1. Javascript basics: comments, variables and casts

1.1. Javascript basics: common knowledge

Javascript is a high level programming language, supported by all modern www-browsers. Javascript is one of the core technologies of the World Wide Web (web for short), together with HTML and CSS. In this course, we will be going over the basic concepts of javascript programming such as variables, arithmetic operations, loops and objects.

Javascript was originally developed by Netscape, and was a dynamic script language meant mainly for use in a web environment. The most important application of Javascript is to use it for adding dynamic functionalities to web pages. It's most commonly used as a part of web browsers which allow for scripts originating from the client to operate with the user, gain limited control of the browser, communicate asynchronically and edit documents that are shown to the user. Javascript is also used when programming servers (such as a node.js runtime environment), for video game development and the creation of desktop and mobile applications.

Javascript should not be confused with Java, the programming language.

1.2. Javascript basics: comments

Like with every other programming language, you can comment your code in Javascript. These comments are not added into the structure of the program being run, but rather are instructions for someone reading the programming code so that they can understand the structure of the program. Comments are therefore notes left by the creator of the code which explain or clarify parts of the code both for the creator himself and anyone coming after them to update the code. The main purpose of commenting is to make work easier, but some programs can also make use of comments. a JS-compiler will pass over every comment without handling them at all.

There are two ways to leave comments:

```
// this is a comment on one row only

/* this is a comment
which
occupies
four rows */
```

1.3. Javascript basics: defining variables (1/4)

By definition *data* is everything that has meaning to a computer. in JavaScript there are seven (7) different data types. These are the following: *undefined, null, boolean, string, symbol, number* and *object.*

As you may have learned earlier, a computer separates *numbers* (such as 90) and *strings* (such as "90"), which are a sequence of symbols. A computer can calculate numbers, but not strings.

variables are where a computer stores and manipulates data dynamically. In practice, this means that the computer calls on the name of the variable, rather than the data itself. Anything from the seven types of data can be stored into variables.

Variables have the same analogies that we are familiar with from mathematics, such as **x** or **y**. In other words, variables are references to the data to which we want to refer. The difference here is, however, that the *values* contained within these variables can change, unlike in math problems where they stay the same.

The following defines the variable **ownName**:

```
var ownName;
```

The word **var** means that this is a variable, and the word **ownName** is the name of the variable. Please take note that variable names have some limitations. For example, a name needs to be a connected string of characters. Therefore *var own Name* is not a variable, since *own* and *Name* are not connected.

When naming strings, you should also take note that JavaScript is case sensitive, so capital letters are separate from small letters.

Every variable with two exceptions behave as objects in JavaScript. These exceptions are **null** and **undefined**.

1.4. Javascript basics: defining variables (2/4)

Values are saved into variables from right to left. This means that everything on the right side of the operator **=** will be solved before moving the value to the left side of the operator. For example:

```
OwnVariable = 5; // (1)
OwnNumber = OwnVariable; // (2)
```

In phase (1), the value for *OwnVariable* is set to five, and is then fed to become the value for *OwnNumber* in phase 2. Additionally, definitions like this are possible:

```
OwnVariable = (100*6)/2 + OwnVariable // (3)
```

In phase (3) we first calculate (100*6)/2 which is 300, and then add the value of *OwnVariable*, which is 5 as per phase (1). This means that after running this operation in phase (3), the value for the variable *OwnVariable* becomes 305.

When multiple operators are being used at the same time, there is an order of operations, a *precedence*. This we already know from basic mathematics. Division (/) and multiplication (*) have a higher precedence than addition (+) and subtraction (-). If the operators have the same precedence, they are calculated from left to right. For example the calculation 1+2-4 is accomplished by first calculating 1+2 and then subtracting 4 from the result, leading to the solution being -1. Similarly in the situation 1*2+3*4/4, we will first calculate 1*2, then 3*4, then /4, leading to 2+3 with the solution being 5.

In addition to these operators, there is also the operator for remainders, *modulo* (%), which tells us the remainder. For example 12 % 3 = 0, while 12 % 5 = 2.

JavaScript also has special numbers such as infinity and Not-a-Number (NaN). For example infinity - infinity gives the result NaN, similarly to Infinity/Infinity = NaN.

In many programming languages numbers can have different *allocations* for them. For example in the C-language numbers have a 8-bit and 16-bit variable definition. In JavaScript, however, these are always 64-bits. This means that if the number being saves is 240, for example, it will only require 8 bits of allocation, but JavaScript always allocates 64 bits for it.

In JavaScript you don't definte the type of variable separately. The data type of variables can be changed dynamically while the program is running.

1.5. Javascript basics: defining variables - strings (3/4)

String is the name for Javascript's method of saving strings. For example, all of the following save a string:

```
'this is a string'
"this is also a string"
´even this is a string´
```

Unlike in many other programming languages, JavaScript accepts other quotation marks for saving a string, as long as they're the same character on both sides of the string.

strings may also include formatting, which can lead to problems with the quotation marks. It's a good idea therefore for compatibility and simplicity to use apostrophe's as the borders for JavaScript strings, since most marking languages use quotation marks. For example in the following code, the quotation marks in the string will not get confused with the apostrophes acting as the borders of the string:

```
'<h3 title="Example">Recommendation</h3>'
```

Strings can use escape characters as well, for example a line break. escape characters are indicated with the symbol \, followed by the symbol you wish. For example a quotation mark within a string can be done by writing \" and tabbed by \t. A Line break can be done with \n. For example the string "this is a string\non two lines" will print out the following:

```
this is a string
on two lines
```

Two \\-symbols merge into one inside a string. If we want to print \n inside the string, we could do the following: "The symbol for a line break can be printed like this: \"\\n\".", which prints the following:

```
"The symbol for a line break can be printed like this: "\n".
```

Strings are saved into Unicode, which is 16-bits. This means that every single symbol in a string, an *element*, can have 2^16 values.

Strings cannot be divided, multipled or added onto, but they can be combined. For example the strings "super"+"califragi"+"listi"+"cexpiali"+"docious" can be combined with the + operator to form the word

supercalifragilisticexpialidocious

```
var first_name = "John";
var last_name = "Johnson";
console.log("Hello " + first_name + " " + last_name);
```

JavaScript contains many other methods for handling strings. In the next table are some methods you can use when programming.

Method	Description
length	the length of a string (feature)
CharAt(position)	Returns the character from the position (the first character is 0)
CharCodeAt(position)	Returns the Unicode from the position
substring(starting#, ending#)	Returns the string from between the starting and ending numbers.
toLowerCase()	Turns the string to lowercase letters
toUpperCase()	Turns the string to uppercase letters
split(separator)	Splits the string to table elements at separator symbols
indexOf(string)	Returns the position of the string (or -1 if it's not found)
lastIndexOf(string)	As above, but starting from right to left.
substr(starting#, length)	Returns the desired number of characters starting from the starting number
replace(string1,string2)	replaces the characters from string1 with the ones in string 2 (only the first found case)
fontcolor()	returns the color of the font
fontsize()	returns the size of the font
search(string)	returns a whole number larger than 0 if the string is found, and -1 if not.

1.6. Javascript basics: defining variables: unary, binary and boolean operators (4/4)

The function **console.log** prints the desired text onto the screen, or more specifically into the console, which is usually the compiler window or a browser window. For example *console.log("Hello World!\n")* prints *Hello World!* and adds a line break.

An operator is a character symbol or combination in a statement, which performs a certain operation.

Operators are used for calculations, comparisons between numbers and to allocate things into variables. The most common operator is the equal-sign (=) which is the allocation operator. With it you can allocate a new value for a variable. It should not be mistaken with the comparative operation ==, which is used to see if two numbers are equal.

```
// position
var a= 2;
var b= 2;

//comparison; whether a equals b
if(a==b)
```

Comparison operators are:

```
==, an abstract equality operator. If necessary, it will convert the variable type.
!=, an abstract inequality operator. It will also convert the variable if necessary.
===, a strict equalitty operator. It will not convert the variable type.
!==, a strict inequalitty operator. It will not convert the variable type.
>, an abstract "larger than" operator, which converts the variable if needed.
<, an abstract "smaller than" operator, which converts the variable if needed.
>=, an abstract "larger than or equal to" operator, which converts the variable if needed.
<=, an abstract "smaller than or equal to" operator, which converts the variable if needed.
```

arithmetic operators are operators that perform calculations, which return a single numerical value. These operators are divided into two types:

- Binary operators, which perform a calculation between two operands, which are either numbers or variables containing a numerical value
- unary operators, which perform a calculation on one operand, which is a variable containing a numerical value.

An unary operator is an operator focusing only on one operand. You can test unary operators with the *console.log()*-function. For example *typeof* is an unary operator. *console.log(typeof 8)* prints that 8 is a number.
Similarly, *console.log(typeof "8")* will print out that "8" is a string.

value changing operators can be used to change a value by one unit. It's possible to add or subtract one, add or subtract another value, or turn the number negative.

```
++ adds one
-- subtracts one
- negates the number
+= adds a value
-= subtracts a value
*= multiplies by value
/= divide by value
```

For example:

```
var b = 5;
b++; // 6

var a +=b // this is the same as a = a + b
```

In comparison - and + are binary operators. They take a value from both sides. On the other hand, they can be used as unary operators as well. For example *console.log(-(10-11)* will result in 1.

In regard to strings, the + operator can be used to combine strings together.

truth values, boolean values are ones that will either give you the answer *true* or *false*.

For example *console.log(10>9)* will be *true*. Similarly, *console.log(9>10)* will be *false*.

Similarly we can compare strings. For example *console.log("Salesman" > "Salesman")* will return the value as *false.*

And as another example, *console.log("Salesman" != "Salesman")* will return the value as *false*.

Logic operators are the following:

```
! NO-operator. It returns the opposite truth value.
&& AND-operator. It returns true if every operand is true, otherwise it returns false.
|| OR-operator. It returns true if at least one operand is true. Otherwise it returns false.
```

When combining arithmetic and boolean operators, it's not always clear when parenthesis are required for precedence purposes. Usually you can get by by knowing that || has the lowest precedence, followed by && and then comparison operators (such as < and ==). This order has been selected to reduce the number of parenthesis that would be needed.

For example the following would return the value as true: 1*2==2 && 3*4 != 13 > 11.

