

JS

JS



# What is JS?

- ▶ JavaScript is a programming language that allows you **to make web pages interactive**. Where HTML and CSS are languages that give structure and style to web pages, JavaScript gives web pages interactive elements that engage a user.

## Focus points:

- Variables
- Data Types
- Conditionals
- Developer Tools
- Functions

# How to run JavaScript code?

All JavaScript we will be writing, in most of this course, will be run via the browser. The simplest way to get started is to simply create an HTML file with the JavaScript code inside of it. Type the basic HTML skeleton into a file on your computer somewhere:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Page Title</title>
</head>
<body>
  <script>
    // Your JavaScript goes here!
    console.log("Hello, World!")
  </script>
</body>
</html>
```

***Note:** `console.log()` is the command to print something to the developer console in your browser. You can use this to print the results from any of the following articles and exercises to the console. We encourage you to code along with all of the examples in this and future lessons.*

# How to run JavaScript code?

Another way to include JavaScript in a web page is through an external script. This is very similar to linking external CSS docs to your website. JavaScript files have the extension **.js** like **.css** for stylesheets. External JavaScript files are used for more complex scripts.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Page Title</title>
</head>
<body>
  <script src="script.js"></script>
</body>
</html>
```

```
'red'; // global scope
```

```
) => {
```

```
color2 = 'blue'; // local (function) scope
```

```
console.log(color1);
```

```
console.log(color2);
```

```
console.log(color1); // 'red'
```

```
console.log(color2); // ReferenceError: color2 is not defined
```

```
var color1 = 'red'; // function scope
let color2 = 'blue'; // block scope
const color3 = 'yellow'; // block scope
console.log(color1);
console.log(color2);
console.log(color3);
}

console.log(color1);
console.log(color2);
console.log(color3);
}

addColor();
// 'red'
// 'blue'
// 'yellow'
// 'red'
// ReferenceError: color2 is not defined
// ReferenceError: color3 is not defined
console.log(color1); // ReferenceError: color1 is not defined
```

# Variables

JAVASCRIPT

# Variables

- JavaScript variables are containers for storing data values. Here are some of the types we will cover:
  - **Numbers**
  - **Strings**
  - **Boolean**
  - **Null**
  - **Undefined**
  - **Objects**

# Variables

You can think of variables as simply “**storage containers**” for data in your code. Until recently there was only one way to create a variable in JavaScript — the **var** statement. But in the newest JavaScript versions we have two more ways — **let** and **const**.

**let** and **const** are both relatively new ways to declare variables in JavaScript. In many tutorials (and code) across the internet you’re likely to encounter **var** statements. The main difference between **let** and **var** is that scope of a variable defined with **let** is limited to the block in which it is declared while variable declared with **var** has the global scope. So, we can say that **var** is rather a keyword which defines a variable globally regardless of block scope.



# Variables

The statement below creates (in other words **declares**) a variable with the name “message”:

```
let message;
```

Now, we can put some data into it by using the assignment operator =

```
let message;  
message = 'Hello'; // store the string 'Hello' in the variable named message
```

The string is now saved into the memory area associated with the variable. We can access it using the variable name:

```
let message;  
message = 'Hello!';  
alert(message); // shows the variable content
```

To be concise, we can combine the variable declaration and assignment into a single line:

```
let message = 'Hello!'; // define the variable and assign the value  
alert(message); // Hello!
```

# Variables

We can also declare multiple variables in one line which might seem shorter, but we don't recommend it. For the sake of better readability, please use a single line per variable:

```
let user = 'John', age = 25, message = 'Hello';
```

The multiline variant is a bit longer, but easier to read:

```
let user = 'John';  
let age = 25;  
let message = 'Hello';
```

Some people also define multiple variables in these multiline styles:

```
let user = 'John',  
    age = 25,  
    message = 'Hello';
```

```
let user = 'John'  
    , age = 25  
    , message = 'Hello';
```

# Variables

We can easily grasp the concept of a “**variable**” if we imagine it as a “**box**” for data, with a uniquely-named sticker on it. For instance, the variable message can be imagined as a box labeled "message" with the value "Hello!" in it. We can put any value in the box and we can also change it as many times as we want:

```
let message;  
message = 'Hello!';  
message = 'World!'; // value changed  
alert(message);
```

When the value is changed, the old data is removed from the variable.

```
let hello = 'Hello world!';  
let message;  
  
// copy 'Hello world' from hello into message  
message = hello;  
  
// now two variables hold the same data  
alert(hello); // Hello world!  
alert(message); // Hello world!
```

We can also declare two variables and copy data from one into the other.

