pirate - adds "Hello" to the name that is fed to the input. Inside this function, we created a "checkCaptain" closure that checks the input parameter of the parent function - if the name of the pirate is "Jack Sparrow", the closure adds the word "CAPTAIN" to the name and for Captain Jack Sparrow displays a personal greeting "Hello CAPTAIN Jack Sparrow".

As a result, when you call, we get the following result:

console.log(greetPirate("Mad Dog"));             // Will return to console "Hello Mad Dog"

console.log(greetPirate("Jack Sparrow"));     // Will return to console "Hello CAPTAIN Jack Sparrow!"

Closures have two important properties.  
  
**1. Closure can access variables of its external function even after its completion.**  
  
In practice, this means that even after the execution of the external function is completed, the internal one can still be called and has access to the variables of the external function.  
  
Let's look at an example:

function pirate() {

    var pirateName = "noname";

    return {

        getName: function() {

            return pirateName;

        },

        setName: function(newName) {

            pirateName = newName;

        }

    }

}

We described a function with two closures: one returns the value of the variable from the calling function, the second - changes it. Let's see what happens in practical use:

var newPirate = pirate();

console.log(newPirate.getName()); //Output the current contents of the variable - there is the "noname":

newPirate.setName("Jack Sparrow"); //Change the value of the variable to "Jack Sparrow"

console.log(newPirate.getName());    //Display the current contents of the variable - get "Jack Sparrow"

In this example, we see that the closures got access to the variable of the external function after it was completed.

**2. Closures do not store the contents of external function variables, but references to these variables.**  
  
Let's consider in this context the classic example of closure, described in most sources - the counter.

function makeCounter(initialValue) {

  var currentState = initialValue;

  return function () {

    currentState = currentState + 1;

    return currentState;

  }

}

In this case, we described a function inside which there is a closure that increments the counter every time and returns it. Let's see what happens if you call it many times in a row.

var myCounter = makeCounter(5); // Create an instance of the counter and set its initial value = 5

console.log(myCounter()); // The console will display the value 6

console.log(myCounter()); // The console will display the value 7

console.log(myCounter()); // The console will display the value 8