JavaScript also has a ternary operator, which works like this: Let's introduce the variable person, and see if he's eligible for a drivers license:

```
var person = {
name: 'jussi',
age: 20,
driver: null
};person.driver = person.age >=18 * 'Yes' : 'No';
```

The last part can be presented like this:

```
person.driver = ((person.age >=18) ? 'Yes' : 'No';)
```

In other words, it tests to see if a person's age is atleast 18 and return the value "Yes" if this is true. Otherwise we return the value "No". In our example the age is 20, and therefore the returned value is:

```
person.driver = 'Yes';
```

1.7. Javascript basics: automatic type conversion

When the type of data is unclear, JavaScript will use a complex set of rules to have the types match. Let's look at the following examples:

```
console.log("5" + 5) // (1)
console.log("5" - 1) // (2)
console.log(10 * null) // (3)
console.log("eight" * 8) // (4)
console.log(true == 0) // (5)
```

In (1), we will get the value 55. In (2) we will get 4. In (3) we will get 0. In (4) we will get NaN and in the last one (5) we will get *TRUE*. Take note: *console.log(null = undefined)* will return *true*, but *console.log(null == 0)* will return *false*.

Operators && and || handle different types of values in different ways, in a slightly odd fashion. For example *console.log(null || "Jack")* returns the value *Jack*.

Rule: The operator || returns the leftmost value, if it can do a type conversion. In other cases it will return the rightmost value. In this case you cannot do a type conversion, as *null* cannot be converted into a string. Therefore the value *Jack* is returned.

&& works similarly to ||, but it's exactly the opposite. Therefore *console.log(null || "Jack")* returns the value *NULL*.

1.8. Javascript: mathematic operations

With the object *math* we can solve "advanced" math problems. Normal calculations such as subtraction and addition can be performed with basic operators or the *eval()*-function.

The *eval()*-function converts a string-type sentence into a mathematic form that can be calculated. The following example demonstrates how it works:

```
<script type="text/javascript">
// Let's assign the calculation into the variable as a string:
var calculation = "2 + 4 - 7 / 16";

// Without the eval()-function this will print "2 + 4 - 7 / 16":
console.log(calculation);

/*The eval-function woöö tirm tjos string-type variable
into a mathematical form and then perform the calculation.
This will print out "5.5625": */

console.log(eval(calculation));
</script>
```

Javascript handles mathematic operators via the *math*-object. The *math*-object contains several constants of some of the most common values used in mathematics:

CONSTANT	DESCRIPTION
Math.E	Euler's number *e*, approximately 2,72
Math.LN10	Natural logarithm of 10
Math.LN2	Natural logarithm of 2
Math.PI	The value of pi, approximately 3,14
Math.SQRT1_2	The square root of 1/2
Math.SQRT2	The square root of 2

With *math*-object's *round*-method, we can round a number given as a parameter under the normal rules for rounding them.

floor and *ceil*-methods will also round a number given as a parameter. *floor* down and *ceil* up to the next whole number.

The method *max* returns the largest of the numbers given as parameters, while *min* will return the smallest one.

random will return a random number from between 0 and 1. The number will be a decimal number, such as 0.737432..., and for it to be useful, you might want to multiply it by 30 to get a decimal number between 0 and 30. By using the *round*-method, it can be turned into a whole number.

Other methods in the *math*-object are:

METHOD	DESCRIPTION
Math.abs(param)	Returns the absolute value of the given value.
Math.acos(param)	returns the arccos of the given value.
Math.asin(param)	Returns the arcsin of the given value.
Math.atan(param)	Returns the arctangent of the given value.
Math.cos(param)	Returns the cosine of the given value.
Math.exp(param)	Returns e to the power of the given value.
Math.log(param)	Returns the natural logarithm of the given value.
Math.pow(param_1, param_2)	Returns the value given as param_1 to the power of param_2.
Math.sin(param)	Returns the sine of the given value.

METHOD	DESCRIPTION
Math.sqrt(param)	Returns the square root of the given value.
Math.tan(param)	Returns the tangent of the given value.

2. Javascript basics: program structures

2.1. Statements, bindings and return values

Source code or just code refers to a program's contents in text form, written in a programming language.

An expression is a part of code that produces a value.

A statement is a single command or action usually written on its own line.

In Javascript, *expression* thus refers a part of a statement while *statement* is, well, a full statement. For example:

```
!TRUE; // (1)
2; (2)
```

(1) and (2) are both functioning Javascript statements: each has an expression and ends with a semicolon.

Statements can be grouped into blocks. A *code block* is a separated part containing one or more statements, used most commonly with loops and functions. Blocks are represented by curly brackets:

```
{ //block begins
  //statement or statements inside the block;
} //block ends
```

Bindings or *variables* are components used by JavaScript for keeping the internal state of the program in order. Let's review this example:

```
var temp = 10 * 10; // (3)
console.log(temp * temp); // (4)
```

(3) is a functioning JavaScript statement which forms the following binding: first it calculates 10 * 10 = 100, then "binds" the result (100) to the variable *temp*. After this, the binding is used and (4) prints the result of 100 * 100 which is 10 000.

When a variable is defined using the keyword *var*, its scope will be the function inside of which it was defined. This means it's that function's local variable. var can be used to define the variable multiple times in different parts of the function, but it's still the same variable.

In version 1.7 of JavaScript, the possibility of defining block-specific variables by using the keyword *let* instead of var.

The need to change a variable's binding during the program is often necessary. Look at the following example (5-8):

```
var mood = "happy"; // (5)
console.log(mood); // (6)
mood = "sad"; // (7)
console.log(mood); // (8)
```

On line (5) we define the variable mood and bind the value "happy" to it. It's printed on line (6) to get the output "happy". After this, the variable mood is initialized again on line (7) so that the value will be "sad". After this, the re-initialized variable will be printed on line (8).

Multiple variables can be defined in the same statement, as long as they're separated by commas (9):

```
var variable1 = 1, variable2 = 2, variable3 = 3; // (9)
console.log(variable1 + variable2 + variable3); // (10)
```

The line (10) outputs 6.

A variable can also be defined to have a constant value which can't be changed. For example, const species = "bird"; would define the value of the variable species to be bird for the rest of the program's execution.

Keywords can't be used as variable names. Some examples would be: *break case catch class const continue debugger default delete do else enum export extends false finally for function if implements import interface in instanceof let new package private protected public return static super switch this throw true try typeof var void while with yield*.

Variable names should also be descriptive. The names may not contain spaces, but capital letters and underscores may be used as shown in the example below (the first line should be replaced by one of the other options shown):

```
smallfurrytoy
small_furry_toy
smallFurryToy
```

Functions are subroutines in which we can store functionality for later or repetitive use. As with variables, functions have to be defined before using them. Function declarations have the same naming conventions and scope rules as variables.

A function declaration's syntax is as follows:

```
function functionName(/*parameter_list*/) {// function block begins
  /* The statement(s) to run */
}// function block ends
```

Already introduced in the previous chapter, JavaScript has the function console.log, which outputs into the system's default output stream. The names of bindings may not have periods, but the console.log function may. This is due to the statement fetching the console binding's log property. This may be too complicated at this point, but it'll be explained in more detail in chapter 4. The line (11) has an example of a return value:

```
console.log(Math.max(2, 4, 5) + 100); // (11)
```

The system prints the value 105. The function Math.max selects the greatest value among the input parameters, then returns it as the function's *return value*. After this, the value 100 is added to it, making the final result 105.

More about functions in chapter 4.

Executing instructions

The program's instructions are executed in order. The next example has two statements, first of which will read a value, while the second will calculate and output the read value's square.

```
var num = Number(prompt("Enter a number")); //(12)
console.log("The entered number's square is " + num * num);
```

The function Number on line (12) will convert the input string into a numeric value.

if-statements

If we want the program's execution to split to another route based on some condition, we use the if statement:

```
if (condition) {// if-block begins
  /* The statement(s) to run if
     the condition returns true.
     Otherwise the execution of the
     program continues as normal
     after the code block. */
}// if-block ends
```

Look at the following example:

```
var num= Number(prompt("Enter a number"));
if (!Number.isNaN(num)) { //(13)
console.log("The entered number's square is " + num* num);
}
```

On line (13) we test the data type of the input using JavaScript's isNaN function. It returns the value true if the given argument doesn't represent a numeric value, being of type NaN (Not a Number). In this specific case, the if-block will be executed if the input value represents a numeric value.

The codeblock is placed between curly brackets { and }. To make reading easier, it's recommended to place them as shown in the example above, even if the block would only contain one line of code.

If necessary, multiple consecutive if-statements may be used.

It's also possible to place if-statements inside of if-statements, which allows checking for multiple different conditions as shown:

```
if (condition1) {// if true, check next condition
  if (condition2) {// if true, check next condition
    if (condition3) {
      /* statement(s) to execute if all conditions were true */
    }
  }
}
```

Coherent indenting makes it easier to read and understand the code when going deeper.

else-statement is a condition statement which can be placed (whenever necessary) after the if-statement's code block. When the condition of the if-statement just before it ends up being false, the else-statement's codeblock will be executed:

```
if (condition) {// if condition is true
  /* Statement(s) to execute if
     condition returns true. */

} else {// otherwise
  /* Statement(s) to execute if
     condition returns false. */
}
```

The following example represents the usage of else-statement:

```
var num= Number(prompt("Enter a number"));
if (!Number.isNaN(num)) {
console.log("The entered number's square is " + num * num);
} else {
console.log("Invalid value");
}
```

if/else-pairs can be chained as follows:

```
if (condition1) {// if condition is true
/* Statement(s) to execute if
   condition returns true. */
} else if (condition2) {// otherwise if condition is true
/* Statement(s) to execute if
   condition returns true. */
} else if (condition3) {// otherwise if condition is true
/* Statement(s) to execute if
   condition returns true. */
} else {// optional statement
  /* Statement(s) to execute if
   none of the conditions are true. */
}
```

The next example is about the usage of if/else structures:

```
var num = Number(prompt("Enter a number"));
if (num < 10) {
  console.log("Small");
} else if (num < 100) {
  console.log("Medium");
} else {
  console.log("Large");
}
```

The program in this example examines the input value and prints "Small" if it's below 10. However, if it's equal to or greater than 10, but smaller than 100, it prints "Medium". Otherwise "Large" will be printed.

2.2. Loops

Loops

Loops are programming structures which repeat specific actions for as long as the specified condition is true.

while-statement is executed for as long as the condition remains true. The statement consists of the while-statement and a condition following right after.

