Introduction

In Object Oriented Browser Programming course one of the objectives is listed:

- The Student is able to design and implement programs by using Object Oriented programming principles.

Javascript Data Types

There is a handful of basic building blocks, known as primitive data types available in JavaScript (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures).

Use typeof to read the datatype.

Boolean

Is a binary data type which can have only two possible values- *true* or *false.*

Used typically in control structures for comparison.

All values which convert and will work like false:

```
undefined
null
0
-0
NaN
""
```

All other values, including all objects (and arrays) convert to, and work like, `true.`

`Truthy/Falsy`

Null

This type has only one possible value – *null*. It means that there is no value. If this type is used in boolean operations it is intrepret as falsy. See typeof null -> object.

Undefined

Any variable, which has not been assigned a value is *undefined.*

The equality operator == considers them to be equal. (Use the strict equality operator === to distinguish them.)

Number

This is a numeric type, which means that you can express decimals with it in contrast to plain integer values. The specification states it is so called double-precision 64-bit binary format IEEE754 value (https://en.wikipedia.org/wiki/Double-precision_floating-point_format).

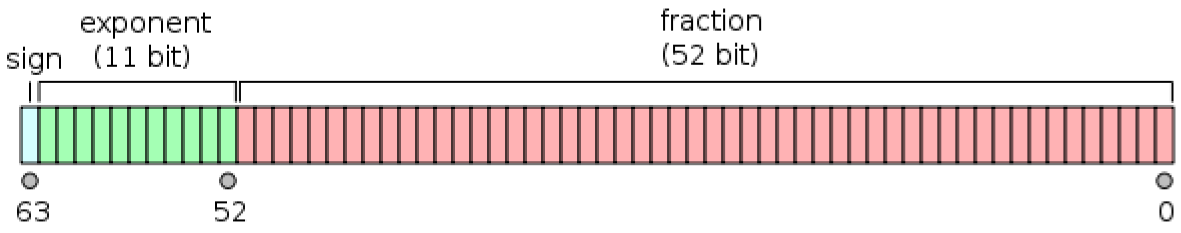


Figure 1 IEE754 bit structure (https://en.wikipedia.org/wiki/Double-precision_floating-point_format)

Maximum positive value with proper accuracy is 9007199254740992 (2^53).

Maximum possible number is 1.7976931348623157 x 10^308 (can be checked with *Number.MAX_VALUE*)

With large numbers you start to loose precision. See example below, where representing four decimals is possible only at 2^38 number. Any larger number with four decimals cannot be represented accurately. Instead the numbers will be rounded.

```
let pow52 = Math.pow(2, 52);
console.log(pow52); // 4503599627370496
let num1 = 4503599627370496.6789;
console.log(num1); //4503599627370497

let pow50 = Math.pow(2, 50);
console.log(pow50); // 1125899906842624
let num2 = 1125899906842624.6789;
console.log(num2); //1125899906842624.8

let pow48 = Math.pow(2, 38);
console.log(pow48); // 274877906944
let num3 = 274877906944.6789;
console.log(num3); //274877906944.6789
```

BigInt

This type can be used to store larger integers. BigInt is initialized with value *n* at the end of integer.

Consider the following code with regular numbers:

```
console.log('/// maximumSafeInt: ' + maximumSafeInt);
console.log('/// maximumSafeInt: ' + (maximumSafeInt + 1));
console.log('/// maximumSafeInt: ' + (maximumSafeInt + 2));
console.log('/// maximumSafeInt: ' + (maximumSafeInt + 3));
console.log('/// maximumSafeInt: ' + (maximumSafeInt + 4));
console.log('/// maximumSafeInt: ' + (maximumSafeInt + 5));
console.log('/// maximumSafeInt: ' + (maximumSafeInt + 6));

/// Output
/// maximumSafeInt: 9007199254740992
/// maximumSafeInt: 9007199254740992
/// maximumSafeInt: 9007199254740994
/// maximumSafeInt: 9007199254740996
/// maximumSafeInt: 9007199254740996
/// maximumSafeInt: 9007199254740996
/// maximumSafeInt: 9007199254740998
```

The output demonstrates that mathematical operations lose accuracy with normal numbers once the maximum limit has been reached.

With BigInt dealing with very large numbers is possible. See example below

```
let bigInt = 9007199254740992n;
console.log('/// bigInt: ' + bigInt);
console.log('/// bigInt: ' + (bigInt + 1n));
console.log('/// bigInt: ' + (bigInt + 2n));
console.log('/// bigInt: ' + (bigInt + 3n));
console.log('/// bigInt: ' + (bigInt + 4n));
console.log('/// bigInt: ' + (bigInt + 5n));
console.log('/// bigInt: ' + (bigInt + 6n));

/// bigInt: 9007199254740992
/// bigInt: 9007199254740993
/// bigInt: 9007199254740994
/// bigInt: 9007199254740995
/// bigInt: 9007199254740996
/// bigInt: 9007199254740997
/// bigInt: 9007199254740998
```

String

Primitive type is used to represent textual data. It is basically an array of elements, each an 16bit unsigned integer. The characters in a string, like in array, can be addressed via index where index 0 is the first character, index 1 second character etc.

JavaScript strings are immutable, they cannot be modified after creation. To modify, a new string must be created for example with concatenation operation +.