# Variables - naming

There are two limitations on variable names in JavaScript:

1. The name must contain only letters, digits, or the symbols \$ and \_
2. The first character must not be a digit.

Examples of valid names:

```
let userName;  
let test123;
```

When the name contains multiple words, camelCase is commonly used. That is: words go one after another, each word except first starting with a capital letter: ***myVeryLongName***.

# Variables - naming

The dollar sign '\$' and the underscore '\_' can also be used in names. They are regular symbols, just like letters, without any special meaning. So, these names are valid:

```
let $ = 1; // declared a variable with the name "$"  
let _ = 2; // and now a variable with the name "_"  
alert($ + _); // 3
```

Examples of incorrect variable names:

```
let 1a; // cannot start with a digit  
let my-name; // hyphens '-' aren't allowed in the name
```

⚠ **Important note:** Variables are case-sensitive, so variables named *chocolate* and *CHOCOLATE* and *cHocOIaTe* are not the same.

## Variables - naming

Non-Latin letters are allowed, but not recommended. It is possible to use any language, including Cyrillic letters or even hieroglyphs, like this:

```
let имя = '...';  
let 我 = '...';
```

Technically, there is no error here. Such names are allowed, but there is an international convention to use English in variable names. Even if we're writing a small script, it may have a long life ahead. People from other countries may need to read it some time.

## Variables - naming

There is a [list of reserved words](#), which cannot be used as variable names because they are used by the language itself. For example: *var*, *let*, *class*, *return*, and *function* are reserved. The code below gives a syntax error:

```
var var = 5; // can't name a variable "var", error!  
let return = 5; // also can't name it "return", error!
```

# Variables - constants

To declare a constant (unchanging) variable, use **const** instead of **let**:

```
const myBirthday = '01.01.1991';
```

Variables declared using `const` are called “constants”. They cannot be reassigned. An attempt to do so would cause an error:

```
const myBirthday = '01.01.1991';  
myBirthday = '01.01.2001'; // error, can't reassign the constant!
```

When a programmer is sure that a variable will never change, they can declare it with `const` to guarantee and clearly communicate that fact to everyone.



## Variables – uppercase constants

There is a widespread practice to use constants as aliases for difficult-to-remember values that are known prior to execution. Such constants are named using capital letters and underscores. For instance, let's make constants for colors in so-called “web” (hexadecimal) format:

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// ...when we need to pick a color
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Benefits:

- **COLOR\_ORANGE** is much easier to remember than **"#FF7F00"**.
- It is much easier to mistype **"#FF7F00"** than **COLOR\_ORANGE**.
- When reading the code, **COLOR\_ORANGE** is much more meaningful than **#FF7F00**.

## Variables – naming conventions

A variable name should have a clean, obvious meaning, describing the data that it stores. Variable naming is one of the most important and complex skills in programming. A quick glance at variable names can reveal which code was written by a beginner versus an experienced developer.

In a real project, most of the time is spent modifying and extending an existing code base rather than writing something completely separate from scratch. When we return to some code after doing something else for a while, it's much easier to find information that is well-labeled. Or, in other words, when the variables have good names.

# Variables – naming conventions

Some good-to-follow rules are:

- Use human-readable names like **userName** or **shoppingCart**.
- Stay away from abbreviations or short names like a, b, c, unless you really know what you're doing.
- Make names maximally descriptive and concise. Examples of bad names are **data** and **value**. Such names say nothing. It's only okay to use them if the context of the code makes it exceptionally obvious which data or value the variable is referencing.
- Agree on terms within your team and in your own mind. If a site visitor is called a “user” then we should name related variables **currentUser** or **newUser** instead of **currentVisitor** or **newManInTown**.