`while`-statement's syntax is as follows:

```
/* The following statement's condition is any
   measurable value inside the brackets */

while (condition) {// the loop's block begins
  /* Statement(s) to execute
     if condition is true.
     Otherwise the execution of the program
     continues from after the codeblock. */
}// the loop's block ends
```

The following program outputs the numbers 1, 3, 5, ... , 11 using while-statement:

```
var num = 1; //(14)
while (num <= 12) { //(15)
  console.log(num); //(16)
  num = num + 2; //(17)
}
```

On line (14) we initialize a counter variable. On line (15) we have the condition for the while-loop. The lines (16) and (17) inside the loop's block are executed, if the condition is true. On line (17) we increment the counter. After executing line (17), we recheck the while-statement's condition. If it's still true, lines (16) and (17) are executed again. The program only exits the loop once the condition is false.

The counter's initial value was set to 0, which is usually recommended.

The next program calculates the value 2^{10}. It uses two variables. One (counter) helps keep track of the amount of loops while the other (result) will store the result of the calculation over each round.

```
var result = 1;
var counter= 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
console.log(result);
```

do-while-statement is a variation of the while-statement in which the statements inside the codeblock will be executed at least once on while-lauseen muunnelma, jossa lauselohkon lauseet suoritetaan ainakin kerran, minkä jälkeen toistoehto evaluoidaan (loppuehtoinen silmukka).

The syntax of a do-while-statement is:

```
do {
/* Statement(s) to be executed at least once.
   Will be repeated as long as the condition is true.
   Remember to alter the condition from within this block! */
} while (condition);
```

In the next example, the do-while-statement is used to read an input until something is entered:

```
var yourName;
do {
  yourName = prompt("Enter your name"); //(18)
} while (!yourName); //(19)
console.log(yourName);
```

The condition on line (19) checks if an empty line has been entered. If true, the loop's statement on line (18) will be repeated. Otherwise, we leave the block.

The previous examples used indentation to make the code more readable. In general, the contents of a new block will be indented two or four spaces. Most code editors allow tab to be used, and the width of indentation to be changed in options. The following example has 2 spaces wide indents:

```
if (false != true) {
  console.log("Absolutely");
  if (1 < 2) {
    console.log("No doubt about it");
  }
}
```

When using **for-statements**, we first initialize or define a counter in a start statement, after which the actual condition statement is checked. When the condition is true, we first execute the statements within the codeblock, then the end statement, then check the condition again. We leave the loop once this check evaluates as false.

for-statement's syntax is:

```
for (/*start statement*/;
     /*condition*/;
     /*end statement*/) {// the loop's block begins

  /* Statement(s) to execute
     if condition is true.
     Otherwise the execution of the program continues
     from after the codeblock.  */

}// the loop's block ends
```

The next program calculates 2^{10} using a for-loop:

```
var result = 1;
for (var counter = 0; counter < 10; counter = laskuri + 1) //(20)
{
  result = result * 2;
}
console.log(result);
```

On line (20) we give the counter an initial value, set an upper limit for it through a condition, and define an incrementation. The loop's statements are executed as many times as the counter's value remains lower than 10.

break-statement can be used to forcibly end a loop from within its block, even if the condition would still be true.

The next example showcases how break functions. The program finds the first number higher than 20 and divisible by 7:

```
for (var num = 20; ; num = num + 1) { //(21)
  if (num % 7 == 0) { //(22)
    console.log(num);
    break;
  }
}
```

On line (21) we define a loop which initializes the counter to 20, incrementing the counter's value by 1 every loop. The condition has been left blank. The statement on line (22) uses the modulo operator to check whether the counter's current value is divisibly by 7. If it is, we escape the loop with the break keyword.

If we were to remove the break from the previous example, we'd end up with a loop that never ends, an infinite loop. This is considered a serious error.

In Javascript, the statement for incrementing the counter's value:

```
counter = counter + 1;
```

can be replaced with a shorter form:

```
counter += 1;
```

This form is common and works with all values.

The earlier example (for calculating 2 to the power of 10) can be written in the form:

```
for (var counter = 0; counter < 10; counter += 1)
```

Similarly

```
counter = counter - 2;
```

can be shortened to:

```
counter -= 2;
```

and

```
result = result * 2;
```

can be shortened to:

```
result *= 2;
```

When incrementing or decrementing a value by just one, it can be shortened further:

```
counter ++
counter --
```

switch-statement works for situations where multiple conditions are mutually exclusive. It functions as follows:

1. Evaluate the argument passed to the switch-statement.
2. Compare the statement with each case's expression value in the order they're presented in, until a matching value is found.
3. Execute any cases matching the value until encountering a break-statement or the switch-block ends.
4. The block can have an optional default-statement. Its statement(s) will be executed if none of the cases matched the expression.

The syntax for the switch-statement is:

```
switch (expression) {// the argument usually being a variable
  case expression_value1: statement(s); break;
  case expression_value2: statement(s); break;
  case expression_value3: statement(s); break;
  case expression_value4: statement(s); break;
  case expression_value5: statement(s); break;
             default: statement(s);
}
```

The break-statement is, in theory, optional, but should almost always be included after each case. Otherwise, the statements for the following cases and default will be executed as well, whether or not their condition values matched.

The next example shows how to use a switch-statement:

```
switch (prompt("How's the weather today?")) { //(23)
  case "rainy":
    console.log("Remember to bring an umbrella.");
    break;
  case "sunny":
    console.log("Dress lightly.");
  case "overcast":
    console.log("Feel free to go outside.");
    break;
  default:
    console.log("Unknown weather!");
    break;
}
```

The line (23) asks for the weather outside. If the value is found in any of the cases, suitable instructions will be given. The case matching "sunny" has no break-statement, so the instructions for the case after it will be printed as well.

3. Javascript basics: CSS, Web & HTML

3.1. HTML

HTML

HTML is a mark-up language for the creation of webpages. It's used to describe the structure and text contained within the webpage. The structure of an HTML-page is defined with elements defined in the HTML-language, and a single HTML-document will consist of multiple elements within and after one another.

The elements defining the structure of a page are separated with the smaller than (<) and larger than (>) symbols. The element is opened with a string that begins with < and ends with >, for example *<html>*, and is closed with a string that has a slash (/) after the <, such as *</html>*. You cannot place other elements inside an element.

Most elements need to be closed. Some HTML5 elements such as *
* are "void", and don't need to have a separate ending element. If you want to, you can use an XHTML-style /, such as *
*

The following is an example of an HTML page:

- The first line tells the browser that this is an HTML document.
- On line 2 we begin the html code, ending on the last row (with /html).
- The HTML page itself consists of two sections: *head* and *body*.

- The *head* section usually consists of programming code, style definitions and programs from elsewhere on the internet to be used on the site.
- The *body* section is where you write everything that will be shown in the browser window. These can be further modified, removed or added with program codes.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>

    Here's the part that'll be in the browser window

  </body>
 </html>
```

The only formatting that will be shown by the browser will be spaces. The rest of the formatting must be done with tags.

Often, we'll also want some functionality on our HTML pages in addition to displaying information on it. This may involve reading user input using things like button, checkboxes and text input fields. All these elements are of the type <input> and their type attribute defines how they work. Some of the most common ones among these would be the ones just mentioned: button, checkbox and text.

```
<input type="button" value="Click!">
```

Click!

```
<input type="checkbox" checked>
```

☑

```
<input type="text" value="HTML & JavaScript">
```

HTML & JavaScript

The state of the checkbox or the content of the text input field can be checked using JavaScript, while the button would usually call some function. We'll learn more about functions in the next chapter.

Additional resources

HTML has been in use since the beginning of the Internet as we know it, so there's plenty of information about it available on the web.

- A large portion of HTML elements are looked into, for example, on the W3Schools website.
- For much more detailed information, there's documentation provided by Mozilla.
- W3Schools offers information about CSS as well - the subject we'll be looking into next.

3.2. CSS

CSS

CSS (Cascading Style Sheets) are files that define how the elements on a web page will be shown to the user. with HTML we will define the structure and content, while the style sheet is used to define its layout.

Style sheets can be used to define what a page looks like. A style sheet is a separate file from the HTML document, containing different style definitions. There can be multiple style sheets. For the HTML document to be able to access the style sheet, the style sheet must have its location specified in the *link*-element, located within the *head*-element like this:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" >
    <link rel="stylesheet" type="text/css" href="stylesheets/style.css">
    <title>Page title (Shown on the browser bar)</title>
  </head>
  <body>

        <!-- Page content: This is how you add a comment on a page -->

  </body>
</html>
```

We tell the *link* element the style of the file we're referring to (*rel="stylesheet"*), the type (*type="text/css"*) and location (*href="location.css"*). The name of the style sheet should be defined in the location. style sheets have the filename extension *.css*. For example if the style sheet is within a folder named *stylesheets* that is in this file and the name of the style sheet is *style.css*, we will set the *href* attribute of the *link*element to be *"stylesheets/style.css"*.

The following is an example of a simple style sheet:

```
body {
    background-color: rgb(200, 200, 200);
    margin: 0;
    padding: 0;
}
```

The above style sheet tells us that the *body* element (meaning the main part of the HTML document) background color has RGB values of 200, 200 and 200 , making it fairly light. RGB comes from Red, Green and Blue, with each number indicating how much of the colour to add. Each color is indicated with a number between 0 and 255. If everything is set to 0, the result is black. If everything is 255, it'll be white.

Next we'll be showcasing some CSS attributes. If we want every paragraph to have a line height of 2 and green text, we add the following rule into the CSS file:

```
p {
  line-height: 2;
  color: green;
}
```

Now all the text that is between the tags *<p></p>* will have a line height of two and will be green.

Every element can be given an ID attribute, which gives it an unique definition. The same ID can only be used on the same page only once, so if we want only one paragraph of text to be green with a line height of 2, this is what we'll do. First, we'll give the desired element the ID *highlight*. Then, we define the following CSS rule:

```
<p id="highlight">The paragraph to be highlighted</p>
```

```
#highlight {
  line-height: 2;
  color: green;
}
```

Please note the #-symbol, used in CSS (but not in HTML) to mark ID.

Classes work similarly to IDs, but with the difference that a class can be defined for multiple elements as per the following example:

```
<p>One highlighted paragraph.</p>
<p>Another highlighted paragraph.</p>
```

```
.highlight {
  line-height: 2;
  color: green;
}
```

Notice the dot (.). It is used in CSS to denote classes.

