JavaScript with jQuery

Course overview

1. Course introduction

1. Kursplan

2. Introduction to Programming

1. Introduction to Programming

3. JavaScript Introduction

1. JavaScript history
2. The DOM
3. Inclusion

4. JavaScript Syntax

1. Identifiers
2. Keywords
3. Operators
4. Values and Types
5. Numbers
6. Strings
7. Date object
8. Variables and Blocks
9. Flow Control Statements
10. Functions
11. Scope
12. Closure

5. Retrieving data with JavaScript

1. XMLHttpRequest

6. jQuery

1. Introduction to jQuery
2. Why use jQuery?
3. Getting Started with jQuery
4. jQuery plugins
5. Selecting elements
6. Handling events
7. Working with Ajax

7. More ES6 features

1. ES2015 aka ES6
2. Destructuring Assignment
3. Extended Parameter Handling
4. Modules
5. Classes
6. Promises

8. Templating

1. Why Templating?
2. HandlebarsJS
3. Handlebars Helpers

Course introduction

Sections in this chapter:

1. Kursplan

1-1. Kursplan

Kursens innehåll

1-1-1

Kursen syftar till att den studerande utvecklar specialiserade kunskaper genom teori och praktiska övningar. Den får utveckla sina kunskaper om, färdigheter i, och kompetenser för att skapa och utveckla enklare front-end tillämpningar.

Kursen omfattar följande moment:

1-1-2

- Introduktion i JavaScript
- Versionshantering i Git
- JavaScript med jQuery
- Event-hantering med JavaScript
- AJAX
- Manipulera DOM
- Dynamiskt uppdatera HTML-sidor med jQuery

Kursens mål

1-1-3

Målet med kursen är att den studerande ska genom teori och praktiska övningar, utveckla sina kunskaper och färdigheter i att använda programmeringsspråket JavaScript med jQuery; hur språket skrivs, hur det exekverar samt dess olika möjligheter och begränsningar.

Kursen introducerar de olika datatyperna i JavaScript och ger en inblick i hur språket fungerar och kan användas i en front-end miljö. De studerande får öva att hantera händelser, göra AJAX- anrop, manipulera DOM och dynamiskt förändra webbsidor.

1-1-4

Efter genomförd kurs ska den studerande ha kunskaper i/om: 1-1-5

- Om jQuery
- Om versionshantering, Git
- Om AJAX-anrop
- Om manipulering av DOM
- Om dynamisk förändring av CSS med jQuery

Efter genomförd kurs ska den studerande ha färdigheter i att: 1-1-6

- hantera jQuery biblioteket för att effektivisera kod och göra asynkrona AJAX-anrop för att hämta data
- kunna hämta, ändra och skapa olika HTML-element med JavaScript och jQuery
- versionshantering med Git

Efter genomförd kurs ska den studerande ha kompetens för att: 1-1-7

- arbeta med versionshantering
- självständigt använda JavaScript för frontendlösningar
- utveckla interaktiva webbhemsidor med JavaScript och jQuery

Former för undervisning 1-1-8

Kursen kommer att genomföras med traditionell undervisning i form av föreläsningar varvat med tid för praktisk träning på övningsuppgifter, med handledning av läraren.

I kursen ingår också att genomföra inlämningsuppgifter på självstudietiden med efterföljande redovisningar. 1-1-9

Undervisningsspråk 1-1-10

Svenska

Förkunskapskrav 1-1-11

Inga

Examination och former av kunskapskontroll 1-1-12

Kunskapskontroller görs under kursen genom fyra obligatoriska laborationer, en större inlämningsuppgift i grupp och en muntlig redovisning. Kursen avslutas med praktisk tentamen.

Betygsskala

1-1-13

Följande betygsskala tillämpas: VG = Väl Godkänd, G = Godkänd, IG = Icke Godkänd

Principer för betygssättning

1-1-14

För betyget Godkänd ska den studerande:

- Den studerande har uppnått samtliga mål med kursen

För betyget Väl Godkänd ska den studerande:

1-1-15

- Den studerande har uppnått samtliga mål med kursen.
- Självständigt skapa webbformulär inklusive validering

Icke Godkänd ges till studerande som inte bedöms uppfylla kraven för betyget Godkänd.

1-1-16

Introduction to Programming

Sections in this chapter:

1. Introduction to Programming

2-1. Introduction to Programming

Writing code is essentially typing text into a text file. 2-1-1

That text is referred to as your **source code**.

Source Code 2-1-2

The **source code** is a text document combined of a set of special words, phrases and operators.

It is designed to instruct the computer **what to do**.

```
if(age > 0) {
    console.log(age + " is greater than 0!");
}
```

However, these words, phrases and operators are not in a form the computer can understand. 2-1-3

It is human readable but the computer needs some **assistance** to read it.

So, we need a step for **converting** the source code into **instructions that the computer can understand**. 2-1-4

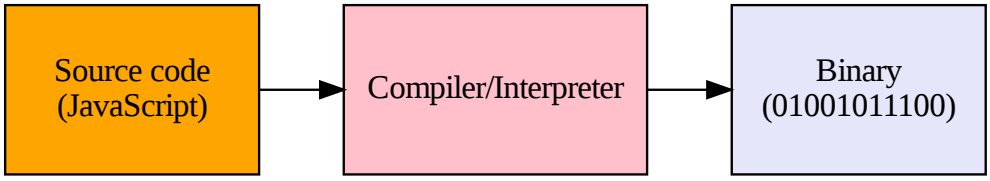
These instructions are in the form of **binary code**, `0's` and `1's`.

A	100 0001	E	100 0101
B	100 0010	F	100 0110
C	100 0011	G	100 0111
D	100 0100	H	100 1000

Your computer **doesn't care** about the source code itself. 2-1-5

Source code can be written in many **different ways** and **different languages**, but in the end it makes **no difference** to your computer how you write it.

The **compiler** or **interpreter** in your computer or browser, whose job it is to convert the source code into **binary code** that the processor can understand, may care. 2-1-6



Compiler 2-1-7

A **compiler** is a special program that processes **source code** and turns it into **code** that a computer's **processor** can read and execute.

Interpreter 2-1-8

An **interpreter** does pretty much the same thing as a compiler, but it will read the source code **line by line** and then convert and execute it.

Compiler	**Interpreter**
Translates whole source code at once	Translates and executes source code line by line
Slow code analyze	Shorter analyze time
Quicker execution time	Slower execution time
C, C#, C++, Java	JavaScript, Python, Ruby

2-1-9

Compiler *2-1-10*

Translates whole source code at once - Takes whole source and translates it to machine code in one step

Interpreter

Translates source code line by line - Translates line by line within the source code as it is about to be executed

Compiler *2-1-11*

Slow code analyze - Since the whole code base is being analyzed at once this requires more time

Interpreter

Shorter analyze time - Shorter analyze time since it only analyzes one statement at a time

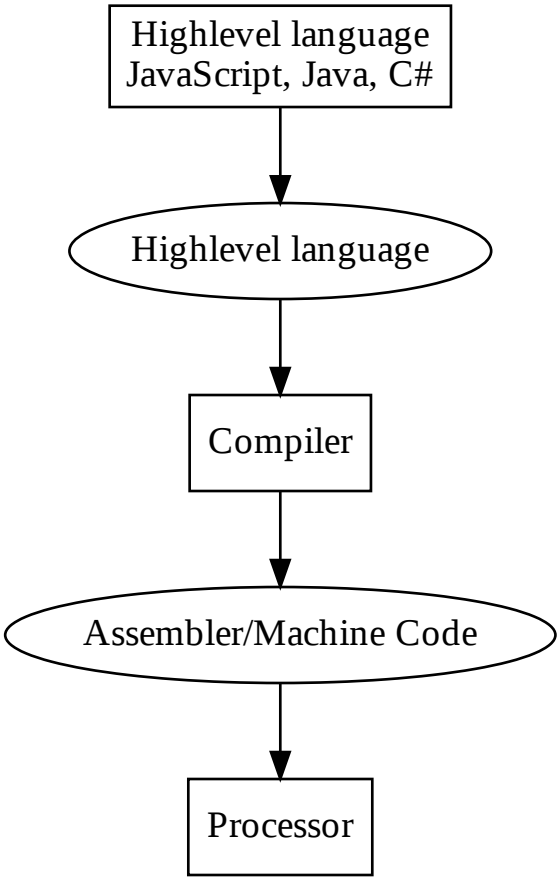
Compiler *2-1-12*

Quicker execution time - Since the whole code base is already compiled to machine code the code can be executed right away

Interpreter

Slower execution time - Slower execution time since not all source code is ready as machine code when it is about to be executed

The browser takes JavaScript code and compiles it to code that the processor can understand and execute. *2-1-13*



Programmers mostly write programs with a high-level language such as JavaScript or C#.

The high-level code has to be translated to machine code for the computer to be able to execute the code or program.

The output of the compiler or interpreter **depends on the language of the source code** and **what compiler or interpreter** that is used to convert the code.

In Chrome, it is by default the **V8 engine** that handles the compilation of JavaScript code.

Different browsers can have different compilers or interpreters.

High-level code

High-level code, source code, is written by humans with a high-level language such as JavaScript.

The code is human read-able and uses English based statements and words. 2-1-19

```
if(x > 0) {
    console.log(x + " is greater than 0!");
}
```

Above example is written in JavaScript

<<<<<<< HEAD 2-1-20

=======

Unhide all slides Popular high-level languages:

- JavaScript
- Java
- C#
- Ruby

<<<<<<< HEAD 2-1-21

Popular high-level languages

JavaScript

Code examples of above high-level languages:

JavaScript

Unhide all slides

```
var x = 3;
if(x > 0) {
    console.log(x + " is greater than 0!");
}
```

<<<<<<< HEAD

Java

Java 2-1-22

Unhide all slides

```
int x = 3;
if (x > 0)
{
   Console.WriteLine(x + " is greater than 0!");
}
```

<<<<<<< HEAD

2-1-23

C#

C#

Unhide all slides

```
int x = 3;
if (x > 0)
{
    Console.WriteLine(x + " is greater than 0!");
}
```

<<<<<<< HEAD

Ruby

Ruby

2-1-24

Unhide all slides

```
x = 1
if x > 2
   puts "x is greater than 2"
```

All high-level code must be interpreted or compiled into code that the processor can understand before being executed.