# Data Types

```
typeof("Hola") // string
typeof(12) // integer
typeof(true) // boolean
typeof(undefined) // undefined
typeof(null) // object
typeof({}) // object
typeof(Symbol()) // symbol
typeof(1n) // bigint
typeof(function(){} ) // function
```

## Data Types

JAVASCRIPT

# Data Types

A value in JavaScript is always of a certain type. For example, a **string** or a **number**.

There are **eight** basic data types in JavaScript.

We can put any type in a variable. For example, a variable can at one moment be a string and then store a number:

```
// no error  
let message = "hello";  
message = 123456;
```

Programming languages that allow such things, such as JavaScript, are called “dynamically typed”, meaning that there exist data types, but variables are not bound to any of them.

# Numbers

Numbers are the building blocks of programming logic! In fact, it's hard to think of any useful programming task that doesn't involve at least a little basic math. So, knowing how numbers work is obviously quite important. Luckily, it's also fairly straightforward.

[This W3Schools lesson](#) followed by [this one](#), are good introductions to what you can accomplish with numbers in JavaScript.

[This MDN article](#) covers the same info from a slightly different point of view, while also teaching you how to apply some basic math in JavaScript. There's much more that you can do with numbers, but this is all you need at the moment.

Read through (and code along with!) [this article](#) about operators in JavaScript. It will give you a pretty good idea of what you can accomplish with numbers in JavaScript.

# Numbers

The number type represents both **integer** and **floating point** numbers.

```
let n = 123;  
n = 12.345;
```

There are many operations for numbers such as:

```
n = 2 + 2; // addition +  
n = 3 - 5; // subtraction -  
n = 2 * 3; // multiplication *  
n = 1 / 2; // division /  
n = 4 ** 2; // exponents **
```

# Numbers

Besides regular numbers, there are so-called “special numeric values” which also belong to this data type: **Infinity**, **-Infinity** and **NaN**. **Infinity** represents the mathematical infinity. It is a special value that is bigger than any other number. We can get it as a result of division by zero:

```
alert( 1/0 ); // Infinity

// We can also reference it directly:
alert( Infinity ); // Infinity
alert( -Infinity ); // -Infinity
```



# Numbers

**NaN** represents a computational error. It is a result of an incorrect or an undefined mathematical operation, for instance:

```
alert ( "not a number" / 2 ); // returns NaN  
alert( NaN + 1 ); // NaN  
alert( 3 * NaN ); // NaN
```

## Numbers - BigInt

In JavaScript, the “number” type cannot represent integer values larger than  $(2^{53}-1)$  (that’s 9007199254740991), or less than  $-(2^{53}-1)$  for negatives. It’s a technical limitation caused by their internal representation.

For most purposes that’s quite enough, but sometimes we need really big numbers, e.g. for cryptography.

BigInt type was recently added to the language to represent integers of arbitrary length.

A BigInt value is created by appending n to the end of an integer:

```
// the "n" at the end means it's a BigInt  
const bigInt = 1234567890123456789012345678901234567890n;
```

# Strings

A string is a sequence of one or more characters that may consist of letters, numbers, or symbols. A string in JavaScript must be surrounded by quotes.

In JavaScript, there are 3 types of quotes:

1. Double quotes: "Hello".
2. Single quotes: 'Hello'.
3. Backticks: `Hello`.

```
let str = "Hello";  
let str2 = 'Single quotes are ok too';  
let phrase = `can embed another ${str}`;
```

Double and single quotes are “simple” quotes. There’s practically no difference between them in JavaScript.

# Strings

Backticks are “extended functionality” quotes. They allow us to embed variables and expressions into a string by wrapping them in `${...}`, for example:


```
let name = "John";

// embed a variable
alert( `Hello, ${name}!` ); // Hello, John!

// embed an expression
alert( `the result is ${1 + 2}` ); // the result is 3
```

```
// the result is ${1 + 2} (double quotes do nothing)
alert( "the result is ${1 + 2}" );
```

The expression inside `${...}` is evaluated and the result becomes a part of the string. We can put anything in there: a variable like `name` or an arithmetical expression like `1 + 2` or something more complex.