Construction hints

You should ensure that web pages are usable regardless of which browser, device or monitor is used. This means that the page should be readable no matter the screen size, resolution or colours available. The pages should also be readable as a print-out, as something you listen to or while using a braille browser.

How can we design pages that conform to these standards and are easy to access?

1. Prioritize the purpose of your website, not how it looks. The service or functionality your page is meant to offer to its user. Let the form of your website follow its function rather than going for a specific look and trying to force it to work.
2. Don't use HTML marking to define the look of your page. Use separate style sheets instead.
3. Use CSS correctly to suggest the layout of the page, not to control it. That way, your page will function wonderfully with any kind of browser, even in the future. For example this can be done by suggesting many different fonts on the style sheet. Never define a font size with strict numbers. A nicer way to go about this is by defining the size of text in relation to other text within the element. For example by defining the headline to be 30% bigger than the text used in the main body. Similarly you can set the margins of the page as percents or other variable numbers.
4. Try to avoid using only color-coding on a page to denote a specific meaning. A lot of people are colorblind.

3.3. JavaScript

JavaScript

HTML is a mark-up language for creating web pages and creating content while CSS is a language for defining the style on web pages. JavaScript on the other hand is a language for adding dynamic functionality. With JavaScript, we can create web pages that "respond" to user actions such as mouse clicks, feeding information by writing it and navigating the page. It can be said that JavaScript can be used to *program* different functions for the browser to execute.

Originally JavaScript programs are included into HTML pages, where they execute webpage functionalities locally. This means that the Browser has a way of handling Javascript, most commonly a compiler. Using the language independently from the browser is also possible. Lately, using it to handle hosting programs has become more common.

JavaScript code is added to be a part of an HTML-document with the *script* element:

- The programming commands within the *<script>* element (tag) within either the *head* or *body* sections.
- Referring to an external JavaScript sourcefile.
- By defining the JavaScript command to be a value of an HTML attribute
- In the tags of an event handler

The following is an example on how to write script within the *<script>*-element:

```
<html>
 <head>
 <title>JavaScript-example</title>
   <script type="text/javascript">
   // code goes here....
   </script>
 </head>
<body>
</body>
</html>
```

Just like CSS style definitions, the JavaScript source code should be separated from the HTML document. The JavaScript file extension is commonly *.js* and it is referenced to with the *script* element. The attribute *src* is used with the *script* element, which tells the location of the source code file. If the source code is in a folder named *javascript* in a file named *code.js*, you'd use the following script element: *<script src="javascript/code.js"></script>*. It's important to note that the *script* element is closed even without containing any text within. In the following example the script is located in an external file *JavaScript-codes.js*, which is referred to in the HTML document:

```
<html>
 <head>
 <title>JavaScript-example</title>
 <script type="text/javascript" src="JavaScript-codes.js"></script>
 </head>
<body>
</body>
</html>
```

There is no one right way to place the *script* element within the document, as it depends heavily on the purpose of the code. If the code needs to be run when the page loads, it needs to be placed within the *head.* Otherwise, it could be placed within the *body* of the document. There is no limit to how many *script* elements you can include, either.

The most common way to add JavaScript code onto a page is to add them into the very end of the page. One of the reasons for this is that the browser will retrieve the JavaScript file as soon as it encounters it within the document, leaving everything else to wait until it has been loaded. If the source code file is loaded at the very end, the user can already see contents of the page before the code is loaded, since browsers commonly show the page to the user as it's being loaded. This makes a page look like it's reacting and loading faster.

In the following example we will show adding JavaScript code onto a webpage and the page on the browser. In the folder *javascript* we have the file *code.js* with the function *SayHello*, which prints out a message for the console:

```
function sayHello() {
    console.log("BAD = browser application development");
}
```

The HTML document is like this:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" >
    <link rel="stylesheet" type="text/css" href="stylesheets/style.css">
    <title>Page title (shown on the browser bar)</title>
  </head>
  <body>
    <header>
      <h1>The header of the page</h1>
    </header>

    <article>
      <p>The regular text shown on the page is inside a <p> element. Below is a button which calls the function sayHello() whe
      <input type="button" value="Greet" onclick="sayHello();" />
    </article>

    <!-- We load the JavaScript code at the end of the file! -->
      <script src="javascript/code.js"></script>
  </body>
</html>
```

The header of the page

The regular text shown on the page is inside a <p> element. Below is a button which calls the function sayHello() when pressed.

Greet

Pop-up windows

A pop-up window is a window that appears on the screen while the page is being loaded or an action is done, which stops executing the code on the page until the user acknowledges it.

An *alert box* requires the user to acknowledge it by clicking the "OK" button. It can be used to make the user aware of something:

```
alert("sometext");
```

A *confirm box* requires the user to acknowledge it, either by accepting ("OK") or rejecting ("Cancel") it. It can be used when you want to confirm something from the user. "OK" returns the value *true*, while "Cancel" returns the value *false*.

```
confirm("sometext");
```

A *prompt box* will ask the user to input characters and to accept or reject it. It can be used when you wish the user to input information. The box will have the buttons "OK" and "Cancel", with "OK" returning the string of characters the user inputted and "Cancel" returns the value *null*.

```
prompt("sometext","oletusarvo");
```

Controlling elements in the HTML document (Document Object Model, DOM)

To use JavaScript for controlling certain elements of an www-page, we need to be able to refer to the elements somehow. This is why the browser window (or tab) and the document being shown have their own objects in the program running the JavaScript Environment: *window* and *document*.

Window object

The *window* object is a global object being executed in the browser. This means that it is an object, which has it's attributes from every global variable. So, every variable created outside of a function, a global variable, becomes an attribute for the window object. The opposite is also true, so if a new attribute is created for the window object, it will become a global variable, and will therefore be usable everywhere where it hasn't been "hidden" by using a local variable with the same name.

Document object

The document object represents the entire document loaded into the browser, and with it by using JavaScript we can examine and edit any part of the document. For example you need to be able to set values for elements on the page, and you need to be able to retrieve them as well. You can access elements on a document with the command *document.getElementById("identifier")*, which will return the element with the ID "*identifier*".

Handling the value of an HTML element

In the next example we will have a text field with the HTML code *<input type"text" id="textfield"/>*. The ID for the field is therefore *textfield*. To access the element *textfield* we will use the command *document.getElementById("textfield")*. The text field element has the attribute *value*, which will be printed like this:

```
var value = document.getElementById("textfield").value;
console.log("the value of the text field was " + value);
```

In the following example we will retrieve the text field of the previous example, and set it's value to 5:

```
document.getElementById("textfield").value = 5;
```

Finally we create a program that asks the user for the new value of the field:

```
document.getElementById("textfield").value = prompt("Write something!");
```

Every element does not have the *value* attribute, but some are shown with the value within the element. You can set a value within an element with the attribute assigned to the value, *innerHTML*.

You can add the *onclick* attribute into an HTML element, which has it's value defined as a JavaScript code that will be executed on clicking the element:

<element onclick="Program code or function call">

4. Javascript basics: functions

4.1. Functions and closures

Functions are subroutines that can be used for introducing functionalities to be used later or repeatedly. Functions need to be introduced before they can be used. Functions have the same naming conventions and visibilities as variables.

The syntax for introducing a function is like this:

```
function functionName(/*set of parameters*/) {// function body starts

  /* Statement(s) to be executed */

}// Function body ends
```

Function introductions begin with the keyword *function*, followed be:

- *functionName*, an identifier the programmer gives for the function.
- () brackets, including optional (0-255) parameters separated with commas (information to be sent to the function when called)
- {} curly brackets, including the body of the function, meaning it's functionality.

In the next example we will introduce a function which returns the square of the calling argument:

```
const square = function(x) {
  return x * x;
};
```

We call the function with the value 12 for the argument:

```
console.log(square(12));
```

Another way for defining a function is to use a *function expression*. In the following example we will be using an anonymous function to calculate the sum of the numbers of the numbers within. The interesting part about the code is that the function only exists for the duration of its execution:

```
var result = (function(beginning, end) {
    var sum = 0;
    for (var i = beginning; i < end; i++) {
       sum += i;
    }
    return sum;
    })(1, 3);

console.log(result); // 3
```

Anonymous functions are therefore functions that aren't bound to variables, and therefore won't get a name.

A function can have zero or more call parameters, as seen in the following examples:

```
const ring = function() {
  console.log("Pling!");
};

ring(); //(1)
```

```
const power = function(base, exponent) {
  var result = 1;
  for (var counter = 0; counter < exponent;   counter++) {
    result *= base;
  }
  return result; //(2)
};

console.log(power(2, 10)); //(3)
```

The first function "rings" the text "pling" and there is no call parameter given it (1)

The latter function gives a base number and an exponent in it's call, and it prints the number to the power of the exponent (3). The function returns the result using the *return* statement (2).

If there's no argument value given to the *return* statement or *return* is missing, the function will silently and invisibly return the value *undefined*.

The function forms a scope. This means that the formal parameters, local variables and local functions are only visible (usable) only within the function. To be more specific, those names and designations are only visible within the function, meaning that they are *local*.

Definitions done outside of the function are still visible inside the function, since they are *global*.

The basic idea of nested scopes is that you can see out from the inside, but not in from the outside.

On an inner scope, the outer scope can have a new definition, which will cover the old definition as per the following example:

```
function f() {
  var a=1, b=2;
  function ff() {
    var a = 11; // covers the surrounding function a
    write(a+" "+b); // 11 2 // This also applies to var-variables
  }
  ff();
  write(a+" "+b); // 1 2
}

f();
```

The *bound variables* of a function are the formal parameters of the function and the local variables defined within the function. Therefore variables that only have meaning within the function and only exist while the function is being executed.

free variables are the variables not included in the function, but which are referred to in the functio, which are visible from the function.

```
function f(x) {
  var a=1, b=2;

  var g = function(y) {
    var c=3;
    return a+b+x+ // free variables
    y+c; // bound variables
  }
  return g(4);
}

write(f(5)); // 15
```

function calls within functions use a *stack* to save their return state and address. If recursive functions take up too much space from the stack, we will receive the error "out of stack space" or "too much recursion". The calls in the following function will fill the stack:

```
function chicken() {
  return egg();
}
```

```
function egg() {
  return chicken();
}
console.log(chicken() + " was first.");
```

In the call for the function there can be a different amount of call arguments, like when introducing a function. The extra arguments are rejected and the missing ones are replaced with *undefined*. The next example returns the square of the call argument regardless of extra call arguments:

```
function square(x) { return x * x; }
console.log(nelio(4, true, "onion"));
// → 16
```

undefined can also be used like this:

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}
console.log(minus(10));
// → -10
console.log(minus(10, 5));
// → 5
```

if the function *minus* calls for only one argument, the function returns it's opposite. If the call has two arguments, the function will return their difference.