2-1-25

Assembly code / Machine code

2-1-26

Assembly code, also known as *symbolic machine code*, is a low-level language.

```
0013           RESETA  EQU    %00010011
0011           CTLREG  EQU    %00010001

C003 86 13     INITA   LDA A  #RESETA   RESET ACIA
C005 B7 80 04          STA A  ACIA
C008 86 11             LDA A  #CTLREG   SET 8 BITS AND 2 STOP
C00A B7 80 04          STA A  ACIA

C00D 7E C0 F1          JMP    SIGNON    GO TO START OF MONITOR
```

Assembly code is somewhat human read-able but **closer to machine language**.

2-1-27

JavaScript code written on one line could result in hundreds lines of assembler code.

Machine code is close to impossible for humans to either read or write, since it **consists of numbers only**, `0's` and `1's`. 2-1-28

```
0000 1001 1100 0110 1010 1111 0101 1000 0000 1001 1100 0110 1010
1010 1111 0101 1000 0000 1001 1100 0110 1010 1111 0101 1000 0000
1100 0110 1010 1111 0101 1000 0000 1001 1100 0110 1010 1111 0101
```

Machine code is being handled by the *processor*, the **CPU** (**C**entral **P**rocessing **U**nit). 2-1-29

The processor is also known as the brain of the computer.

You have written a program with the following functions. 2-1-30

Machine Instructions	Machine Operations
0000	Stop Program
0001	Turn light on
0010	Turn light off
0100	Dim light

Translated, the combinations of `0's` and `1's` will represent some functionality.

Syntax 2-1-31

When we talk about writing source code, there is a word that is often referred to, which is **syntax**.

The set of rules that puts syntax together is called the **grammar**. 2-1-32

The **syntax** and the **grammar** together tells you how to write for instance a JavaScript program.

It basically tells you the **valid combinations of characters and words** that will make sure your program does what you want it to do. 2-1-33

```
// Syntax for declaring a variable in JavaScript
var speed = 0;
```

```
// Syntax for declaring a variable in Java
int speed = 0;
```

This is much like the syntax for Swedish or English. 2-1-34

There are punctuation marks, like the comma, the period and the exclamation mark. 2-1-35

There are parts of **phrases that work together**, like a word used as a verb versus a word used as a noun.

The way you put these things together, first into a phrase and then phrases into sentences, is called **grammar**. 2-1-36

A similar concept exists in programming, where you first learn the syntax, and then the rules of how to put the syntax together to make a coherent statement. 2-1-37

Many IDEs have support for **catching syntax errors**. 2-1-38

If the code does not follow the correct syntax an error will be displayed.

2-1-39

```
int speed = 0;
```

Writing this in JavaScript would give you an error since JavaScript is **dynamically typed**, meaning we don't define data types.

Pseudo code 2-1-40

Pseudo code is a detailed yet readable description of what a computer program or algorithm does.

It is many times used in the process of creating a program, since it is an easy way for non-programmers to express the functionality they need. 2-1-41

Programmers can then use the pseudo code as a template to write code in whatever programming language they are most comfortable with. 2-1-42

Let's say you wanted to write the pseudo code to print "Hello World!". It could look like any of the following: 2-1-43

```
Display Hello World!
Print Hello World
Write Hello World
```

As you may have noticed, pseudo code is essentially a mix between natural language, like how most humans speak to each other, and logical statements that will be understood by programmers. 2-1-44

The pseudo code for the following code: 2-1-45

```
if(age > 0) {
    console.log("Greater than 0!");
}
else {
    console.log("Zero or less!");
}
```

could look like this:

```
If age is greater than zero
    Print "Greater that 0!"
Else
    Print "Zero or less!"
```

JavaScript Introduction

Sections in this chapter:

1. JavaScript history
2. The DOM
3. Inclusion

3-1. JavaScript history

Originally and (in)famously created in 10 days by Brendan Eich, who wanted to build a LISP interpreter for the browser. 3-1-1

Name is still highly confusing, and it still gets confused with Java every now and then.

Timeline 3-1-2

... Finding a name is hard

- 1995 - Created by Brendan Eich. Was initially called **Mocha**
- 1995 - Later same year, renamed to **LiveScript**
- 1995 - Hey, still same year. Renamed to **JavaScript** after licensing from Sun (Now Oracle)

- 1996 - JavaScript taken to the ECMA committee. Official name: ECMAScript 3-1-3
- 1998 - ECMAScript 2 released
- 1999 - ECMAScript 3 released
- 2000 - Work on ECMAScript 4 begins, but gets cancelled for various reasons

Then, nothing happens for a few years.

During 2005: 3-1-4

- Work restarts on ES4. Lots of things happening.
- Later this year... Crockford, Microsoft and a few other players opposes ES4. Forms their own subcommittee and designs ES3.1
- Bickering between ES3.1 and ES4 goes on for quite some time...

During 2008: 3-1-5

- Finally, an agreement. ES3.1 prevails but gets renamed to ES5.

So, to summarize: ES5 used to be ES3.1, but shortly before that it was ES4 which got cancelled twice. 3-1-6

Awesome.

In 2015 and 2016, respectively, ES6 and ES7 specifications are finalized. Officially known as ECMAScript 2015 and ECMAScript 2016. 3-1-7

... *And here we are*

3-2. The DOM

The **D**ocument **O**bject **M**odel is the bridge between the JavaScript and HTML. *3-2-1*

It is the **browsers internal representation of the HTML**.

A web page is a document. *3-2-2*

This document can be displayed visually in the browser or as the HTML source, the markup we write.

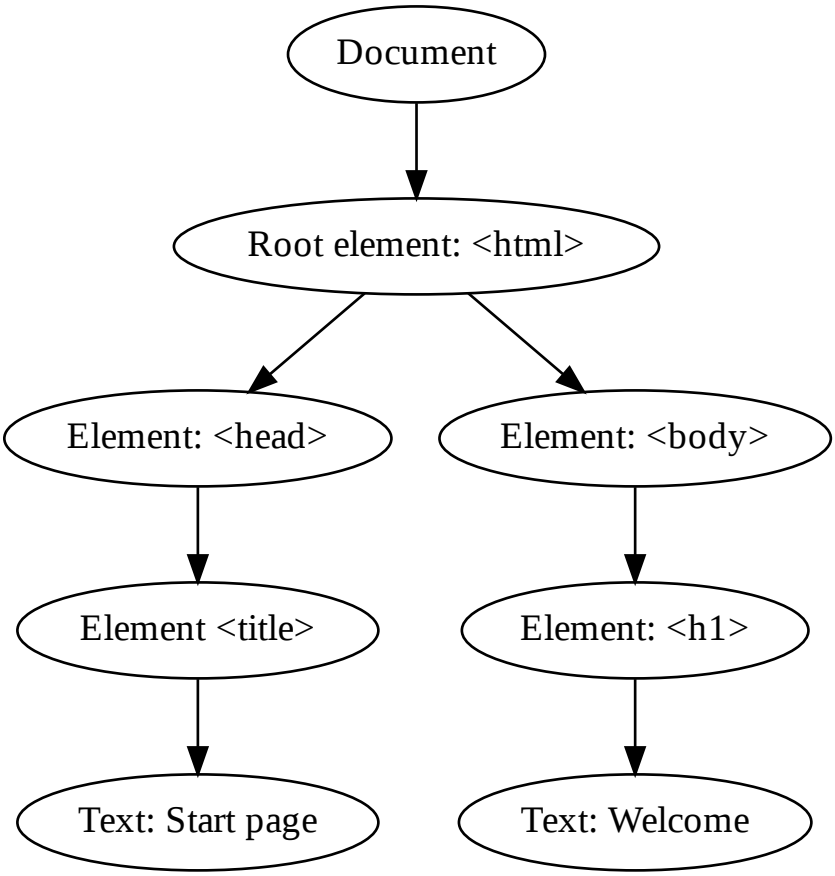
The DOM is this same document represented in **objects and nodes**. *3-2-3*

We can read and mutate these objects and nodes with JavaScript.

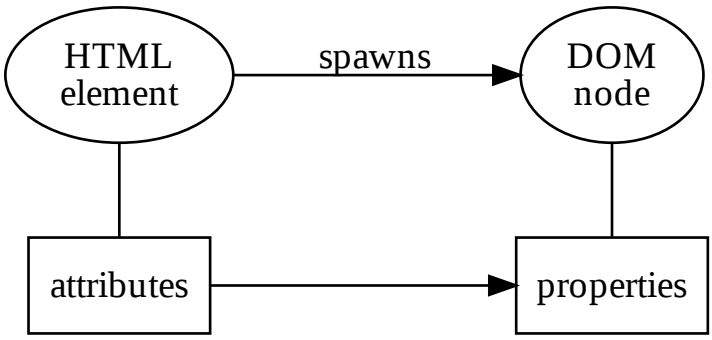
There are two major ways we can mutate the DOM: *3-2-4*

1. Insert nodes into the DOM, take them out, move them, etc.
2. Change content or properties of DOM nodes themselves.

The DOM is structured as an object tree: 3-2-5



HTML element **attributes** are represented in the nodes as **properties**. 3-2-6



Showcase 3-2-7

We will now **showcase the difference** of attributes and properties using this simple page:

It is published at http://blog.krawaller.se/attrvsprops.