 **Important note:** This can only be done in backticks. Other quotes don't have this embedding functionality!

# Strings

There is very little difference between the single and double quotes, and which you use is down to personal preference. You should choose one and stick to it, however; differently quoted code can be confusing, especially if you use two different quotes on the same string! The following will return an error:

```
const badQuotes = 'What on earth?';
```

The browser will think the string has not been closed because the other type of quote you are not using to contain your strings can appear in the string. For example, both of these are okay:

```
const sglDb1 = 'Would you eat a "fish supper"?';  
const dblSgl = "I'm feeling blue.";
```

# Strings

However, you can't include the same quote mark inside the string if it's being used to contain them. The following will error, as it confuses the browser as to where the string ends:

```
const bigmouth = 'I've got no right to take my place...';
```

To fix our previous problem code line, we need to escape the problem quote mark. Escaping characters means that we do something to them to make sure they are recognized as text, not part of the code. In JavaScript, we do this by putting a backslash just before the character:

```
const bigmouth = 'I\'ve got no right to take my place...';  
console.log(bigmouth);
```

See [escape sequences](#) for more details.

# Booleans

The boolean type has only two values: **true** and **false**. This type is commonly used to store yes/no values: **true** means “yes, correct”, and **false** means “no, incorrect”. For instance:

```
let nameFieldChecked = true; // yes, name field is checked
let ageFieldChecked = false; // no, age field is not checked
```

Boolean values also come as a result of comparisons:

```
let isGreater = 4 > 1;
alert( isGreater ); // true (the comparison result is "yes")
```

# Null

The special **null** value does not belong to any of the types described above. It forms a separate type of its own which contains only the null value:

```
let age = null;
```

In JavaScript, null is not a “reference to a non-existing object” or a “null pointer” like in some other languages. It’s just a special value which represents “nothing”, “empty” or “value unknown”. So the code above states that age is unknown.



# Undefined

The special value **undefined** also stands apart. It makes a type of its own, just like **null**. The meaning of undefined is “value is not assigned”. If a variable is declared, but not assigned, then its value is undefined:

```
let age;  
alert(age); // shows "undefined"
```

Technically, it is possible to explicitly assign undefined to a variable:

```
let age = 100;  
// change the value to undefined  
age = undefined;  
alert(age); // "undefined"
```

**Note:** We don't recommend doing that. Normally, one uses **null** to assign an “empty” or “unknown” value to a variable, while undefined is reserved as a default initial value for unassigned things.

# Objects

The object type is special. All other types are called “primitive” because their values can contain only a single thing (be it a string or a number or whatever). In contrast, objects are used to store collections of data and more complex entities.

In JavaScript, objects penetrate almost every aspect of the language. So we must understand them first before going in-depth anywhere else. An object can be created with figure brackets {...} with an optional list of properties. A property is a “key: value” pair, where key is a string (also called a “property name”), and value can be anything.

An empty object can be created using one of two syntaxes:

```
let user = new Object(); // "object constructor" syntax
```

```
let user = {}; // "object literal" syntax
```

Usually, the figure brackets {...} are used. That declaration is called an object literal. [Here's](#) the difference between the two.

# Objects

We can immediately put some properties into `{...}` as “key: value” pairs:

```
let user = {  
  // an object  
  name: "John", // by key "name" store value "John"  
  age: 30, // by key "age" store value 30  
};
```

A property has a key (also known as “name” or “identifier”) before the colon “:” and a value to the right of it. In the user object, there are two properties:

1. The first property has the name “name” and the value “John”.
2. The second one has the name “age” and the value 30.

# Objects

We can add, remove and read files from it at any time. Property values are accessible using the dot notation:

```
// get property values of the object:  
alert( user.name ); // John  
alert( user.age ); // 30
```

The value can be of any type. Let's add a boolean one:

```
user.isAdmin = true;
```

To remove a property, we can use the **delete** operator:

```
delete user.age;
```

# Objects

We can also use multiword property names, but then they must be quoted:

```
let user = {  
  name: "John",  
  age: 30,  
  "likes birds": true, // multiword property name must be quoted  
};
```

The last property in the list may end with a comma:

```
let user = {  
  name: "John",  
  age: 30,  
};
```

That is called a “trailing” or “hanging” comma. Makes it easier to add/remove/move around properties, because all lines become alike.