If we give the argument a value after the = operator when introducing the function, the function will use it if an argument is missing when the function is called. In the following example, the function returns the value of the first parameter to the power of the next call parameter.

```
function power(base, exponent = 2) {
  var result = 1;
  for (var counter = 0; counter < exponent; counter++) {
    result *= base;
  }
  return result;
}
```

If the function *power* is missing it's second call parameter, the function returns the first parameter to it's next power.

```
console.log(power(4));
// → 16
```

The function can also be recursive, meaning it will call on itself like in this example:

```
function power(base, exponent) {
  if (exponent == 0) {
    return 1;
  } else {
    return base * power(base, exponent -1);
  }
}
```

In JavaScript implementations, the previous recursive function is three times as slow as the one done with a loop. However, using a recursive function is the smart thing to do in many cases.

A **Closure** is a programming technique, in which we create an inner function inside a function, with it's scope including the variables of the outer function. This isn't anything special as far as the outer function is in use and there's a reference to it from somewhere. The best parts of a closure will only be seen when there's no longer a reference to the outer function, as it doesn't exist anymore in practice, but there's still a reference to the inner function. In cases like this, despite the "disappearance" of the outer function the inner function can still see it's variables: a closure has formed around them, in which the inner function exists. To make the matter even more complicated, the same closure can contain many functions, which they can all see and handle the variables within the same closure.

In the next example the function *greet()* the defined greeting will be unaccessible by normal means after the function has been executed. There is no reference to it anywhere from outside, nor is there a reference to the *greet()* function itself. The function returned by the *greet()* function is saved into the variable hello, from where it can be called. In this situation the function *hello()* lives in the closure, where it can access the variable *greeting*.

```
function greet(name) {
  var greeting = "Hello, " + name;
  var helloToConsole = function() { console.log(greet); }
```

```
  return helloToConsole;
}
```

A closure is not function specific but call specific. Every time the outer function is called a new closure is formed, in which the inner function of that specific time lives on.

As we saw above, you can program functions without names as **anonymous functions**. In programming terms we call these **function literals**:

```
// function literal into a variable value
var sum = function(a,b) {return a+b}
console.log(sum(1,2));
othersum = sum;  // Copy the reference to the object function
console.log(othersum(3,4));
```

JavaScript has a neat way to write function literals as we can see in the following example:

```
var sum = (a,b) => a+b
console.log(sum(1,2));
othersum = sum;
console.log(othersum(3,4));
```

Functions in JavaScript are so called *first-class* values, which means that they can be placed into variables and into items in arrays like numerical values could. Functions can be used as parameters or be returned as function values etc.

Function literals are especially useful when forwarding functions as parameters for other functions. In the following example we give the formula for calculating the term as a parameter for a function calculating the sum of a sequence.

```
function sequenceSum(term, limit) {
  var sum = 0;
  for (var i=1; i<=limit; ++i)
    sum += term(i);
  return sum;
}

arithmetic = sequenceSum(i => i, 10);
console.log(arithmetic);  // 55

harmonic = sequenceSum(i => 1/i, 10);
console.log(harmonic);    //  2.9289682539682538
```

5. Javascript basics: objects and tables

5.1. Objects

JavaScript supports using Objects, even though it isn't a completely object-based programming language. To put it simply, an **object** can be thought as the parent or original copy of something. Objects have attributes, which are composed of an identifier and what it means, an attribute field or method. In JavaScript, these attributes are often called properties. An object is an associative array, which is a key-value pair. Keys are unique, and keys have a set value attached to them, which can be a field, a function, an object etc...

An object is created by writing an *object literal* into the program text. In JavaScript this is called an *object initializer*. The definition of an object begins with a curly bracket {, followed by a variable name and a value to give it. To set the value for an object variable happens with a colon, for example name:"Donna". More variables can be defined, separated by a comma. To stop defining the object you place the curly bracket } at the end. You can access the object variables with dot notation.

```
var trip =
  {distance: 150,
   time: 2,
   speed: function() {return this.trip/this.time}
  }
console.log(trip.speed());  // 75
```

When we're referring to an attribute of the object, we need the reference "to this object", which is *this*.

The next example uses variables defined outside of the object, and the returned value is different from the last:

```
var distance=20, time=10
var trip =
  {distance: 150,
```

```
   time: 2,
   speed: function() {return distance/time}
  }
console.log(trip.speed());  // 2 !!
```

In addition to dot notation, we can also refer to the attribute of the object with bracket notation: *name['attribute']*. With bracket notation we place the name of the attribute within the brackets as a string. This has two benefits. Firstly, if the name of the attribute contains characters like -,. or a space, it can't be used with dot notation. Secondly, bracket notation makes it possible to refer to an attribute, which has it's name saved in a variable. This makes using the object dynamic, which isn't possible with dot notation.

An object can also be created like this:

```
var trip = new Object();
trip.distance = 150;
trip.time = 2;
trip.speed = function() {return this.distance/this.speed}

console.log(trip.speed());  // 75
```

The third method for creating an object is to use a *constructor function*. Creating your own objects can be done with functions. Objects are instances of functions created with the keyword "new". With the attribute "this" we tell the program that the value of the variable being handled is tied to this specific object.

```
function traveling(distance, time) {
  this.distance = distance;
  this.time = time;
  this.speed = function() {return this.distance/this.time}
}

var trip = new traveling(150, 2);
console.log(trip.distance);     // 150
console.log(trip.speed());  // 75

var f1 = new traveling(150, 0.5);
console.log (f1.time);        // 0.5
console.log(f1.speed());     // 300
```

The above method of writing the accessors into the constructor function is bad: this way, each object gets its own copy of the *speed*-function!

It's better to attach the function for calculating speed into the *prototype object* of the *traveling*-function:

```
function Traveling(distance, time) {
  this.distance = distance;
  this.time = time;
}
Traveling.prototype.speed = function() {return this.distance/this.time}

var trip = new Traveling(150, 2);
console.log(trip.speed());           // 75

var f1 = new Traveling(150, 0.5);
console.log(f1.speed());              // 300
```

The prototype contains all the information related to the attributes of the function. The functions that are added with the prototype lets us access the objects *"this"*-reference, allowing for us to change the internal status of the object.

Many object-oriented programming language supports classes. JavaScript eventually got support for class as well, yet JavaScript objects function mostly the same way whether we use class or constructor functions. For example, there's still a prototype for objects even if they're made using the `class` keyword like in the example below. JavaScript's class should thus be simply considered an alternate method to write constructor functions and the prototype's methods. A class always requires the internal constructor method, `constructor`.

```
class Traveling {
  constructor(distance, time) {
    this.distance = distance;
    this.time = time;
  }
  speed() {
    return this.distance / this.time;
  }
}
```

```
var trip = new Traveling(150, 2);
console.log(trip.speed()); // 75

var f1 = new Traveling(150, 0.5);
console.log(f1.speed()); // 300
```

5.2. Arrays

An **array** is an object which is meant for the creation and handling of a group of items (which can also be called elements). An array is different from a normal object in the sense that it's keys are whole numbers and it has the *length*-attribute, which always tells how long it is (length is always one smaller than the largest index being used). Like with normal objects, values saved into an array can be any type, including functions.

Array variables can be created with the command [], which is short for *new array();*. For example we will create a new array with space for three items:

```
var passwords = ["password", "pawsword", "wordpass"];
```

The numbers that act as keys for the array are called *indexes* and the position a certain index occupies is marked with square brackets similarly as with objects.

There exist methods for arrays that handle them like a queue or a stack. This means adding or removing items to the start or end of the array. These operations change the length of the array.

The methods for handling an array are:

Method	Function	Returns	Example
push()	adds items to the end of the array	New array length	list.push('Jack')
pop()	Removes an item from the end of the array	Last array item	list.pop()
unshift()	adds items to the start of the array	new array length	list.unshift('Mickey')
shift()	Removes items from the start of the array	The first array item	list.shift()

We can add new values using the index or by using the methods *push* and *unshift*.

In the following example we will add the *push*-method to add a fourth item into the array:

```
var info = ["Minnie", 1983];
info[2] = "John";
info.push("Sara");

console.log(info[3]); // Sara
```

If new values are added into the array over it's current length, the length of the array grows and the indexes in the middle will be left empty, their value set to *undefined*. Normally, the *length*-value of the array will be the new index plus one.

```
var list = ['a', 'b', 'c'];
console.log(list.length);        // 3
list[9] = 'j';
console.log(list.length);        // 10
console.log(list);               // ['a', 'b', 'c',,,,,,,'j']
```

You can ask an array the position of a certain item within with the *indexof()*-method. If the item is in the array, this returns the first index where it's found. If the item is not within the array, the returned value is -1. So if the item is in the array multiple times, the returned value will only be the first index where it's found. The item to be found must be identical according to the operator ===. It isn't enough that it's similar, it needs to be the exact same one.

```
var list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'a', 'b', 'c',];
list.indexOf('c');          // 2
list.indexOf('h');          // -1

var values = [1, 2, 3, 0, 5, false, true];
values.indexOf(false);       // 5
```

You can copy parts from the middle of the array with the *slice()*-method. The syntax for it is: *array.slice(start, end)*. This method returns a new value, with the original arrays items from the index "start" until the index end minus 1 (so "end" is

the last array not included!). If the end-index is not specified, this copy will happen until the ened of the original array. Indexes can also be negative, in which case their position will be counted from the end of the array. The original array will not be changed by this.

You can remove or add items into the middle of an array with the *splice()*-method. It's syntax is: *array.splice(index, how many, item1, ..., itemX)*. *Index* tells from where in the array the operation starts. The parameter *how many* items from the array are removed, starting from *index*. The other parameters are items that will be added into the array starting from that location. This method returns a list of the removed items.