If you **`view source`** you'll see this: *3-2-8*

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
  </head>
  <body>
    <input value="John Doe"> <button>Submit</button>
  </body>
</html>
```

Note that the **`input`** has the **initial value `"John Doe"`**.

In the console we can **get a reference to the input field node** like this: *3-2-9*

```
var field = document.querySelector("input");
```

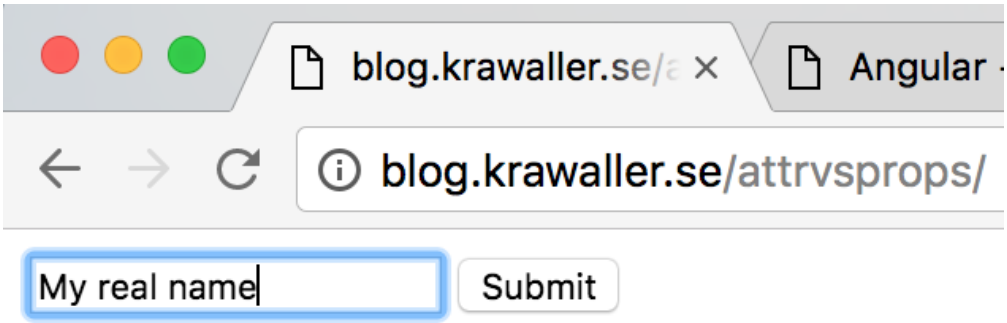
Using that reference we can **confirm the value of the `value` attribute**: *3-2-10*

```
field.getAttribute("value") // "John Doe";
```

And we can also **read the `value` property**: *3-2-11*

```
field.value // "John Doe";
```

Now **type in the field** to say something else: *3-2-12*



If we **query the attribute again**, we see that it is still `John Doe`: *3-2-13*

```
field.getAttribute("value") // "John Doe";
```

But the **property is updated**: 3-2-14

```
field.value // "My real name";
```

This reflects the fact that **attributes are characteristics of the original HTML elements**, while **properties belong to the live DOM nodes**. 3-2-15

To wit: 3-2-16

- **HTML elements have attributes**, and **DOM nodes have properties**.
- **Attributes often initialize properties** (but not always)
- **Attributes never change**, but **properties can change**.

We traverse through the DOM to access children or parents of an element, or the element itself. 3-2-17

document object 3-2-18

We **interact with the DOM** from JavaScript space through the global `document` object.

The `document` object represents the actual web page, our document. 3-2-19

Go to Google and try this in the console: 3-2-20

```
document.body.setAttribute("style","background-color:red;")
```

We *write* JavaScript but *use* the DOM to access elements.

Also, try `document.designMode = "on"`. 3-2-21

Now you can edit anything on the page! *Think about the power inherent in this. Never trust a screenshot.*

Through the `document` object we can access and see all nodes in the DOM. 3-2-22

Go to Google and try this in the console:

```
document.childNodes; // (2)[<!DOCTYPE html>, html]
```

`childNodes` gives you the nodes in the current node.

We get an array of objects which in turn contains more arrays and objects. 3-2-23

The **child nodes** of `document` is usually the `doctype` and `html` elements.

These objects of the array contains a lot of properties 3-2-24

```
assignedSlot : null
attributes : NamedNodeMap {0: lang, 1: class, length: 2}
baseURI : "https://www.google.se/_/chrome/newtab?espv=2&ie=UTF-8"
childElementCount : 2
childNodes : NodeList(2)
   length : 2
   0 : head
     accessKey: ""
     assignedSlot : null
     attributes : NamedNodeMap {length: 0}
     baseURI : "https://www.google.se/_/chrome/newtab?espv=2&ie=UTF-8"
     childElementCount : 12
     childNodes : (12) [style, style, link, style, link, ...]
   1 : body
     accessKey: ""
```

Shortened for convenience

Each node contains `childNodes` of itself. 3-2-25

Child nodes of `html` can be `head` and `body`

We could do this: 3-2-26

```
let html = document.childNodes[1];
let body = html.childNodes[1];
body.style.backgroundColor = "green";
```

But there is an easier ways if we want to modify nodes in the DOM. 3-2-27

The `document` object serves properties and methods for common access, such as `body` for example.

You can find a list of available properties and methods of the `document` object at MDN 3-2-28

Through `document.body` we get an **HTML node**, an **object that represents the element**. 3-2-29

```
document.body;
```

We can chain further and further since it is an object, so `body` in turn have a lot of properties and methods just like `document`. 3-2-30

```
document.body.onresize = () => {
    console.log("RESIZED")
};
```

On `body` we could for example 3-2-31

- call the **`setAttribute`** method
- query attributes using the **`getAttribute`** method
- change its content by assigning to **`innerHTML`** or **`textContent`**
- iterate over child nodes using the **`children`** property, allowing us to walk down the tree.
- go back up the tree using the **parentNode** property.

We don't have to walk the tree from `body` or trough the `childNodes` property to get access to children or grand children and so on. 3-2-32

We are served **several methods** to **find a specific node or set of nodes** on `document`: 3-2-33

- `getElementById("someId")`
- `getElementsByClassName("someClass")`
- `getElementsByTagName("div")`

You can find a list of all available methods at MDN 3-2-34

3-2-35

```
<button id="myButton">Click me!</button>
```

```
let button = document.getElementById("myButton");
```

Some of these methods returns one element while others returns several.

Notice the `getElement`**`s`**`ByTagName` for example 3-2-36

```
document.getElementsByTagName('div');
// (46) [div#modal-collapse.modal-background, div.navbar-inner, ...]
```

It is plural and will always return an array with element; zero, one or more.

The `getElementById` for example only returns one specific element. 3-2-37

An id is unique and should only be present on one element in your document.

```
document.getElementById('modal-collapse');
// <div id="modal-collapse" class="modal-background"></div>
```

And, there are two methods which **gets elements using CSS selectors**: 3-2-38

- `querySelector("#someId")`

  Returns one element

- `querySelectorAll("article > p")`

  Returns list of elements

querySelector(target) 3-2-39

The `querySelector()` functions returns only one element.

```
document.querySelector("#modal-collapse");
// <div id="modal-collapse" class="modal-background"></div>
```

The passed argument to the function can be what css selector you want, preferably an id since it returns only one element.

If you use the class selector for example as an argument to the `querySelector()` 3-2-40

```
document.querySelector(".col-sm-4");
// <div class="col-sm-4">...</div>
```

It will only return the first occurring element and then stop searching.

querySelectorAll(target) 3-2-41

The `querySelectorAll()` will always return an array, with none, one or several elements.

```
document.querySelectorAll(".col-sm-4");
// (3) [div.col-sm-4, div.col-sm-4, div.col-sm-4]

document.querySelectorAll(".col-sm-423");
// []
```

window object

3-2-42

The window object **represents an open window in a browser**.

To access information regarding the browser rather than the document, you use the window object.

Scrolling the whole browser window to a specific point for example

3-2-43

```
window.scrollTo(0, 1000);
```

This will scroll the window based on the given X- and Y-coordinates

console

3-2-44

We have seen the following in a few places

```
console.log("...")
```

This an object to access the browsers debugging console.

It is exposed as `window.console` but can be accessed simply as `console`

3-2-45

```
window.console.log("...");

// same as...

console.log("...");
```

We use `.log()` to log a message or value to the console.

3-2-46

But there are more methods on the console object

3-2-47

- `error()`
- `warn()`

You can find the full list of methods available at MDN

The window object has several properties and methods, amongst many can be: 3-2-48

- `.alert()` - Alert box showing a message
- `.confirm()` - Confirmation box
- `.location` - Current location, url
- `.sessionStorage` - Object used to store data, in session
- `.localStorage` - Object used to store data, no expiration

You can find a full list of available properties and methods at MDN 3-2-49

Get and set values 3-2-50

We have seen how to access elements, but how do we access the **value of an element**.

Input elements 3-2-51

The following gives us an element that the user can use to input a value.

```
<input type="text" id="fname">
```

To get the value of the input we use the `.value` property. 3-2-52

```
let firstName = document.getElementById("fname");
console.log(firstName);
// <input id="input" value="My input">

console.log(firstName.value);
// "The value in the input field"
```

Other elements 3-2-53

We can get the content of other elements that are not input elements, such as paragraphs.

```
<p id="paragraph">My little paragraph</p>
```

For this we can use `innerHTML` 3-2-54

```
let paragraph = document.getElementById("paragraph");

console.log(paragraph.innerHTML);
// "My little paragraph"
```

We could also use `innerText` 3-2-55

```
console.log(paragraph.innerText);
// "My little paragraph"
```

Q Is there any difference between them? 3-2-56

A Yes, if we would have another HTML element within our paragraph, a `span` for example. 3-2-57

```
<p id="paragraph">My little paragraph <span>with span</span></p>
```

Using the `innerText` we would get only the text content 3-2-58

```
let paragraph = document.getElementById("paragraph");

console.log(paragraph.innerText);
// "My little paragraph with span"
```

While `innerHTML` we would get the HTML elements included in the string also. 3-2-59

```
console.log(paragraph.innerHTML);
// "My little paragraph <span>with span</span>"
```

We can use the same properties to set the value or content of an element. 3-2-60

```
let paragraph = document.getElementById("paragraph");
let input = document.getElementsByTagName('input')[0];

paragraph.innerHTML = "Text <span>span</span>";
paragraph.innerText = "Text";
input.value = "Some text";
```

Adding new nodes 3-2-61

We have seen how we can access and operate on existing elements in the DOM.

With JavaScript we can also insert new nodes.

createElement()

3-2-62

Lets say we have a page looking like this

```
<body>
    <div>
        <p>Hello</p>
    </div>
</body>
```

Now we want to add a new paragraph to the div with JavaScript.

We need to create the `p` element using `createElement()`.

3-2-63

```
let paragraph = document.createElement("p");
```

Next we might want to add some content to our paragraph.

3-2-64

Since it is not yet a node in the DOM we cannot use the `innerText` or similar methods.

In this case we need to use `createTextNode()`

3-2-65

```
let text = document.createTextNode("My paragraph text");
```

appendChild()

3-2-66

Now the actual paragraph and the text is two separate things, and there is nothing that indicates that they belong together.

To add the text to the paragraph we use `appendChild()`

3-2-67

```
paragraph.appendChild(text);
```

The paragraph is still not included in the DOM, though.

3-2-68

We need to append our created element to an existing one.