# Objects

For multiword properties, the dot access doesn't work:

```
// this would give a syntax error
user.likes birds = true
```

JavaScript doesn't understand that. It thinks that we address `user.likes`, and then gives a syntax error when it comes across unexpected `birds`. The dot requires the key to be a valid variable identifier. That implies: contains no spaces, doesn't start with a digit and doesn't include special characters (`$` and `_` are allowed).

There's an alternative "square bracket notation" that works with any string:

```
let user = {};  
// set  
user["likes birds"] = true;  
// delete  
delete user["likes birds"];
```

**Note:** Any type of quotes will work.

# Objects

Square brackets also provide a way to obtain the property name as the result of any expression – as opposed to a literal string – like from a variable as follows:

```
let key = "likes birds";  
// same as user["likes birds"] = true;  
user[key] = true;
```

Here, the variable key may be calculated at run-time or depend on the user input. And then we use it to access the property. That gives us a great deal of flexibility. For instance:

```
let user = {  
  name: "John",  
  age: 30,  
};  
let key = prompt("What do you want to know about the user?", "name");  
// access by variable  
alert(user[key]); // John (if enter "name")
```

# Typeof operator

The **typeof** operator returns the type of the argument. It's useful when we want to process values of different types differently or just want to do a quick check. A call to **typeof x** returns a string with the type name:

```
typeof undefined // "undefined"
typeof 0 // "number"
typeof 10n // "bigint"
typeof true // "boolean"
typeof "foo" // "string"
typeof Symbol("id") // "symbol"
typeof Math // "object" (1)
typeof null // "object" (2)
typeof alert // "function" (3)
```

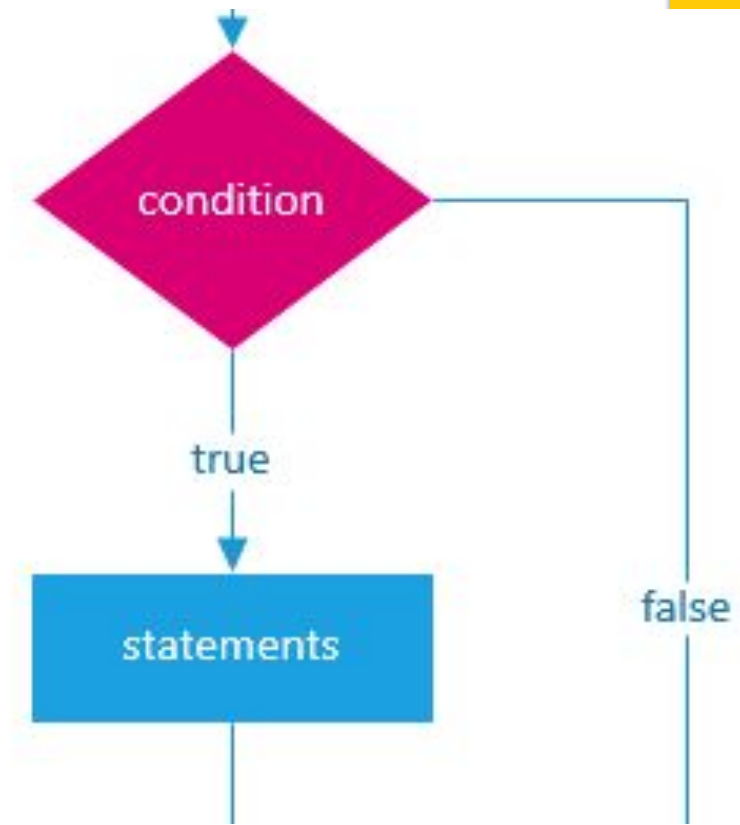
(1) **Math** is a built-in object that provides mathematical operations. We will learn it in the chapter Numbers. Here, it serves just as an example of an object.

(2) The result of **typeof null** is "object". That's an officially recognized error in **typeof**, coming from very early days of JavaScript and kept for compatibility. Definitely, null is not an object. It is a special value with a separate type of its own. The behavior of **typeof** is wrong here.

