JavaScript Syntax

Sections in this chapter:

4-1. Identifiers

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

This can be names of variables, keywords, 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 tough, 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 word 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

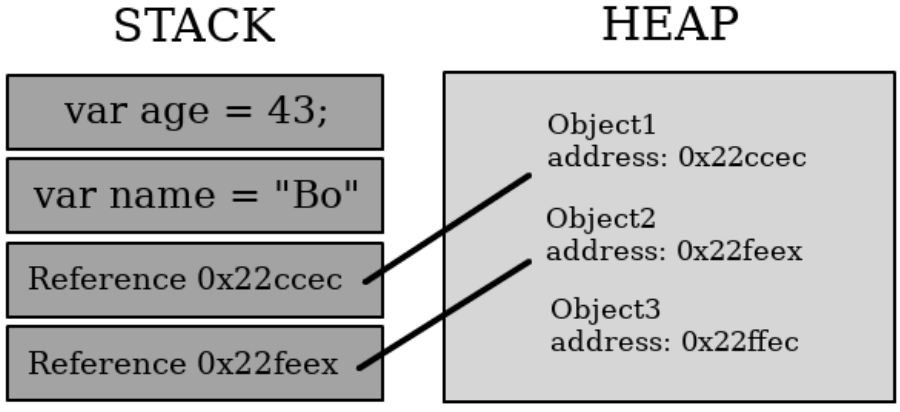
When we create variables or functions in JavaScript, memory gets allocated.

```
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

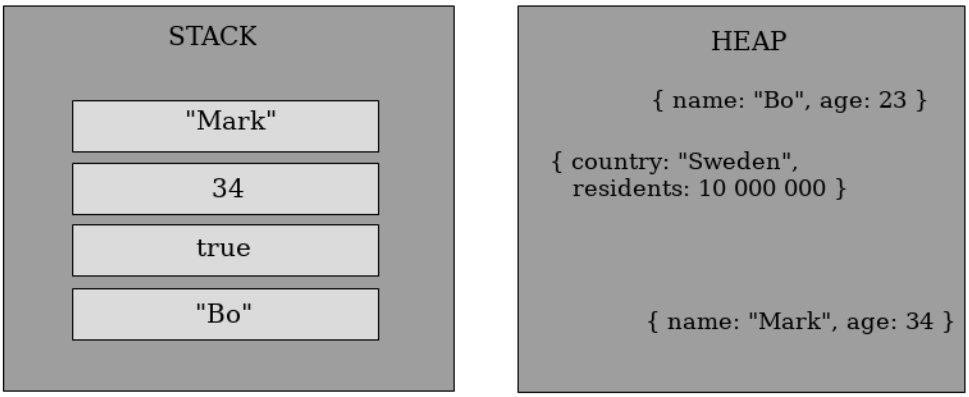
Heap 4-4-25

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

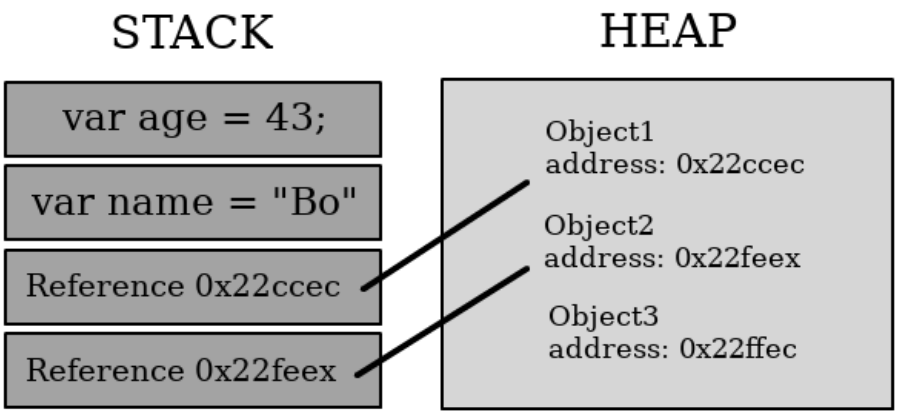
```
var myObj = {
    name: "Bo",
    age: 43
}

// 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:

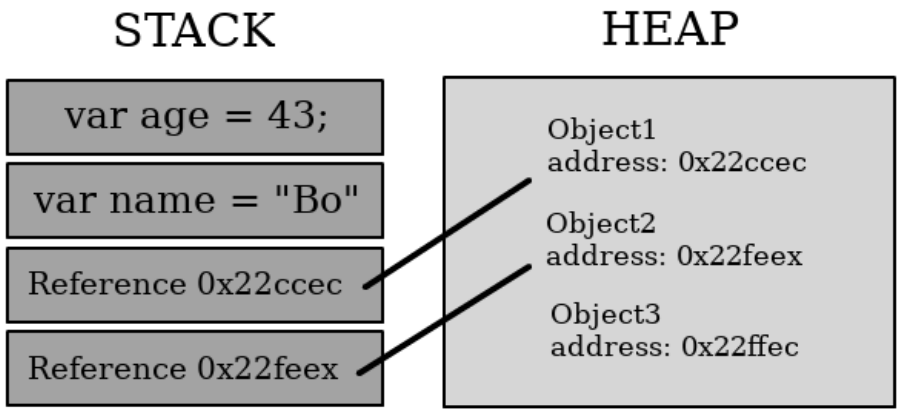
- Allocate the memory you need
- Use the allocated memory (read, write)
- Release the allocated memory when it is not needed anymore

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"]
```