```
let existingElement = document.getElementsByTagName("div")[0];

existingElement.appendChild(paragraph);
```

So by doing these steps 3-2-69

```
let paragraph = document.createElement("p");
let text = document.createTextNode("My paragraph text");
paragraph.appendChild(text);

let existingElement = document.getElementsByTagName("div")[0];
existingElement.appendChild(paragraph);
```

We get a new paragraph within our div.

```
<body>
    <div>
        <p>Hello</p>
        <p>My paragraph text</p>
    </div>
</body>
```

DOM events 3-2-70

With CSS, the only interaction we could offer was some animations using `:hover` and `:active`.

But now is the time for some **true user interaction**!

The DOM lets us **add event listeners to elements**. 3-2-71

These are **functions that will be called whenever that particular event happen** on that element.

Say we **have this button** in our document: 3-2-72

```
<button id="doomsdaybtn">Don't click me!</button>
```

And a **reference** to the corresponding node:

```
var btn = document.getElementById("doomsdaybtn");
```

We now **create a function to be used as an event listener**... 3-2-73

```
var listener = function(){
  alert("BOOM!");
}
```

...and **attach it using the `addEventListener` method** on the node:

```
btn.addEventListener("click",listener);
```

We could also **add the anonymous function directly**: 3-2-74

```
btn.addEventListener("click",function(){
  alert("BOOM!");
});
```

Now when the user clicks the button, the event handler function will run. 3-2-75

There are three ways to register event handlers for an element in the DOM. 3-2-76

- EventTarget.addEventListener
- HTML attribute
- DOM element properties

Using the `EventTarget.addEventListener` is the preferred way with pure JavaScript to register event handlers.

EventTarget.addEventListener 3-2-77

As we saw in previous example, we select the element we want to target with the event listener.

```
let button = document.getElementById("myButton");

button.addEventListener("click", event => {
    alert("Button is clicked");
});
```

When the button is clicked, the event will be triggered and perform the given task.

In this case it would log *Button is clicked* to the console.

On this element we call the `addEventListener` method into which we pass: 3-2-78

- **what event it should listen to**
- a **callback function**, or **handler function**, what should be done when the event is triggered

The event we pass into the `addEventListener` method can vary. 3-2-79

Common events to listen to are:

- `click` - element gets clicked
- `change` - element changes
- `mouseover` - hover element
- `keydown` - any key down

There is a **full list of events** at MDN:

https://developer.mozilla.org/en-US/docs/Web/Events

HTML attribute 3-2-80

We can also apply event handlers as an attribute on an element.

```
<button onclick="alert('Button is clicked');">Click me!</button>
```

This approach should be **avoided due to separation of concerns**. 3-2-81

Including JavaScript within our HTML makes the markup bigger and a bit messy.

The key is to separate our JavaScript from the HTML-file into a separate JS-file. 3-2-82

The same way we want to separate CSS from our HTML.

DOM element properties 3-2-83

We can register event handlers via properties on a given element.

```
let button = document.getElementById("myButton");

button.onclick = event => {
    alert('Hello world');
};
```

This approach is not that usual, it also limits us to only set one handler per element and per event. 3-2-84

Event object

3-2-85

We have seen the passed argument `event` into the event handler.

```
button.addEventListener("click", event => {
    // handle the event
});
```

This is an object that is created when the event occurs.

The `event` object can be used within the handler function.

3-2-86

```
let button = document.getElementById("myButton");

button.addEventListener("click", event => {
    console.log(event);
    // MouseEvent {isTrusted: true, screenX: 39,
    // screenY: 144, clientX: 39, clientY: 24, …}
});
```

The `event` object contains a bunch of information about the event.

3-2-87

```
let button = document.getElementById("myButton");

button.addEventListener("click", event => {
    console.log(event.target);
    // <button id="myButton">Click me!</button>
});
```

For example `target` gives us information about what element triggered the event.

3-3. Inclusion

Including JavaScript is **extremely similar** to how **CSS inclusion** works. 3-3-1

There are **three ways to include JS** 3-3-2

- script element containing code
- script element referencing a file
- inline event handler in an element attribute

Script element with code 3-3-3

A **script element with code** might look like this:

```
<script type="text/javascript">
    /* ...javascript code here... */
</script>
```

This is a script element in which you write your JavaScript code **within the HTML document**. 3-3-4

This comes handy if it is only a **small amount of code**.

Script element referencing a file 3-3-5

Usually though, we want to **separate our files**, having HTML in a `html` file and JavaScript code in a `js` file.

The same way we want to have separate `css` files.

A **script element referencing a separate file** might look like this: 3-3-6

```
<script type="text/javascript" src="js/mycode.js"></script>
```

Several script elements can be included in your HTML document 3-3-7

```
<script type="text/javascript" src="js/math.js"></script>
<script type="text/javascript" src="js/order.js"></script>
<script type="text/javascript" src="js/employees.js"></script>
```

Note that you **should not self-close the script element**. 3-3-8

This is because the script element **may** contain code, and thus needs to be closed in that case.

The script element is placed within your HTML document. 3-3-9

It can be placed in the head section, within the body or everywhere actually.

But what to keep in mind is that HTML documents is **read from top to bottom**. 3-3-10

This means that the included JavaScript code will be **executed in the order it is placed**.

Due to this behavior, the following would result in an error 3-3-11

```
<script>
    let input = document.getElementById("input");
    console.log(input.getAttribute('value'));
    // Uncaught TypeError: Cannot read property 'getAttribute' of null
</script>
<input id="input" value="My Input">
```

This is because the JavaScript code is executed before the actual element is created, therefore we cannot access it. 3-3-12

For our JavaScript code to work we have to make sure that the **document has loaded with the elements** that the script use. 3-3-13

We can do this simply by placing the script at the **end of the body** element, right before `</body>`. 3-3-14

```
<body>
<input id="input" value="My Input">
<script>
    let input = document.getElementById("input");
    console.log(input.getAttribute('value'));
</script>
</body>
```

This way the elements have loaded before the script executes and can be accessed in the JavaScript code.

Inline event handler

An **inline event handler** looks like this:

```
<button onclick="alert('You clicked, OMG!');">Don't click me</button>
```

As you might have guessed, **inline handlers are a bad idea**, for pretty much the same reasons as inline CSS.

JavaScript Syntax

Sections in this chapter:

4-1. Identifiers

Identifiers are simply names. *4-1-1*

This can be names of variables, functions or labels.

We use these identifiers to target, a variable for example: *4-1-2*

```
var myVariable = "This is my variable";
        ^
        |
        |
    identifier

console.log(myVariable);
                ^
                |
                |
        target variable by
         name (identifier)
```

Or a function: 4-1-3

```
function myFunction() {...}
              ^
              |
              |
          identifier

myFunction();
        ^
        |
        |
target function by
 name (identifier)
```

In JavaScript, an identifier must start with one of the following: 4-1-4

- a letter
- underscore
- dollar sign

The following characters may be letters, digits, underscores or dollar signs.

Valid names: 4-1-5

```
firstName
_firstName
$firstName
_first_name2
```

NOT valid names:

```
-firstName
first-name
*firstName
2firstName
```

Case sensitive 4-1-6

In JavaScript identifiers are **case sensitive**, meaning that the casing of characters matters.

```
var emailAddress = "example@example.com";
var emailaddress = "example2@example.com";
```

emailAddress and **emailaddress** are not the same.

The first one is the one we want to approach though, using camel case

Since JavaScript is case sensitive, when we create a *variable* as in the previous example we have to be sure the keyword `var` is written all lowercase. 4-1-7

`var` is not the same as `Var`.

Casing 4-1-8

There are a lot of different ways of joining multiple words into one.

```
socialSecurityNumber
SocialSecurityNumber
social-security-number
social_security_number
```

camelCase 4-1-9

In JavaScript it is very common to use **camelCase**, for variables, functions and such.

```
function getUserInformation() {
    return {name: "Pelle", age: "37"}
}

var userName = getUserInformation().name;
```

When using camel case we capitalize the first character of each word except the first word. 4-1-10

```
firstName
socialSecurityNumber
emailAddress
getUserInformation
```

PascalCase 4-1-11

As said, there are a lot of ways to structure names.

In JavaScript we use **camelCase** but there are other languages that uses **PascalCase**.

With **PascalCase** we capitalize first letter in every word. 4-1-12

```
GetUserInformation
FirstName
EmailAddress
```

C# is an example of programming language that uses mostly **PascalCase** 4-1-13

```
public void WritePassedValue(int passedValue)
{
    Console.WriteLine(passedValue);
}

WritePassedValue(54);
```

Except for parameters which is written in camel case

kebab-case 4-1-14

In CSS for example it is very common to separate words with a hyphen.

```
.page-heading {
    font-size: 30px;
}

#about-section {
    padding: 20px;
}
```

This can be called **kebab-case**.

Lisp has been using this convention for decades

Using **kebab-case** in JavaScript would give you errors. 4-1-15

```
var my-hyphen-variable = "This is not right";
// Uncaught SyntaxError: Unexpected token -

console.log(my-hyphen-variable);
// Uncaught ReferenceError: my is not defined
```

snake_case 4-1-16

In some cases we choose to separate word by using underscore, this is called **snake_case**.

This is used in for example **C++**.