(3) The result of **typeof alert** is "function", because alert is a function. We'll study functions in the next chapters where we'll also see that there's no special "function" type in JavaScript. Functions belong to the object type. But **typeof** treats them differently, returning "function". That also comes from the early days of JavaScript.



```
condition01) {  
  condition01 = true (will  
  if(condition02) {  
    condition02 = true (will  
  {  
    condition 02 = false (wi
```



# Conditionals

JAVASCRIPT

# Conditionals

We know many comparison operators from math. In JavaScript they are written like this:

- Greater/less than: **a > b**, **a < b**.
- Greater/less than or equals: **a >= b**, **a <= b**.
- Equals: **a == b**, please note the double equality sign **==** means the equality test, while a single one **a = b** means an assignment.
- Not equals: In math the notation is **≠**, but in JavaScript it's written as **a != b**.

All comparison operators return a boolean value:

- **true** – means “yes”, “correct” or “the truth”.
- **false** – means “no”, “wrong” or “not the truth”.

```
alert( 2 > 1 ); // true (correct)
alert( 2 == 1 ); // false (wrong)
alert( 2 != 1 ); // true (correct)
```

## Conditionals – string comparison

To see whether a string is greater than another, JavaScript uses the so-called “dictionary” or “lexicographical” order. In other words, strings are compared letter-by-letter. For example:

```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
alert( 'Bee' > 'Be' ); // true
```

The algorithm to compare two strings is simple:

- Compare the first character of both strings.
- If the first character from the first string is greater (or less) than the other string's, then the first string is greater (or less) than the second. We're done.
- Otherwise, if both strings' first characters are the same, compare the second characters the same way.
- Repeat until the end of either string.
- If both strings end at the same length, then they are equal. Otherwise, the longer string is greater.

## Conditionals – different type comparison

A regular equality check `==` has a problem. It cannot differentiate 0 from false:

```
alert( 0 == false ); // true
```

The same thing happens with an empty string:

```
alert( '' == false ); // true
```

This happens because operands of different types are converted to numbers by the equality operator `==`

## Conditionals – different type comparison

An empty string, just like false, becomes a zero. What to do if we'd like to differentiate 0 from false? A strict equality operator === checks the equality without type conversion. In other words, if a and b are of different types, then a === b immediately returns false without an attempt to convert them. Let's try it:

```
alert( 0 === false ); // false, because the types are different
```

There is also a “strict non-equality” operator !== analogous to !=. The strict equality operator is a bit longer to write, but makes it obvious what's going on and leaves less room for errors.

## Conditionals – null and undefined

There's a non-intuitive behavior when **null** or **undefined** are compared to other values.

- *For a strict equality check ===*

These values are different, because each of them is a different type.

```
alert( null === undefined ); // false
```

- *For a non-strict check ==*

There's a special rule. These two are a “sweet couple”: they equal each other (in the sense of ==), but not any other value.

```
alert( null == undefined ); // true
```

- *For maths and other comparisons < > <= >=*

**null/undefined** are converted to numbers: null becomes 0, while undefined becomes NaN.

## Conditionals – null and undefined

Let's compare null with a zero:

```
alert( null > 0 ); // (1) false  
alert( null == 0 ); // (2) false  
alert( null >= 0 ); // (3) true
```

Mathematically, that's strange. The last result states that "null is greater than or equal to zero", so in one of the comparisons above it must be true, but they are both false.

The reason is that an equality check `==` and comparisons `>` `<` `>=` `<=` work differently. Comparisons convert null to a number, treating it as 0. That's why (3) `null >= 0` is true and (1) `null > 0` is false.

On the other hand, the equality check `==` for undefined and null is defined such that, without any conversions, they equal each other and don't equal anything else. That's why (2) `null == 0` is false.

## Conditionals – null and undefined

The value undefined shouldn't be compared to other values:

```
alert( undefined > 0 ); // false (1)
alert( undefined < 0 ); // false (2)
alert( undefined == 0 ); // false (3)
```

Why is it always false? We get these results because:

- Comparisons (1) and (2) return false because undefined gets converted to NaN and NaN is a special numeric value which returns false for all comparisons.
- The equality check (3) returns false because undefined only equals null, undefined, and no other value.

Treat any comparison with undefined/null except the strict equality === with exceptional care.

Don't use comparisons >= > < <= with a variable which may be null/undefined, unless you're really sure of what you're doing. If a variable can have these values, check for them separately.