Two arrays can be combined together with the *concat()*-method. The method doesn't change the original array but returns a new array, in which the items from both arrays are placed after each other.

```
var list = [1, 2, 3, 4];
var array = ['a', 'b', 'c'];
var new = list.concat(array);     // [1, 2, 3, 4, 'a', 'b', 'c']
```

You can arrange arrays with the *sort()*-method. It rearranges the original array and returns the arranged array. By default *sort()* arranges the items as strings in alphabetical order.

```
var names = ['Emil', 'Celcius', 'Aaron', 'David','Bertha',  'Frank'];
names.sort();    // ["Aaron","Bertha","Celcius","David","Emil","Frank"]

var numbers = [6, 3, 1, 7, 10, 21];
numbers.sort();  // [1,10,21,3,6,7]
```

6. Javascript basics: Higher-order functions

6.1. Higher-order functions

Abstraction

Let's compare the following two functions.

```
var total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
```

And

```
console.log(sum(range(1, 10)));
```

They both accomplish the same task, but the latter is considerably simpler, which reduces the chance for programming errors. The term *Abstraction* is used for this method, hiding unnecessary details and focusing on the problem itself.

In addition to singular functions, abstraction can be used in loops as per the following example. In that the call of the function is another function.

```
function repeat(n, action) {
  for (var i = 0; i < n; i++) {
    action(i);
  }
}
repeat(3, console.log);
// → 0
// → 1
// → 2
```

Higher-order functions

Functions that have another function as their argument, or which return another function, are called Higher-order functions.

A function could create other functions, for example:

```
function greaterThan(n) {
  return m => m > n;
}
var greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));
// → true
```

A function can also alter another function:

```
function noisy(f) {
  return (...args) => {
    console.log("calling with", args);
    var result = f(...args);
    console.log("called with", args, ", returned", result);
    return result;
  };
}
noisy(Math.min)(3, 2, 1);
// → calling with [3, 2, 1]
// → called with [3, 2, 1] , returned 1
```

The next example clarifies how to use a higher-order function:

```
// The higher-order function not() returns
// a new function, which is brings to function f which,
// acts as it's arguments, and returns f's return value as it's negation
function not(f) {
  return function() { // Returns a new function
                      // which calls on f
    var result = f.apply(this, arguments);
    return !result;   // and returns the negation
  };
}
// Function to check if a number is even
var even = function(x) {
  return x % 2 === 0;
}
// Function that works the opposite of the above
var odd = not(even);
[1,1,3,5,5].every(odd); // => true: every item is odd
```

The most commonly used higher-order function in JavaScript is an arrays *forEach*-method, which takes a function as a parameter and runs it for each cell in the array.

The *map{}* function creates a new array, and it's cells are subjected to the given function:

```
var new_array = arr.map(function callback(currentValue[, index[, array]]) {
    // Return element for new_array
}[, thisArg])
```

The new array is the same length as the original, but it's items are delineated with the function. The following example describes how the *map{}*-function works:

```
const users = [
  {
    name: 'Jack',
    age: 25
  },
  {
    name: 'Matt',
    age: 30
  },
  {
    name: 'Bob',
    age: 1
  }
];

const newData = users.map( user => {
  return {
    ...user,
    onLapsi: user.age < 2 ? true: false
  }
});

console.log(newData);
```

The function *filter()* creates a new array, it's items being items that fill the criteria that the function sets for them:

```
var newArray = arr.filter(callback(element[, index[, array]])[, thisArg])
```

This example clarifies how *filter()* works:

```
const users = [
  {
    name: 'Jack',
    age: 25
  },
  {
    name: 'Matt',
    age: 30
  },
  {
    name: 'Bob',
    age: 1
  }
];

function isAdult(user) {
  return user.age >= 18;
}

const adultUsers = users.filter(isAdult);

console.log(adultUsers);
```

7. Javascript: object properties (encapsulation, methods, prototypes, classes, ...)

7.1. Objects, inheritance, classes

An **Object** can be thought to a common concept or a combination of information. An object can include different attributes which are saved into variables and methods which we can use to handle the information contained within the object. An object can be created with curly brackets like this:

```
var nora = {name: "Nora", Age: 35};
```

The attributes can be defined after creation like this:

```
var milly = {}; milly.name = "Milly"; milly.age = 18;
```

An object can also be defined by osung the object-definition:

```
var nora = new Object(); nora.name = "Nora"; nora.age = 35;
```

How we go about creating our object depends on what we're going to use the object for. The above examples are fast and can be used for simple one-use objects which have no common attributes.

The most popular way to create objects in JavaScript is to use a *constructor function*. The following is an example of defining an object with the constructor function and creating an instance of the object:

```
function Car(brand, model){
  this.brand = brand
  this.model = model
}

vehicle = new Car('Ford', 'Focus')
```

Constructor functions should be named starting with an uppercase letter to separate them from normal functions. A constructor function using the this-keyword cannot be called upon like a normal function since "*this*" would be referring to a global object, causing weird bugs.

Even though JavaScript doesn't have specific classes, it can be programmed very similarly by using prototypes. **Inheritance** reduces on the amount of code that needs to be copied. In JavaScript, inheritance between objects can be handled with prototypes. Using the functions *call*-functionality we can handle inheritance like class-based programming languages would:

```
function Car(brand, model){
  this.brand = brand
  this.model = model
}
function Truck(brand, model){
  Car.Call(this, brand, model);
this.gears = 6;
}
```

```
Truck.prototype = new Car();
vehicle = new Truck("GMC", "Bulldog");

console.log(vehicle.model); // "GMC"
console.log(vehicle.gears); // 6
```

The constructor function *Truck* calls on the constructor function *Car*, which give the attributes defined in Car to the *Truck-object*. Truck then builds upon this by adding a new attribute, *gears.*

In general the idea of inheritance is to create only the unique attributes for each object within the object, and using the "original object" otherwise. This way we won't need to copy as much code.

In the inheritance hierarchy, the prototype objects of functions can also contain accessor methods. If the inherited accessor sets a value into the inherited attribute, *this* will refer to the inheriting object. If the inherited object does not have it's own version of the attribute, it will be created. Here's an example detailing this:

```
function Animal(kg) {
  this.weight = kg || 0; // Unusable weight to zero
}
Animal.prototype.eats =
   function (amount) {this.weight+=amount;};
Animal.prototype.usage =
   function (amount) {this.weight-=amount;};

function Farmanimal(kg, liter) {
  Animal.call(this, kg);
  this.produce = litr || 0; // Unusable produce to 0
}
Farmanimal.prototype = new Animal();
Farmanimal.prototype.producing =
   function () {return this.produce;};

function Cow(name, kg, liter) {
  Farmanimal.call(this, kg, liter);
  this.name = name || "noname";   // Unusable name as default
}
Cow.prototype = new Farmanimal();
Cow.prototype.toString =
   function () { return this.name+
                 ": "+this.weight+" kg, "+
                  this.produce+" liters"
               };

var m = new Cow("Rosey", 560, 12);
console.log(m.tostring()); // Rosey: 500 kg, 12 liters
```

8. Javascript basics: regular expressions

8.1. Regular expressions

Regular expressions are patterns, which are used when searching and replacing for a combination of characters in a string. In Javascript, they're also objects.

Patterns are used in the RegExp *exec* and *test* methods, and in the String *match, matchAll, replace, search* and *split* methods.

Regular expressions are created by using a literal placed between strokes, or by using a constructor of the RegExp-function:

```
var re = /ab+c/;
```

```
var re = new RegExp('ab+c');
```

The former is defined when the script is loaded, and the latter while the script is being run.

In the common format, the definition is:

```
/pattern/modifiers;
```

Modifiers-switches are used to further clarify searches. They are:

```
i     The search does not separate uppercase or lowercase letters
g     All matching instances will be searched for
```

```
m     The search will be done over multiple lines
```

The most commonly used string-methods are *match()*, *search()* and *replace()*.

match() will search and return the given string:

```
var str = "Welcome to GEEKS for geeks!";
var res = str.match(/eek/1); // EEK
```

search() will find the given string and return it's location, as seen in this example:

```
var srt = "Visit MetSchools";
var n = str.search(/metaschools/i); // 6
```

replace() will replace the string with another. Here's an example on how it works:

```
var str = "Visit Microsoft!";
var res = str.replace(/microsoft/i, "MetSchools"); // Visit MetSchools
```

You can use the the following types of specifiers with patterns:

```
[abc]     Searching for any of the characters within the brackets
[^abc]    Searching for characters not within the brackets
[0-9]     Searching for any of the numbers within the brackets
[^0-9]    Searching for any of the numbers not within the brackets
(x|y)     Searching for for the choices divided with the | symbol
+         One or more, for example [0-9]+ could be 123, 000
*         Zero or more, for example [0-9]* could be empty, 123, 000
?         Zero or one
```

You can use the following metacharacters in patterns:

```
.      Searching for one symbol (not a line change/newline)
\n     Searching for a line change
\d     Search for a number
\s     Search for a whitespace
\b     Search from the beginning or end of a word
\uxxxx Search for the Unicode character xxxx
```

The following quantifiers can be used in patterns:

```
n+     Searching for a string with atleast one n
n*     Searching for a string with zero or more n
n?     Searching for a string with zero or one n
n{X}   Searching for a string with X times n
n{X,Y} Searching for a string with X-Y times n
n{X,}  Searching for a string with atleast X times n
n$     Searching for a string ending with n
^n     Searching for a string starting with n
?=n    Searching for a string that is followed by n
?!n    Searching for a string that is not followed by n
```

The following examples clarify how modifiers and patterns work:

```
// Searching for atleast one "o"
var str = "Hellooo World! Hello MetSchools";
var patt1 = /o+/g;
var result = str.match(patt1); // ooo,o,o,oo
```

```
// Searching for "1" followed by zero or more "o"
var str = "Hellooo World! Hello MetSchools!";
var patt1 = /lo*/g;
var result = str.match(patt1); // l,looo,l,l,lo,l
```

```
// Searching for "1" followed by zero or one "0"
var str = "1, 100 or 1000?";
var patt1 = /10?/g;
var result = str.match(patt1); // 1,10,10
```

The Regexp-method *test()* looks for a string and returns the value true if it is found:

```
var patt = /e/;
patt.test("The best things in life are free!"); // true
```

The previous program can be made even shorter:

```
/e/.test("The best things in life are free!"); // true
```

The Regexp-method *exec()* searches for a string and returns it as an object:

```
var obj = /e/.exec("The best things in life are free!");
```