```
int main() {
    auto myPointer = std::unique_ptr(new MyClass("some setting"));
    myPointer.makeStuff();

    return 0;
}
```

4-2. Keywords

In JavaScript there is a bunch of words that are **reserved**, meaning you can not use them as identifiers, to name variables, functions or labels. 4-2-1

Such can be `break`, `case`, `this`, `if`, `else`.

You can find lists online of all the reserved keyword. 4-2-2

For example on MDN Lexical grammar page, scroll down to the **keywords** section.

These words are reserved for other **already defined purposes** within the JavaScript language. 4-2-3

```
if(x > 0) {
    console.log(x + ' > 0')
}
```

If you use these words alone for naming variables, functions or labels you will get an error. 4-2-4

```
var if = "This is not right";
// Uncaught SyntaxError: Unexpected token if
```

But you can of course use them in **combination with other words**. 4-2-5

```
var ifSomething = "This will work";
```

4-3. Operators

Operators are used to perform **actions on variables and values**, such as calculations, comparison or concatenation. 4-3-1

```
var x = 4;
var y = x + 6;
// y = 10

var str = "Snow";
var str2 = str + "ball";
// str2 = Snowball

var amount = 7;
console.log(amount === 3);
// Returns false
```

Some common operators in JavaScript: 4-3-3

Assignment (=)

Used to assign a value, to a variable for example.

```
var amount = 6;
```

*Math (+, -, * and /)*

Used to perform a calculation or concatenation.

```
var addition = 3 + 3;
var subtraction = 3 - 3;
var multiplication = 3 * 3;
var division = 3 / 3;

var firstName = "Sean";
var lastName = "Banan";
var fullName = firstName + lastName;
```

*Compound Assignment (+=, -=, *= and /=)*

Combine a math operation with assignment. Can be used when we want to append a number or a string.

```
var sum = 0;
for(i = 0; i <= 10; i++) {
    sum += i;
}
```

Using += is the same thing as if we would write `sum = sum + i`

Increment (++)

Increments the previous value by one

```
var counter = 0;
for(i = 0; i <= 10; i++) {
    counter++;
    //Increments the count by one
}
```

Decrement (- -)

Decrements the previous value by one

```
var counter = 10;
for(i = 0; i <= 10; i++) {
    counter--;
    //Decrements the count by one
}
```

Object Property Access

`.` as in `console.log`. Objects are values that hold other values at specific named locations called properties. `obj.a` means an object value called `obj` with a property of the name `a`. Properties can alternatively be accessed as `obj["a"]`. We will cover this in the next section.

```
var obj = {
    name: "Sean Banan",
    age: 34
}

console.log(obj.name); // Sean Banan
console.log(obj.age); // 34
```

Equality (== loose-equals, === strict-equals, != loose not-equals, !== strict not-equals)

Used to compare the equality of values.

```
var x = 4;
if(x === 4) {
    console.log("SAME!")
}
```

Loose vs Strict

`Loose` equality or not-equal checks the value that is being compared.

```
console.log(3 == 3) // true
console.log(3 == '3') // true

console.log(3 != '3') // false
console.log(3 != 3) // false
```

`Strict` equality or not-equal checks both the actual value AND the data type of this value.

```
console.log(3 === 3) // true
console.log(3 === '3') // false

console.log(3 !== 3) // false
console.log(3 !== '3') // true
```

Comparison (< less than, > greater than, <= less than or loose-equals, >= greater than or loose-equals)

Used to compare if values are greater than or less than.

```
console.log(3 > 2) // true;
console.log(3 < 2) // false;

console.log(3 <= '3') // true;
console.log(3 >= 3) // true;
```

Logical (&& AND, || OR)

These operators are used to express compound conditionals.

```
var x = 4;

if(y && x) {
    //Won't enter this block, y is not defined
}

if(y && x) {
    //Will enter this block, one of them is defined
}
```

There is a whole bunch of these expressions and operators.

You can find a list of the available ones here

4-4. Values and Types

JavaScript has 7 types: 4-4-1

- `null`
- `undefined`
- `boolean`
- `string`
- `number`
- `object`
- `symbol`

Type	Meaning	4-4-2
null	Absence of object value, or empty object (type object)	
undefined	Variable that never had an assigned value, has been declared or does not exist	
number	Positive or negative numbers	
string	Text such as "Hello"	
boolean	Logical value, true or false	
object	Can hold several values	
symbol	New in ES6	

Primitives	Composite data types	4-4-3
null	object	
undefined	array (technically an object)	
boolean		
string		
number		
symbol		

The **primitives** include 4-4-4

- `null`
- `undefined`
- `boolean`
- `string`
- `number`
- `symbol`

There are also **composite data types** 4-4-5

- `object`
  - `array`(technically an object)

Primitive data types can be referred to as **simple types**. 4-4-6

Composite data types can be referred to as **complex types**, or **reference types**.

typeof 4-4-7

We can use *typeof* to get what type a specific value is.

```
var x = 4;

console.log(typeof x); // number
console.log(typeof "Hello"); // string
console.log(y); // undefined
```

Primitives / Single values 4-4-8

Primitives are single values, with no special capabilities.

```
var myNumber = 3; //Number
var myString = "My String"; // String
var undefinedVariable; // Undefined
```

When *changing* a primitive value a `new value` in memory will be created. 4-4-10

Symbol 4-4-11

`symbol` is mentioned here for completeness. Symbols can be used as a kind of "private" key in objects. We will come to this later in the course.

Reference values 4-4-12

An object in JavaScript is a **collection of key-value pairs**, often called a **dictionary** in other languages.

The **property names (keys) are always strings**, and the **property values** can be anything.

We create an **object literal** like this, using the **curly brackets**: 4-4-13

```
var person = {
    name: "Therése",
    age: 37
}
```

Arrays in JavaScript are **technically also objects.** 4-4-14

Arrays is a collection of data that can be iterated through, they are defined using the **hard brackets**.

```
var myArr = [2, 3, 4, 5];
```

We can create arrays with simple values or with objects. 4-4-15

```
var myArr = [{ name: "Bo", age: 34 }, { name: "Mark", age: 23 }]
```

Memory allocation 4-4-16

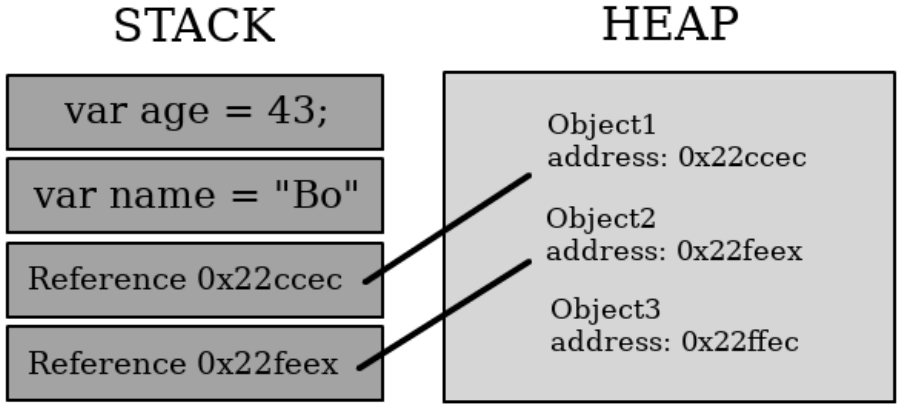
When we create variables or functions in JavaScript, memory gets allocated.

```
var x = 4; // allocates memory for a number

function myFunction(x) {
    return x;
} // allocates memory for a function (callable object)
```

When we create or use variables or functions we basically **write and read to the allocated memory**. 4-4-17

In memory we have something called **stack** and **heap**. 4-4-18



These are locations where data gets stored. 4-4-19

Primitive values are stored on the **stack** and reference values are stored on the **heap**.

Stack 4-4-20

The stack is a **stack of data stored in your memory**.

"Mark Zuckerberg"
423523
false

The stack is a small region of memory that keeps **simple values, such as numbers or strings, the primitives**. 4-4-21

It is very quick to access but also somewhat limited.

For each created simple value, a new entry on top of the stack is entered. 4-4-22

```
var firstName = "Bo";
// The string Bo will be stored on top of the stack
console.log(firstName) // Bo
var fullName = firstName;
// The value Bo will be created as a new value on the stack
console.log(fullName)
// Bo
firstName = "Klas";
// The value Klas will be created as a new value on the stack
console.log(fullName)
// Bo
```

New data gets added structurally **on top of the stack**, on top of the other already existing values. 4-4-23

var myVar = "Klas"

"Klas"
"Bo"
"Bo"
"Mark Zuckerberg"
423523
false

The stack also stores pointers, **or references**, to an address on the **heap** where objects are stored. 4-4-24

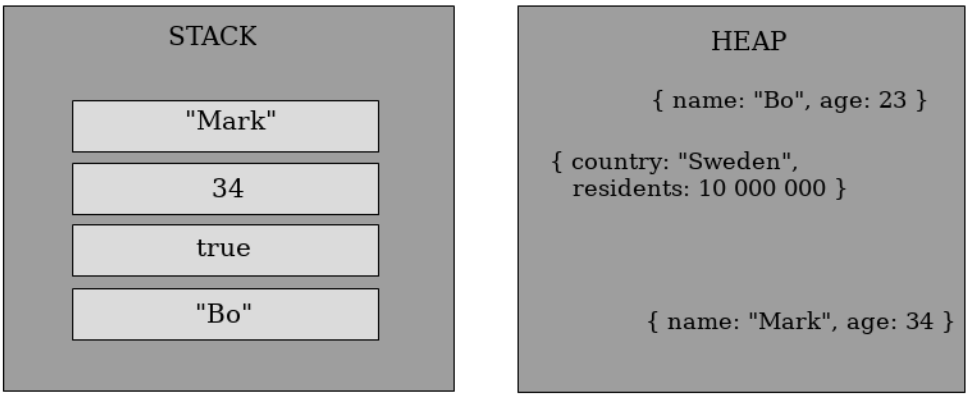
Heap 4-4-25

The heap is another place in memory in which data gets stored.

It is able to hold **more information**, which makes it slower to access compared to the stack that keeps simple values.

The values on the heap is often **bigger amount** of data and **data that changes frequently or dynamically**, such as objects. 4-4-26

Unlike the stack, the heap is not stacked or structured the same way. 4-4-27



On the stack values get stored properly on top of each other but on the heap the values are stored in somewhat random order. 4-4-28

To keep track of each stored value, **they all have unique memory addresses**. 4-4-29