# Conditional statements

Very often when you write code, you want to perform different actions for different decisions. You can use conditional statements in your code to do this. In JavaScript we have the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false
- Use switch to specify many alternative blocks of code to be executed

# Conditional statements - examples

```
if (hour < 18) {  
    greeting = "Good day";  
}
```

Make a "Good day" greeting if the hour is less than 18:00:

```
if (hour < 18) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}
```

If the hour is less than 18, create a "Good day" greeting, otherwise "Good evening":

```
if (time < 10) {  
    greeting = "Good morning";  
} else if (time < 20) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}
```

If time is less than 10:00, create a "Good morning" greeting, if not, but time is less than 20:00, create a "Good day" greeting, otherwise a "Good evening":

# Conditional statements – switch statement

How switch statements work:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed

The **getDay()** method returns the weekday as a number between 0 and 6.  
(Sunday=0, Monday=1, Tuesday=2 ..)

This example on the right uses the weekday number to calculate the weekday name.

```
switch (new Date().getDay()) {  
  case 0:  
    day = "Sunday";  
    break;  
  case 1:  
    day = "Monday";  
    break;  
  case 2:  
    day = "Tuesday";  
    break;  
  case 3:  
    day = "Wednesday";  
    break;  
  case 4:  
    day = "Thursday";  
    break;  
  case 5:  
    day = "Friday";  
    break;  
  case 6:  
    day = "Saturday";  
}
```

```
st a= 9;  
st b= 9;
```

(param1, param2, paramN) =  
ES5

```
add = function(x, y) {  
  return x + y;
```

sole.log(add(a,b));  
ES6

```
= (x, y) => { return x +  
sole.log(add(a,b));
```

NAME



```
function addNumbers(a,  
  return a + b;  
}
```

# Functions

JAVASCRIPT

# Functions

Functions allow you to store a piece of code that does a single task inside a defined block, and then call that code whenever you need it using a single short command — rather than having to type out the same code multiple times. In this article we'll explore fundamental concepts behind functions such as basic syntax, how to invoke and define them, scope, and parameters.

The built-in code we've made use of so far comes in both forms: functions and methods. You can check the full list of the built-in functions, as well as the built-in objects and their corresponding methods [here](#).

To actually use a function after it has been defined, you've got to run — or invoke — it. This is done by including the name of the function in the code somewhere, followed by parentheses:

```
function myFunction() {  
  alert("hello");  
}  
  
myFunction();  
// calls the function once
```

# Function parameters

Some functions require parameters to be specified when you are invoking them — these are values that need to be included inside the function parentheses, which it needs to do its job properly. As an example, the browser's built-in `Math.random()` function doesn't require any parameters. When called, it always returns a random number between 0 and 1:

```
const myNumber = Math.random();
```

The browser's built-in string `replace()` function however needs two parameters — the substring to find in the main string, and the substring to replace that string with:

```
const myText = 'I am a string';  
const newString = myText.replace('string', 'sausage'); // 'I am a sausage'
```

# Function parameters

In JavaScript, parameters are the items listed between the parentheses in the function declaration. Function arguments are the actual values we decide to pass to the function. In the example below, the function definition is written on the first line: **function favoriteAnimal(animal)**. The parameter, `animal`, is found inside the parentheses. We could just as easily replace `animal` with `pet`, `x`, or `blah`. But in this case, naming the parameter `animal` gives someone reading our code a bit of context so that they don't have to guess what `animal` may eventually contain. By putting `animal` inside the parentheses of the `favoriteAnimal()` function, we are telling JavaScript that we will send some value to our `favoriteAnimal` function. This means that `animal` is just a placeholder for some future value.

```
function favoriteAnimal(animal) {  
  console.log(animal + " is my favorite animal!");  
}  
  
favoriteAnimal("Goat");
```

The last line, **favoriteAnimal('Goat')**, is where we are calling our `favoriteAnimal` function and passing the value `Goat` inside that function.