The following example sheds some more light into the usage of Regexp:

```
<!DOCTYPE html>
<!-- JSRegexNumbers.html -->
<html lang="en">
<head>
<meta charset="utf-8">
<title>JavaScript Example: Regex</title>
<script>
var inStr = "abc123xyz456_7_00";

// RegExp.test(inStr) can be used to see if the pattern is included:
// Return true or false
console.log(/[0-9]+/.test(inStr));  // true

// String.search(regex) can be used to see if the template is included:
// Return starting index or -1
console.log(inStr.search(/[0-9]+/));  // 3

// String.match():lla and RegExp.exec():lla are used to find
// subgroup, reference and index
console.log(inStr.match(/[0-9]+/));
  // ["123", input:"abc123xyz456_7_00", index:3, length:"1"]
console.log(/[0-9]+/.exec(inStr));
  // ["123", input:"abc123xyz456_7_00", index:3, length:"1"]

// Let's use the g (global) modifier
console.log(inStr.match(/[0-9]+/g));
  // ["123", "456", "7", "00", length:4]
// RegExp.exec() repeated with the g-modifier:
// The search is repeated until the last match is found.
// Placement into RegExp.lastIndex.
var pattern = /[0-9]+/g;
var result;
while (result = pattern.exec(inStr)) {
  console.log(result);
  console.log(pattern.lastIndex);
    // ["123"],  6
    // ["456"], 12
    // ["7"],   14
    // ["00"],  17
}

// String.replace(regex, replacement):
console.log(inStr.replace(/\d+/, "**"));
  // abc**xyz456_7_00
console.log(inStr.replace(/\d+/g, "**"));
  // abc**xyz**_**_**
</script>
</head>
<body>
  <h1>Hello,</h1>
</body>
</html>
```

9. Javascript basics: Modules

9.1. Modules

Modules are done with anonymous functions. Due to the visibility of variables in functions, variables can be encapsuled within an anonymous function, so they cannot be accessed from outside the function. Functions defined inside an anonymous function can access variables, so the values of the variables can be altered. The module will return the interface defined within the module with all the references to the inner functions.

The following example will shed light onto how to use modules. We will build a system needed for organizing a store. We will create a shopping cart module, which offers a public interface for adding products and calculating their number.

```
var store = {};
```

```
shop.cart = (function() {
  var products = [];
  function addProduct(productName) {
    if(!products[productName]) {
      // If the product is not added into the shopping cart, add it there.
         products[Productname] = 0;
    }
    // Raise the number of products by one
    products[productName]++;
  }

  function totalProducts() {
    var amount = 0;
    for(var productName in products) {
      amount += products[Productname];
    }
    return amount;
  }

  // interface
  return {
    add: addProduct,
    productAmount: productTotal
  };
})();
```

The shopping cart is now used like this:

```
store.cart.add("keksi");
store.cart.add("keksi");
store.cart.add("apple");
console.log(store.cart.productAmount()); // 3
```

10. Javascript basics: Asynchronic programming

10.1. Asynchronic programming

There is a huge difference between programming web applications and desktop programs. Unlike in a desktop environment, web browsers only have one thread for everything that requires access to the interface. Single threading limits the program code altering elements of the interface, as it limits the execution of the rest of the code. In other words, functions and threads stop the interface from being used until the thread is completed. This is why it is important to take advantage of all the asynchronic functions available in a browser.

The **synchronic** execution of a program means that every line of code is executed in order, and the next line cannot be executed until the one before it is completed. When functions are called the execution of the program will wait for the function to be returned from so that the execution can be continued from the next line. If the function uses actions that take time such as retrieving data from a server, the main program will wait all this time before it continues executing the code.

With **asynchronic** execution the program does not wait for a function to return, but instead immediately continue executing the main program.

JavaScript uses an event-driven model when executing programs with one thread. In this model, the *event loop* handles tasks from the task queue one at a time, making a call stack for each task being in execution. Only when the stack is fully resolved, will there be a new task taken from the queue.

When asynchronic tasks are being handled in the stack, they are moved into the browser's API to either be timed or, in the case of retrieving data from the server, to wait for the server to answer. From there, they will be added into the task queue when the task is completed. The following example shows how this event-driven model works in JavaScript:

```
function main() {
  function firstFunction() {
    setTimeout(function cb() {
      secondFunction();
    }, 0}
  }
  function secondFunction() {
    console.log('Im second function');
  }
  firstFunction();
  console.log('Done');
```

```
}
main();
```

When the code in the example is being executed, the call stack first takes the function call *main()* from the last line. After that, the function call *FirstFunction()* and from within that *setTimeout cb*. When the *setTimeout*-call is being run, it moves as an asynchronic action to the browser API for the duration of the timing, from where the callback-function cb will move to the back of the task queue when the time is up (in this time the given time was 0, so it moves there immediately). The next task in the queue can only be executed once the task being run currently has been handled.

When *setTimeout cb* is removed from the call stack, *firstFunction()* can be removed as well, since the function has now finished it's execution. Next the call stack will handle *console.log('Done')*, which is executed immediately. The console will print Done and can be removed from the stack. The Function *main()* is executed and can be removed from the call stack.

The stack is executed, so the next task from the queue will be taken into execution. This means, in order, *cb(), secondFunction()* and *console.log('Im second Function')*.

Finally *console.log* is executed, printing *Im second function* and the call stack will be emptied one at a time from the top down, since all the tasks have been completed. As you can deduct from the example, the *setTimeout*-method doesn't determine that the callback-function will be done after a certain time we've given as a parameter, but rather it will be moved to the back of the task queue after a certain amount of time.

The **Callback**-function is given as a parameter to another function. It will either be run immediately before the function is returned, or after returning from the function. If it's run immediately, it's a synchronic callback, like in the *forEach*-method of a array where the same code block is executed for each cell in the array. An asynchronic callback will be run in the future, for example when a HTTP request gets a response from the server.

11. JavaScript basics: ES6 and ES7

11.1. Ecmascript (ES)

Ecmascript is the standard that JavaScript is based on. The first edition of the Ecmascript was released in 1997. The ES6 (also known as Ecmascript 2015) was released in 2015 and it introduced lot of nice features to JavaScript. It is now much easier to write big and complex programs with the language. While ES6 was a huge update after a long time period of no changes to JavaScript at all, there have since been more smaller updates. In this chapter we will go through the most important features of these updates, such as class declarations and the arrow functions.

11.2. let & const

JavaScript has a keyword `let` for defining variables. The `var` keyword was already described earlier. The scope of the `var` variables is function scope, which means that variable can be only used inside the function where it was defined.

For example, in the following code, the variables `a` and `b` can be used only inside the `myFunction` function.

```
function myFunction() {
  var a = 5;
  var b = a * 10;
}
```

Now, if you try to print variable `a` value to the console like shown in the code below.

```
function myFunction() {
  var a = 5;
  var b = a * 10;
}

console.log(a);
```

You will get the following error, because you can use variable `a` only inside the `myFunction`.

```
ReferenceError: a is not defined
```

If you define the `var` variable outside of any function, it can be used everywhere in your javascript file. Then it is so called global variable in you javascript file. That is always dangerous and can cause unexcpected errors therefore you should always use as narrow scope as it is possible.

The scope of the `let` variable is block scope and then variable can be used only inside the block where it was defined. In the following code we have the same function as earlier but now the variables are defined using the let keyword. Nothing is actually changing because the variables can be used only inside the function block.

```
function myFunction() {
  let a = 5;
  let b = a * 10;
}
```

But, if we have for example `if` statement inside the function and variables are defined there. Then, the variables can be used only inside the `if` block. In the following code, the variable `b` can be used only inside the `if` block where it was defined.

```
function myFunction() {
  let a = 5;
  if (a > 3) {
    let b = a * 10;
  }
  // variable a can be used here
  // variable b canno't be used here
}
```

With the `let` keyword you can define more narrow scope than using the `var` keyword. Therefore, you should always use `let` instead of `var` if it is possible because it makes your code easier to maintain.

The `const` keyword can be used to define constants. The scope of the `const` is also block scoped. In the following example, we define constant called PI.

```
const PI = 3.14159
```

The constant values are immutable and you will get the following error if you try to change these values.

```
TypeError: Assignment to constant variable.
```

11.3. Arrow functions

Arrow functions are an alternative way to define functions in JavaScript. The basic idea of the arrow function expression is the following. On the left side of the arrow (=>) are function parameters. On the right side of the arrow is a function expression.

```
parameters => expression
```

If we have the following anonymous function.

```
function() {
  return "I am a function";
}
```

With the arrow function it can be written like the code below.

```
() => "I am a function"
```

As you can see, it gets much shorter. If there isn't a single parameter, you must use empty brackets like shown in the example above.

If you have only one function argument, you don't need brackets. In the next example, one parameter is passed to the function and it returns the parameter value plus ten.

```
x => x + 10
```

If there are multiple parameters, the brackets are needed as shown in the example below.

```
(x, y) => x * y
```

You can have multiple lines in the arrow function and the you have to define function block using the curly braces and the `return` keyword.

```
(x, y) => {
  let a = 10;
  return x + y + a;
}
```

All functions above are anonymous functions that you can't call. These are mostly used as a callback functions. The callback function is a function that is passed into another function as a parameter. Let's go through one practical example. Javascript

function `map()` is really handy for array iteration. The `map()` function accepts callback function as a parameter and it then calls a function on each item in an array. The following example demonstrates the usage of the `map()` function. The callback function is `x => x * 2` and then the all `arrayA` elements are multiplied by two. The callback function runs for each value in the array and return new values in the resulting array (`arrayB`).

```
let arrayA = [1, 2, 3, 4, 5];

// arrayB = [2, 4, 6, 8, 10]
let arrayB = arrayA.map(x => x * 2);
```

This can be also written with the traditional function in the following way.

```
let arrayA = [1, 2, 3, 4, 5];

// arrayB = [2, 4, 6, 8, 10]
let arrayB = arrayA.map(function(x) {
  return x * 2;
});
```

Now, you can see that you can get much shorter and readable code by using the arrow functions.

You can also save an arrow function to a variable and then call it like shown in the code below.

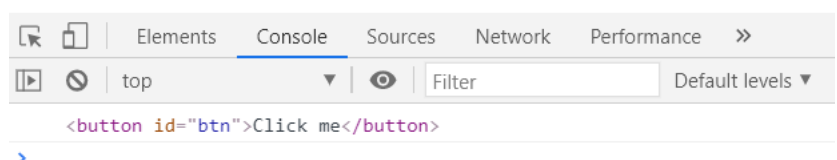
```
const sayHello = name => console.log("Hello " + name)

// Call function
sayHello("John");
```

It is very important to know that `this` keyword behaves differently with arrow functions compared to traditional functions. In the arrow functions there is no bindings to the `this` keyword. In the traditional functions this keyword is binded to the object that called the function. The following example demonstrates the difference.