```
                        --- Stack ---                --- Heap ---
var myObject = {
    name: "Bo",  ---->   myObjectPointer        {name:"Bo", age: 34}
    age: 34                 0x22ecce                  0x22ecce
}

// The actual data of the object will be stored on the heap
// On the stack we store a pointer that points to
// the address of where the data is located on the heap
```

The heap holds complex types, such as an object. 4-4-30

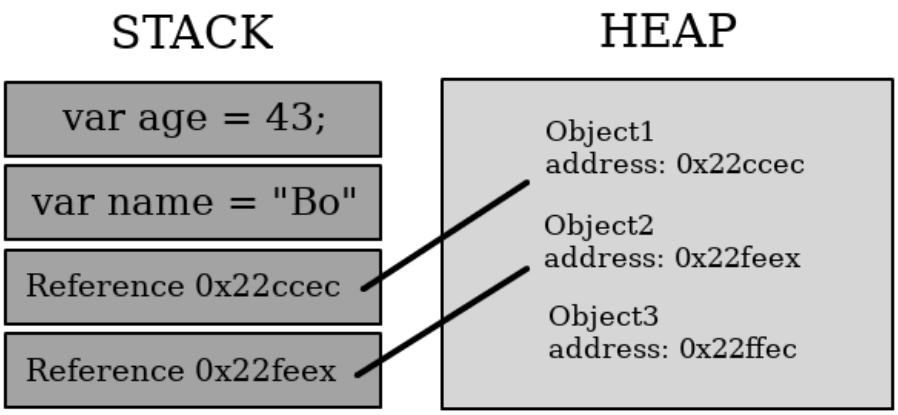
```
var myObj = {
    name: "Bo",
    age: 43
}

// The value within the curly brackets will be stored on the heap
// {
//    name: "Bo",
//    age: 43
// }
```

The stored object will have a unique memory address such as `0x22effe`, a hex number.

The stack in turn stores a **pointer to the object on the heap**, with help of this unique address 4-4-31

4-4-32



When the interpreter analyzes the code, it decides where the data should be located, on the heap or the stack. 4-4-33

Memory - life cycle 4-4-34

In all languages the memory life cycle looks pretty much the same:

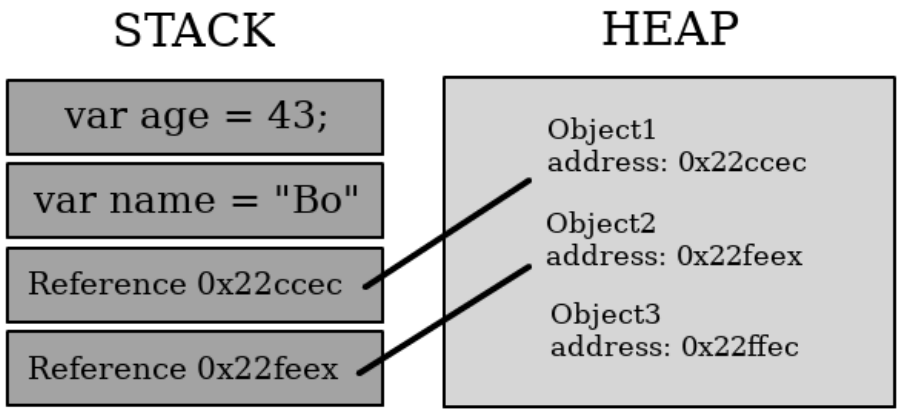
- Allocate the memory you need
- Use the allocated memory (read, write)
- Release the allocated memory when it is not needed anymore

Garbage collection 4-4-35

JavaScript is a **garbage collection language**.

This means that when a value in memory is not used anymore it will be *freed* from memory.

When an object does not have any pointers to it, it is considered garbage and can be removed. 4-4-36



Object 3 has no pointer to it

Access values 4-4-37

As previous said, when creating or using variables or function we write and read the allocated memory.

Access simple values 4-4-38

To access the primitives, such an variable holding a string or a number, we simple call its identifier.

```
var mySimpleValue = "Hello there";

console.log(mySimpleValue);
```

Access object values 4-4-39

There are **two different ways to access a value in an object**

We can use **dot notation**:

```
var person = {
    name: "Bo",
    age: 43
}

var name = person.name; // Bo
```

Or, we can use **bracket notation**: 4-4-40

```
var age = person["age"]; // 43
```

Bracket notation also allows us to do **dynamic lookup**: 4-4-41

```
var age = 43;
var myObject = person[age]; // 43
```

We can add a new property simply by assigning to it: 4-4-42

```
person.address = "Drottninggatan 57";

console.log(person);
//{
//     name: "Bo",
//     age: 43,
//     address: "Drottninggatan 57"
// }
```

And we **delete a property** using the `delete` keyword: 4-4-43

```
delete person.address;

console.log(person);
//{
//     name: "Bo",
//     age: 43,
//}
```

If we **access a non-existing property** we always get `undefined`. 4-4-44

```
console.log(person.phoneNumber); // undefined
```

Comparison 4-4-45

Contrary to primitives, **objects are references**, which means comparing to objects will give us false since that is exactly what they are, **two different objects**:

```
var person1 = { name: "Bo" };
var person2 = { name: "Bo" };

console.log(person1 === person2)
```

This is due to the fact that when we create an object, the object gets stored on the heap with a unique address, and a **reference** to that object is stored on the stack. 4-4-46

```
var person1 = { name: "Bo" };
var person2 = { name: "Bo" };

console.log(person1 === person2); // false
```

So, in the above code, the reference for the `person1` object is **not** the same as the reference for the `person2` object, all though the contents are the same.

If we on the other hand assign a new variable with an already created obejct, the comparison will show true.

```
var person1 = { name: "Bo" };
var person2 = person1;

console.log(person1 === person2); // true
```

Now the reference for both objects will be the same.

Since **objects are references** they are **always truthy**:

```
if ({}){
  console.log("An empty object is truthy!"); // will be shown!
}
```

We can **iterate over objects** using a **`for..in`loop**:

```
for (var key in person){
  console.log(key + " has value " + person[key]);
}
```

An **object can contain other objects**:

```
var me = {
  name: "Therése",
  address: {
    street: "Drottningatan 57",
    zip: 25222,
    city: "Helsingborg"
  }
}
```

We can **access properties to any depth**:

```
var myZip = me.address.zip;
```

A special kind of objects are **arrays**, which are an **ordered list of values**. 4-4-53

You could say that they are **objects where the keys are always named 0, 1, 2....**

We can **create arrays** using the **array literal notation**: 4-4-54

```
var list = ["tomatoes",42,false];
```

Since **arrays are objects** we **access elements as before**: 4-4-55

```
var firstItem = list[0]; // "tomatoes"
```

An array **has a length property** that tells us how many elements it contains: 4-4-56

```
list.length // 3
```

Which means I can **pick the last item** by doing this: 4-4-57

```
var lastItem = list[list.length-1];
```

We can **add a new value** by calling the `push` method: 4-4-58

```
list.push("foobar");
list.length // 4
```

It is common to **iterate using a regular for loop**: 4-4-59

```
for(var i=0; i < list.length; i = i+1){
  console.log("Item "+i+" is "+list[i]);
}
```

There are a **whole bunch of array methods** - see the MDN reference for a full list: 4-4-60

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Q The `length` property was rather convenient - is there something **similar on regular objects**? 4-4-61

A Not directly, but we can get an **array of all keys for an object** using the `keys` method of the global `Object` object:

```
var keys = Object.keys(person); // ["name","age", "address"]
```

4-5. Numbers

As we know, number is a type, a primitive type. 4-5-1

```
var x = 4;
```

As in the real world, we have different forms of numbers, like negative numbers and numbers with decimals for example. 4-5-2

```
var x = -3;
var y = 3.4;
```

But we do **not have different types of numbers**, like integers, float and such like some other languages. 4-5-3

Numbers in JavaScript are always stored as **double precision floating point number**, no matter what form it might be.

The floating-point standard is called **IEEE 754**. 4-5-4

Many other programming languages, Java and C# included, implement this standard.

```
-5.5        // negative number
-0          // negative zero
+0          // zero
1234.567    // positive number
```

There are also a few special values: 4-5-5

```
-Inf        // negative infinity
NaN         // "not a number", from a failed calculation
+Inf        // infinity
```

One very important thing to know about numbers in JavaScript is that they are a **pragmatic approximation of real numbers**. 4-5-6

There is "necessary" precision loss involved.

Numbers without a period, also known as integers, are definite up to 15 digits. 4-5-7

With this comes that numbers that are bigger than 15 digits *might* not be the exact value they should be.

```
var u = 999999999999999; // 999999999999999
var i = 9999999999999999; // 10000000000000000

var x = 123456789011121; // 123456789011121
var y = 123456789123456789; // 123456789123456780
```

Number of decimals is 17 as maximum, but floating point arithmetic *might* not always be completely accurate, as with integers. 4-5-8

```
var x = 0.1 + 0.2; // 0.30000000000000004
var y = 0.1 + 0.4 // 0.5
```

Numbers as strings 4-5-9

We can have numbers as the content of a string.

```
var x = "20";
```

Q But does JavaScript handle it as a number or a string? 4-5-10

A Well, that depends! 4-5-11

As we've seen before, in JavaScript we have different operators to work with strings and numbers.

+ operator 4-5-12

In JavaScript the + **operator** is used for both addition and concatenation.

We can use it as regular addition of one or several numbers. 4-5-13

```
var x = 4;
var y = 2;

var u = x + y; // 6
```

But if both the + operator and a string with number as content is involved in a calculation, JavaScript will read it as a string. 4-5-14

```
var x = "twenty";
var y = "20";
var u = "10";

var n = x + y; // "twenty20"
var m = y + u; // 2010
```

The result will be a string concatenation.

If the values are numbers, the + operator will count as addition of numbers. 4-5-15

```
var x = 2 + 2; // 4
```

If the values are strings only or strings and numbers, the + operator will count as string concatenation.