# Function parameters

Sometimes parameters are optional — you don't have to specify them. If you don't, the function will generally adopt some kind of default behavior. As an example, the array `join()` function's parameter is optional:

```
const myArray = ['I', 'love', 'chocolate', 'frogs'];
const madeAString = myArray.join(' ');
console.log(madeAString);
// returns 'I love chocolate frogs'

const madeAnotherString = myArray.join();
console.log(madeAnotherString);
// returns 'I,love,chocolate,frogs'
```

If no parameter is included to specify a joining/delimiting character, a comma is used by default.



# Function parameters

If you're writing a function and want to support optional parameters, you can specify default values by adding = after the name of the parameter, followed by the default value:

```
function hello(name = "Chris") {  
  console.log(`Hello ${name}!`);  
}
```

```
hello("Ari"); // Hello Ari!  
hello(); // Hello Chris!
```

# Anonymous functions and arrow functions

So far we have just created a function like so:

```
function myFunction() {  
  alert('hello');  
}
```

But you can also create a function that doesn't have a name:

```
function() {  
  alert('hello');  
}
```

This is called an anonymous function, because it has no name. You'll often see anonymous functions when a function expects to receive another function as a parameter. In this case the function parameter is often passed as an anonymous function. An anonymous function is not accessible after its initial creation, it can only be accessed by a variable it is stored in as a function as a value. An anonymous function can also have multiple arguments, but only one expression.

## Anonymous functions and arrow functions

In this example, we define an anonymous function that prints a message to the console. The function is then stored in the greet variable. We can call the function by invoking greet().

```
var greet = function () {  
  console.log("Hey guys!");  
};  
  
greet();
```

# Anonymous functions and arrow functions

There's another very simple and concise syntax for creating functions, that's often better than Function Expressions. It's called "arrow functions", because it looks like this:

```
let func = (arg1, arg2, ..., argN) => expression;
```

This creates a function func that accepts arguments arg1..argN, then evaluates the expression on the right side with their use and returns its result. In other words, it's the shorter version of:

```
let func = function(arg1, arg2, argN) {  
  return expression;  
};
```

# Anonymous functions and arrow functions

Let's see a concrete example:

```
let sum = (a, b) => a + b;

/* This arrow function is a shorter form of:

let sum = function(a, b) {
  | return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

As you can see, **(a, b) => a + b** means a function that accepts two arguments named **a** and **b**. Upon the execution, it evaluates the expression **a + b** and returns the result.

# Anonymous functions and arrow functions

If we have only one argument, then parentheses around parameters can be omitted, making that even shorter.

```
let double = n => n * 2;  
// roughly the same as: let double = function(n) { return n * 2 }  
  
alert( double(3) ); // 6
```

If there are no arguments, parentheses are empty, but they must be present:

```
let sayHi = () => alert("Hello!");  
  
sayHi();
```

Arrow functions may appear unfamiliar and not very readable at first, but that quickly changes as the eyes get used to the structure. They are very convenient for simple one-line actions, when we're just too lazy to write many words.

# Function scopes

Let's talk a bit about **scope** — a very important concept when dealing with functions. When you create a function, the variables and other things defined inside the function are inside their own separate scope, meaning that they are locked away in their own separate compartments, unreachable from code outside the functions.

The top level outside all your functions is called the global scope. Values defined in the global scope are accessible from everywhere in the code.

JavaScript is set up like this for various reasons — but mainly because of security and organization. Sometimes you don't want variables to be accessible from everywhere in the code — external scripts that you call in from elsewhere could start to mess with your code and cause problems because they happen to be using the same variable names as other parts of the code, causing conflicts. This might be done maliciously, or just by accident.

# Return values

Return values are just what they sound like — the values that a function returns when it has completed. You've already met return values a number of times, although you may not have thought about them explicitly.

```
const myText = 'The weather is cold';
const newString = myText.replace('cold', 'warm');
console.log(newString); // Should print "The weather is warm"
// the replace() string function takes a string,
// replaces one substring with another, and returns
// a new string with the replacement made
```

The `replace()` function is invoked on the **myText** string, and is passed two parameters:

- the substring to find ('cold').
- the string to replace it with ('warm').

When the function completes (finishes running), it returns a value, which is a new string with the replacement made. In the code above, the result of this return value is saved in the variable **newString**.



# Assignment

aleksa.fd.tutor@gmail.com

JS



Thank you  
for your  
attention!

JS