We have the following HTML file where we have one button. The button invokes `hello` function when it is pressed. The hello function prints `this` object to the console. In the first case the function is declared using the traditional function notation.

```
<!doctype html>

<html lang="en">
<head>
  <meta charset="utf-8">
</head>

<body>
  <button id="btn">Click me</button>

  <script>
    function hello() {
      console.log(this);
    }

    document.getElementById("btn").addEventListener("click", hello);
  </script>
</body>
</html>
```

Now, if we press the button, we can see the button in the console like shown in the image below. So, the `this` keyword is currently bound to the button.

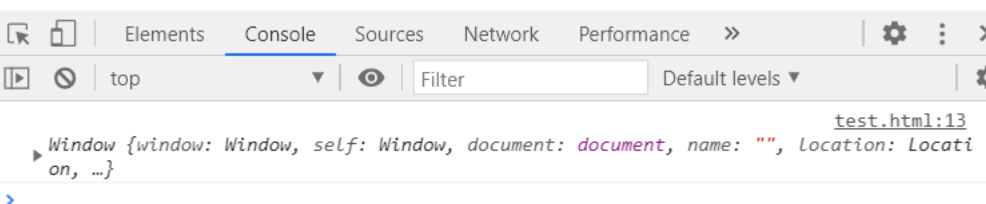


Next, we will replace the traditional function with an arrow function like shown in the code below.

```
<!doctype html>

<html lang="en">
<head>
  <meta charset="utf-8">
</head>

<body>
  <button id="btn">Click me</button>

  <script>
    hello = () => {
      console.log(this);
    }

    document.getElementById("btn").addEventListener("click", hello);
  </script>
</body>
</html>
```

Now, if we press the button, we can see that `this` is now bound to the window object like shown in the image below.

Click me



The arrow functions are excellent in the callback functions, but you have to be aware of this difference when using these in other purposes.

11.4. Function parameters

Functions can have default parameter values like shown in the code below.

```
function calcSum(x, y = 5) {
  return x + y;
}
```

Now, if we call the function above using the statement `calcSum(2)`, it returns 7 because x = 2 and y = 5. Next, if we call the function using the statement `calcSum(2, 3)`, it returns 5 because x = 2 and y = 3. If you don't pass value to the y, the function uses the default value.

You can also use rest notation (...) in the function parameters and then a function can accept infinite number of parameters as an array. For example the following code takes parameter called `params` using the rest notation. In the function body, we iterate the `params` array and print all values to the console.

```
function myFunction(...params) {
  params.forEach(param => console.log(param));
}
```

Now, you can call the `myFunction` by using as many parameters as you want.

```
myFunction("Hello", "World", "John");
```

In this case, the output is the following.

```
Hello
World
John
```

11.5. Classes & inheritance

The `class` keyword can be used to define objects in ES6. The usage is similar to other object oriented languages, like Java. The code below creates a class that is named Shape.

```
class Shape {
  // Class code
}
```

The class have a method called `constructor`. That is invoked when the new object is created from the class. The class can also have class properties and methods like shown in the example below.

```
class Shape {
  // Constructor where x and y are class properties
  constructor (x, y) {
    this.x = x;
    this.y = y;
  }

  // Class method
  move(x, y) {
    this. x = x;
    this.y = y;
  }

  printLocation() {
    console.log(this.x + ", " + this.y);
  }
}
```

You can create class objects by using the `new` keyword. You can call class method by using the `object_name.method_name` notation like shown in the code below.

```
class Shape {
// Constructor where x and y are class properties
  constructor (x, y) {
    this.x = x;
    this.y = y;
  }

  // Class methods
  move(x, y) {
    this. x = x;
    this.y = y;
  }

  printLocation() {
    console.log(this.x + ", " + this.y);
  }
}

// Create new object from the Shape class
let shape = new Shape(0, 0);

// Call class methods
shape.move(10, 10);
shape.printLocation(); // prints 10, 10 to console
```

The class inheritance can be implemented by using the `extends` keyword. Let's first create one class that is named Person.

```
class Person {
  constructor(first_name, last_name) {
    this.last_name = last_name;
    this.first_name = first_name;
  }

  printName() {
   console.log(first_name + " " + last_name);
  }
}
```

Next, we will create class called Student that inherits the Person class.

```
class Student extends Person {
  constructor(first_name, last_name, student_nr, departement) {
```

```
    super(first_name, last_name);
    this.student_nr = student_nr;
    this.department = department;
  }
}
```

The Student (=subclass) class inherits all properties and methods from the Person (=superclass) class. In the constructor of the Student class, we have to call superclass constructor using the `super` keyword. Now, you can create a new student object like shown in the code below.

```
let student = new Student("John", "Johnson", S23324, "Computer Science");
```

In practice, the JavaScript class specifically made using the class keyword does not really differ from the result we would have gotten using the previously discussed constructor functions and prototypes. It can be considered an alternative way for creating JavaScript classes, especially for those already used to classes in other proper object-oriented programming languages.

11.6. Template literals

Template literals can be used to concatenate strings. As a reminder, the traditional way would be to use plus signs for the concatenation like shown in the example below.

```
let first_name = "John";
let last_name = "Johnson";
console.log("Hello " + first_name + " " + last_name);
```

Template literals can be used to make string concatenation more intuitive. Template literals can contain placeholders for string substitution `${ }`, that contain JavaScript expressions. Now, the example above can be written in the following way. **Note!** With the template literal you must use backticks (``) instead of quotation marks.

```
let first_name = "John";
let last_name = "Johnson";
console.log(`Hello ${first_name} ${last_name}`);
```

The following code prints the sum of two variables.

```
let x = 5;
let y = 8;
console.log(`Sum = ${x + y}`);
```

You can also call functions using the template strings like shown in the following example. It prints *Hello World John* to the console.

```
function hello() {
  return "Hello World";
}

console.log(`${hello()} John`)
```

11.7. Exponentiation operator

The Math library already contained the method Math.pow(base, power) . Later on, the operator ** was added to JavaScript, considered far faster and more intuitive to use. As with other common mathematical operators, the exponentiation also has a shortened version for assignment, **=.

```
x = 2 ** 2      // 2 to the power of 2 is 4
x **= 2         // 4 to the power of 2 is 16
console.log(x)  // 16
```

The only real difference between Math.pow() and the ** operator is that some ancient browsers may not support the latter. Using the exponentiation operator is recommended as it's more clear and short and allows the aforementioned **= operation as well.

11.8. Includes

There already exists a method for checking where the specified value is found on an array.

```
someList = ["first val", "second val", "third val"];
console.log(someList.indexOf("third val");      // 2
console.log(someList.indexOf("some other val"); // -1
```

.indexOf() returns the index of said value, or in case it's not found, -1. Later on .includes() was added as well, which returns either a true or false depending on whether the specified value was found. There isn't a big difference between the two, as .indexOf() can be used to get a boolean value with the addition of a simple if-statement.

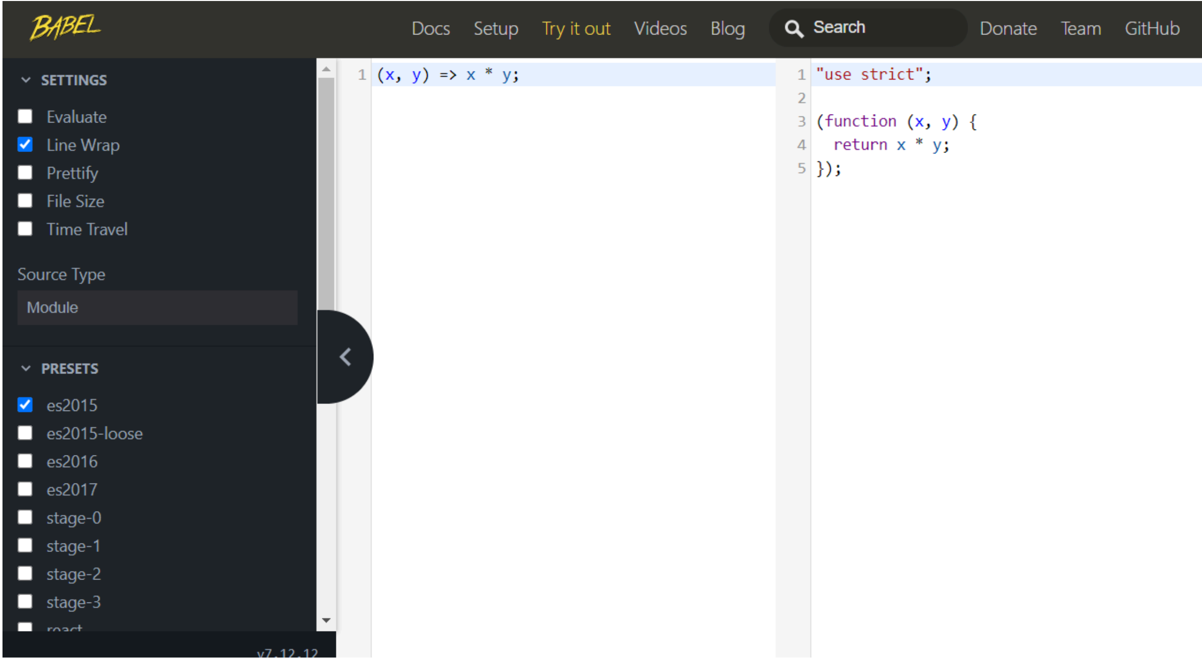
.includes() may make more sense to use, however. Additionally, it can recognize a NaN value on an array, something which .indexOf() is unable to do.

```
anotherList = [1, 2, NaN, 3, 4];
anotherList.indexOf(NaN);  // -1
anotherList.includes(NaN); // true
```

11.9. Babel

You have to be careful with the latest Ecmascript features because some browsers might not support the latest features. Babel (https://babeljs.io/) is the javascript compiler that can be used to compile javascript to older browser compatible version.

You can test the Babel in practivce by navigating to their web site and selection 'Try it out' from the top menu. In the following image you can see how the ES6 arrow function will be compiled back to the traditional function.



You can use Babel in the browser by importing Babel library and defining script type to `text/babel` like shown in the following example. Now, you can use the latest Ecmascript features and these are compiled back to older javascript version.

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>

<!-- Your custom script here -->
<script type="text/babel">
  const getMessage = () => "Hello World";
</script>
```