```
var x = 2 + "2"; // 22
```

Other operators 4-5-16

With other operators on the other hand, *, / or -, the content of a string which is a number will not be read as a string.

The JavaScript compiler will try to convert the string into a number. 4-5-17

```
var x = 10;
var y = "60";
var u = "30";

var n = x * y; // 600
var m = y / u; // 2
```

So doing operation where a string counts as a number work for all the operators but addition. 4-5-18

```
var x = "100";
var y = "10";

var n = x + y; // 10010
var m = x * y; // 1000
var o = x / y; // 10
var p = x - y; // 90
```

NaN

4-5-19

NaN stands for **Not a Number**, but it is still of the type *number*.

This type indicates that a given number is not valid as a number.

```
var x = 5 / "NotValid"; // NaN, we cannot divide 5 with "NotValid"
```

We can check if something is `NaN` by using the `isNan()` function.

4-5-20

```
var x = 5 / "NotValid";
var y = 5 / 5;

var m = isNaN(x); // true
var n = isNaN(y); // false
```

Infinity

4-5-21

In JavaScript we have two values for infinity, `-Infinity` and `+Infinity`

If the result of a calculation is outside the largest number possible, this will be returned.

`+Infinity` is the result of reaching the largest positive number, whereas `-Infinity` is the result of the largest negative number.

4-5-22

```
var myNumber = 2;
while (myNumber !== Infinity) {
    myNumber = myNumber * myNumber * 100000;
    console.log(myNumber);
}

// The output would be the following:
//  400000
//  16000000000000000
//  2.56e+37
//  6.5536e+79
//  4.294967295999999e+164
//  Infinity
```

Scientific (exponent) notation

4-5-23

As we could see in the result from the `while loop` some of the values had something like `e+37` at the end.

```
//  400000
//  16000000000000000
//  2.56e+37
//  6.5536e+79
//  4.294967295999999e+164
//  Infinity
```

This is a quick way to write large numbers, called **exponent notation**. *4-5-24*

```
var x = 324e+6; // 324000000, 6 zeroes
var y = 324e-6; // 0.000324, 6 decimals
```

- +: defines if it is zeroes *4-5-25*
- -: If we want decimals instead

The following number after the operator defines how many zeroes or decimals it should be.

4-6. Strings

Strings can be close to anything, in text form. 4-6-1

```
var str1 = "Hello world";
var str2 = "12345";
```

When we write string literals we use **quotation marks**, doubles or simple. 4-6-2

```
var str1 = "Hello world!";
var str2 = 'Hello world!';
```

Quotation marks within a string 4-6-3

There are occasions when we want to have quotation marks within our strings, for words like **It's** or **we "like" it**.

If we use the same quotation mark to define our string literal as the one we want within, we'd get an error. 4-6-4

```
var str = "We "like" it";
// Uncaught SyntaxError: Unexpected identifier
var str1 = 'It's a beautiful day';
// Uncaught SyntaxError: Unexpected identifier
```

Instead we want to use the opposite quotation mark than the one we want within the string 4-6-5

```
var str = 'We "like" it';
// "We "like" it"
var str1 = "It's a beautiful day";
// "It's a beautiful day"
```

Template literals 4-6-6

As of ECMAScript 2015, string literals can be so-called **template literals or template strings**.

Template strings are defined using **backtick (`)**. 4-6-7

```
var templateString = `This is my template string`;
```

This allows us to insert expressions into our string literals. 4-6-8

```
var templateString = `2 + 2 = ${2+2}`; // 2 + 2 = 4
```

To get the same result using a regular string we would need to end our string and concatenate it with the result of the expression. 4-6-9

```
var regularString = "2 + 2 = " + (2 + 2); // 2 + 2 = 4
```

We can also create a multi line string using template literals. 4-6-10

```
var templateString = `Hello.

Lets learn about template strings.

This is fun.`

//"Hello.
//
//Lets learn about template strings.
//
//This is fun."
```

Doing this using regular strings would give us an error. 4-6-11

```
var regularString = "Hello.

Lets learn about template strings.

This is fun."
// Uncaught SyntaxError: Invalid or unexpected token
```

It will think that we forgot to close the string.

To avoid this error we would have to concatenate several strings if we want to write it on several lines. 4-6-12

```
var regularString = "Hello." +
"Lets learn about template strings." +
"This is fun."
// "Hello.Lets learn about template strings.This is fun."
```

But writing the above would not result in line breaks. Instead, the string would be printed as below: 4-6-13

```
// "Hello.Lets learn about template strings.This is fun."
```

To achieve the result with newlines we would have to use escape notation.

Escape notation 4-6-14

Within our string literals we can use special characters using escape notation, such can be `newline`, `tab`, `quotes`, amongst many.

We define escape notation within a string literal with a **backslash (\)** 4-6-15

```
var regularString = "Hello.\n\nLets learn about template strings.\n\nThis is fun."
//"Hello.
//
//Lets learn about template strings.
//
//This is fun."
```

So we can see that with **template string** we get a quicker and more visually logic way to structure our string.

```
var templateString = `Hello.

Lets learn about template strings.

This is fun.`

//"Hello.
//
//Lets learn about template strings.
//
//This is fun."
```

The above gives the same result as the following

```
var regularString = "Hello.\n\nLets learn about template strings.\n\nThis is fun."
//"Hello.
//
//Lets learn about template strings.
//
//This is fun."
```

We earlier looked at including quotation marks within our strings. 4-6-16

Instead we want to use the opposite quotation mark than the one we want within the string

```
var str = 'We "like" it';
// "We "like" it"
var str1 = "It's a beautiful day";
// "It's a beautiful day"
```

For this we can also use escape notation. 4-6-17

```
var doubleQuotationMarks = "We \"like\" it";
var singleQuotationMarks = 'It\'s beautiful';
```

This can be handy if we in one string want to include both forms of quotation marks.

Length of a string 4-6-18

We can check the length of a string, meaning how many characters it contains, using `.length`

```
var str = "Hello, check the length!";
console.log(str.length);  // 24
```

The length of a string also includes whitespace.

Methods on strings 4-6-19

We can modify our strings with functions on the string type.

```
var str = "HelLo, HoW aRE YOU todAY?" // "HelLo, HoW aRE YOU todAY?"
var str2 = str.toLowerCase(); // "hello, how are you today?"
var str3 = str2.toUpperCase(); // "HELLO, HOW ARE YOU TODAY?"
```

Some of the functions on strings:

- `toLowerCase()`
- `toUpperCase()`
- `charAt()`
- `concat()`
- `indexOf()`
- `slice()`
- `replace()`

Characters and substring

4-6-20

A string is an array of characters. Each character has a specific index, where the first index in the string is `0`.

```
var str = "Hello you";
           012345678
```

We can access part of a string with functions on the string type, such as getting the character with specific index in the string using `charAt(x)`.

4-6-21

```
var str = "Hello you";
console.log(str.charAt(1)); // e
```

We can also get a part of the string using `substring(start, end [optional])`

4-6-22

```
var str = "Hello you";
var str2 = str.substring(1); // ello you
```

If we pass one number as the parameter to the `substring` function, we will get a new string that includes the characters from the given index to the end.

4-6-23

If we pass two parameters to the `substring` function, we tell at what index our substring should start and end.

4-6-24

```
var str = "Hello you";
var str2 = str.substring(1, 6); // ell
```

The substring will not include the character with index 6

When specifying start and end indices for a substring, the extracted substring will include the characters from the given start index up to the given end index, but it will not include the character at the end index.

4-6-25

4-7. Date object

The Date object is used to represent date and time in JavaScript. 4-7-1

Objects created are based on a time value that is the number of milliseconds since **1 January 1970 UTC.**

The reason for this exact date, **1 January 1970 UTC**, is that it is the Unix epoch, meaning it was around that time the Unix operating system was created. 4-7-2

To get a Date for the current date, time and time zone, you can use `new Date()` to create a new date instance. 4-7-3

```
var today = new Date(); // Fri Dec 22 2017 12:14:51 GMT+0100 (CET)
```

There are different ways to create a Date object, depending if we want the current date or a custom one. 4-7-4

```
new Date();
// Current time
new Date(value);
// Value representing the number of milliseconds since January 1, 1970 00:00:00
new Date(dateString);
// A string representing a date, need to be specific format
new Date(year, month, day, hours, minutes, seconds, milliseconds);
// Integers representing date and time
```

new Date(value);

4-7-5

```
var date = new Date(100e+10);
// Sun Sep 09 2001 03:46:40 GMT+0200 (CEST)
```

The value passed into new Date() represents the number of milliseconds since **January 1, 1970 00:00:00** 4-7-6

new Date(dateString);

4-7-7

```
var date = new Date("December 4, 1994 04:04");
// Sun Dec 04 1994 04:04:00 GMT+0100 (CET)
```

The `dateString` you pass in must be formatted correctly, the complete format is as follows: **YYYY-MM-DDTHH:mm:ss.sssZ**.

4-7-8

You can find the format and how to split it here

There are different ways to structure this format though, not every part needs to be included.

4-7-9

```
var date = new Date("December 4, 1994 04:04");
// Sun Dec 04 1994 04:04:00 GMT+0100 (CET)
var date2 = new Date("1994-12-04T04:04:04");
// Sun Dec 04 1994 04:04:04 GMT+0100 (CET)
var date3 = new Date("1994-12-04");
// Sun Dec 04 1994 01:00:00 GMT+0100 (CET)
```

new Date(year, month, day, hours, minutes, seconds, milliseconds);

4-7-10

You can create a custom date object by passing integers representing the date or time.

Lets try it out:

4-7-11

```
var date = new Date(1994, 12, 04);
// Wed Jan 04 1995 00:00:00 GMT+0100 (CET)
```

Q MY EYES ARE FOOLING ME!

4-7-12

Passing in **1994-12-04** gave us **1995-01-04**?

A No, you saw correct!

4-7-13

But we do not want it like that.

The numeric representation of months in JavaScript starts on `0`.

4-7-14

Meaning, we have 12 months but in JavaScript the **numeric representation of December is 11**.

It is like the indices of characters in strings.

Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
0	1	2	3	4	5	6	7	8	9	10	11

4-7-15

To get the correct date we have to keep this in mind

4-7-16

```
var date = new Date(1994, 11, 04);
// Sun Dec 04 1994 00:00:00 GMT+0100 (CET)
```

Same goes for weekdays, almost...

4-7-17

The week in JavaScript, same as the english standard, start on a Sunday. So the numeric representation for the Sunday is 0, 1 for Monday and so on.

Sun	Mon	Tue	Wed	Thu	Fri	Sat
0	1	2	3	4	5	6

4-7-18

Date Instance Methods

4-7-19

There is a whole bunch of methods to use on a date instance.

Such as `getMinutes()`, `setHours()`, `toDateString()`.

Getter methods

4-7-20

We have methods to get data out of a Date object.

```
var date = new Date(1994, 11, 04);
var dayOfMonth = date.getDate(); // 4
var dayOfWeek = date.getDay(); // 0
var year = date.getFullYear(); // 1994
```

Some of the getter methods for the Date object:

4-7-21

- `getDate()`
- `getDay()`
- `getMonth()`
- `getMinutes()`
- `getHours()`

You can find a full list of getters here

Conversion getters

4-7-22

We have methods to return the date in different versions.

```
var date = new Date(1994, 11, 04);
// Sun Dec 04 1994 00:00:00 GMT+0100 (CET)
var dateString = date.toDateString();
// "Sun Dec 04 1994"
var isoString = date.toISOString();
// "1994-12-03T23:00:00.000Z"
var str = date.toString();
// "Sun Dec 04 1994 00:00:00 GMT+0100 (CET)"
```

Some of the conversion getters for the Date object:

4-7-23

- `toTimeString()`
- `toString()`
- `toDateString()`
- `toJSON()`

You can find a full list of conversion getters here

Setter methods

4-7-24

We also have methods to set values on the Date object, if we for example want to change the Date late on.

```
var date = new Date(1994, 11, 04);
// Sun Dec 04 1994 00:00:00 GMT+0100 (CET)
date.setDate(27);
// Tue Dec 27 1994 00:00:00 GMT+0100 (CET)
date.setHours(4);
// Sun Dec 04 1994 04:00:00 GMT+0100 (CET)
```

Some of the setter methods for the Date object:

4-7-25

- `setDate()`
- `setHours()`
- `setMinutes()`
- `setTime()`
- `setFullYear()`

You can find a full list of setters here

4-8. Variables and Blocks

We have seen a lot of variables along the way, now it is time to understand what it is. 4-8-1

Variables exist in many other programming languages and are **used to store data**.

A variable is **declared** with the **var keyword followed by an identifier**, a name which will be used to access the variable. 4-8-2

```
var identifier; // undefined
```

Global vs local 4-8-3

We can access variables without declaring them, these are **global variables.**

All undeclared variables are global variables, meaning we can access them from anywhere. 4-8-4

```
y = "20";
```

Undeclared variable meaning we have not created it with the var keyword but only an identifier.

An undeclared variable does not exist until we assign it a value 4-8-5

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

```
x = 23;
console.log(x); // 23
```

As said, global variables can be accessed from anywhere. 4-8-6

```
function executeMe() {
    x = "Hello";
    var y = "World";
}

executeMe();

console.log(x); // Hello
console.log(y); // Uncaught ReferenceError: y is not defined
```

Q Why can we access one but not the other? 4-8-7

A As well as global variables, we have **local variables.** 4-8-8

As soon as we use the **var** keyword, we create a variable that **belongs to the function scope in which it is was declared**. 4-8-9

In previous versions of EcmaScript we only had one type of scopes, which is function scopes. 4-8-10

The following would work, since because `i` is not within a function it becomes a global variable 4-8-11

```
for(var i = 0; i <= 10; i++) {
    console.log(i); // logs 1 to 10
}

console.log(i); // 10
```

But, as soon as `i` is wrapped within a function it becomes inaccessible from the outside. 4-8-12

```
function counter() {
    for(var i = 0; i < 10; i++) {
        console.log(i); // logs 1 to 10
    }
}
counter();
console.log(i); // Uncaught ReferenceError: i is not defined
```

Local variables are limited to the enclosing function's scope. They are **not** accessible from the outside. 4-8-13

```
function executeMe() {
    var y = "World";
    console.log(y); // World
}

executeMe();

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

Declared variables within a function are so-called dead outside the function, *outside the scope.* 4-8-14

Assignment 4-8-15

If we declare a variable without assigning any value, it will be `undefined` until we do.

```
var identifier; // undefined
identifier = "Some value"; // Some value
```

Variables can be **assigned with values of any type**, such as functions, numbers, strings and so on. 4-8-16

```
var foo = "foo"; // foo is of type string
var bar = 42; // bar is of type number
```

Loosely typed 4-8-17

In JavaScript, variables are **loosely typed**, meaning variables of certain **types** can change during the course of their lifetime:

```
var foo = "foo"; // foo is of type string

// later in the code
foo = 42; // foo is of type number
```

We do not explicitly define what type a variable should be, it rather becomes the type of its value. 4-8-18

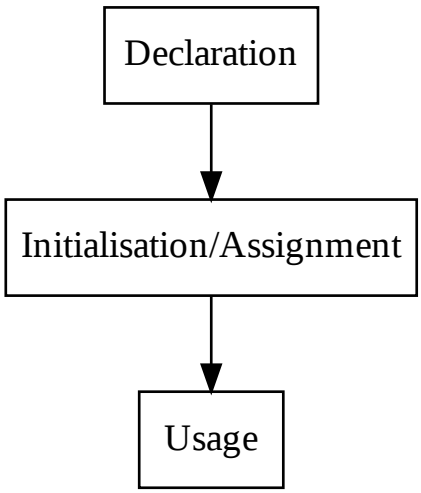
In statically typed languages, like Java or C#, assigning value of another type would result in a error: 4-8-19

```
int age = 37;

// later in the code
age = "foo"; // Error: Cannot implicitly convert type 'string' to 'int'
```

Hoisting 4-8-20

Variable declarations are processed before the execution of the code.



Thus, the code below: 4-8-21

```
var foo = "bar";
```

will look like this to the interpreter:

```
var foo;
foo = "bar";
var foo = "bar";
```

It moves the declaration to the top of its scope, or in other words the **enclosing function block** that it belongs to. 4-8-22

This is called **hoisting**. 4-8-23

In the background JavaScript first declares the variable, then initializes it.

We could write this: 4-8-24

```
function myFunction() {
    console.log("console.log in function");
    var bar = "foo";
}

var foo = "bar";
```

The interpreter will read it like: 4-8-25

```
var foo;
function myFunction() {
    var bar;
    console.log("console.log in function");
    bar = "foo";
}

foo = "bar";
```

`foo` and `bar` is being **hoisted**.

So, variables can be used **before** they are declared. But they will be initialised with a value of **undefined**. Because of this, you should always **declare** and **initialise** your variables **before** you use them. 4-8-26

"use strict" 4-8-27

As of ES5, to make sure that we always declare a variable before using it, we can use `strict` mode.

The strict mode prevents us from accidentally creating global variables. 4-8-28

Add "use strict" at the very top of your JavaScript file, if you want it to be applied to the entire code. 4-8-29

```
"use strict"

function myFunction() {
    x = 3;
}

myFunction(); // Uncaught ReferenceError: x is not defined
```

You can also use it for only specific functions. 4-8-30

```
// non-strict code
x = 3;

function myFunction() {
    "use strict";
    y = 3; // Uncaught ReferenceError: x is not defined
    var u = 3; // We need to declare our variables
}
```

Using the `use strict` we get an error when we write *insecure* JavaScript, such as undeclared variables. 4-8-31

Q How does JavaScript know what is applicable based on this string, `use strict`? 4-8-32

A New compilers recognize the `use strict` string. They know what is applicable when this is present. 4-8-33

Good to keep in mind is that strict mode behaves differently in different browsers. 4-8-34

var vs let 4-8-35

In ES5 we only had one way to declare variables, using the `var` keyword.

In ES6 we get introduced to the `let` keyword. 4-8-36

Unlike `var` that lives inside function scope, `let` lives inside the block in which it is declared, not necessarily a function.

Imagine this code declaring variables with the `var` keyword. 4-8-37

```
for(var i = 0; i <= 10; i++) {
    console.log(i);
}

console.log(i);
```

This works just fine without any errors, we can access `i` outside the for-loop.

Doing this with `let` would not work. 4-8-38

```
for(let i = 0; i <= 10; i++) {
    console.log(i);
}

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

This is because `let` lives within the block in which it is declared. 4-8-39

One can say that the `let` is the new `var`, using `let` is the preferred option. 4-8-40

Unless there are no specific reasons to stick with `var`.

const 4-8-41

`const` is the exact same thing as `let`, except that you can not reassign it.

```
const PI = 3.142;
```

A const is not completely **immutable**, but prevents reassignment of a variable. 4-8-42

Q But wait, what does **immutable** mean? 4-8-43

A Something that is immutable **cannot be modified** after it is created. 4-8-44

Trying to change the value of a const would result in an error. 4-8-45

```
const PI = 3.142;
PI = 300; // Uncaught TypeError: Assignment to constant variable.
```

But if we have an object, we could change the value of a property within that object. 4-8-46

```
const x = {
    y: 3
}

x.y = 4; // Works fine!
x = { } // Uncaught TypeError: Assignment to constant variable.
```

But we cannot reassign the actual variable.

`let` and `const` will not use hoisting. 4-8-47

They will exists and be accessible when they actually are created.