String is already a complete object with large number of methods available (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String).

Symbol

A recent ES6 addition to JavaScript, which provides a unique and immutable value. This has use cases for example if you need to create absolutely unique property keys for objects.

Below is a code example which tries to illustrate the uniqueness of symbols. Example of use case for symbols would be some reusable library functionality where there is a high chance of possible property key name conflicts.

```
let sym1 = Symbol("Demo");
let sym2 = Symbol("Demo");

console.log("Are symbols sym1 and sym2 equal when initialized with same constructor values?")
console.log(sym1 === sym2);  //false

let str1 = "Demo";
let str2 = "Demo";

console.log("Are strings str1 and str2 equal when initialized with same constructor values?")
console.log(str1 === str2);  // true
```

JavaScript Objects

Individual variables with primitive data types presented above allow you to deal with small amount of data. When application complexity increases also the demand for more complex data structures increase.

Objects allow this. An object is in essence a collection of variables, each with their own name and individual data type, which can be either one of the primitive types, object, array or function.

Below is an example, which highlights how a basic object is created with technique called object literal to serve as a collection of data:

```
// Example of a music record with normal primitives
let albumName = "The Razors Edge";
let songName = "Thunderstruck";
let recorded = "Jan 1990";
let length = "4:52";
let genre = ["Hard rock", "Heavy Metal"];
let songwriters = ["Angus Young", "Malcolm Young"];
let reviewStars = 4;

// example of same with an object
let songInfo = {
  albumName: "The Razors Edge",
  songName: "Thunderstruck",
  recorded: "Jan 1990",
  length: "4:52",
  genre: ["Hard rock", "Heavy Metal"],
  songwriters: ["Angus Young", "Malcolm Young"],
  reviewStars: 4
};

// read access for an object property with the dot notation.
console.log(songInfo.albumName);
```

As seen in the above example, an object is created by using curly brackets *{ }*. Properties are then listed in key-value manner, where there is *:* -character separating the key and value. Properties are separated from eachother with comma.

Read object property value

To read a value of an object property you simply use the dot notation together with property name. Example below

```
// read access for an object property with the dot notation.
console.log(songInfo.albumName);
```

Object property value is modified in the same way

```

songInfo.albumName = "Changed album name";
```

Exercise 1

Write code, which stores information of Belgium in an object. The object information should be stored into a variable named *belgium*.
Information to be stored is the following

Country	Belgium
Capital	Brussels
Languages	Dutch, French, German
Area	30689 km2
Population	11492641
GDP per capita	$50114
ISO3166 code	BE

Exercise 2

Enhance your Exercise 1 solution by creating objects similar to Belgium for Germany (do France and Sweden as well if you have time) (country data available at wikipedia – for example https://en.wikipedia.org/wiki/Germany).

Next write a function, which prints the name of the country and its population in a string. "Belgium, population 11492641". The function should accept single parameter, the object of the country information.

Call the function for all your countries.

Exercise 3

Create an array, which has similar country objects as in Exercise 2 and 1 as the elements of the array. The array should containt information about the same countries as exercise 2.

Write a function, which accepts an array of countries as an parameter and returns the one with maximum population.

```
function getMaxPopulationCountry(arrayOfCountries) {
  // implement your solution

  return countryWithMaxPopulation;
}
```

Exercise 4

Use the same array of countries as in exercise 3. Write a function which calculates and returns the sum of the populations of the countries in the array.

Exercise 5

Use the same array of countries as in exercise 3. Write a function which accepts the array of countries as a parameter, returns a new array of the country codes of the countries given as parameter.

Exercise 6

Use the same array of countries as in exercise 3. Write a function which accepts the array of countries as parameter. The function should print out a list of countries ordered by their GDP. Biggest should be first and smallest should be last. Print out only the name of the country and the GDP.

Do this exercise WITHOUT looking for help from the internet or AI about sorting algorithms.

First implement your own solution for this. Then you research about existing sorting algorithms, choose one and make an implementation of it.

Exercise 7

Use the same array of countries as in exercise 3. Modify the country information so that each country object has a new property called *largestCities* and the value of the property is an array of objects. The objects in the array should be formatted as follows:

```
{
    name: "Antwerp",
    population: 523248
}
```

The five biggest cities of each country should be recorded. Next create a function, which accepts the array of countries as parameter. The function should print out for each country

- the name of the country,
- the total population of the country,
- the names of the five biggest cities in the country and their population ,
- the total sum of population in the five biggest cities
- the percentage of the population in those cities out of the total population in the country.

Exercise 8

Take his dataset of earth meteorite landings from NASA https://data.nasa.gov/resource/y77d-th95.json. Create an application which loads the dataset as a JSON file. You can load JSON file to node application with *`const someObject = require('./somefile.json').`* Create a function, which prints the names of all meteorites.

Exercise 9

Your task is to create JS classes to represent students and teachers.

Student should have the following properties (There should be methods to set and get each property as well):

- Name
- Birthplace (country only, use two letter country codes from ISO 3166-1 alpha-2)
- Birthday
- Student Id (number, for example 34626)
- Class Id (eg. DIN21SP)
- Student email address

Teacher should have the following properties (There should be methods to set and get each property):

- Name
- Birthplace (country only, use two letter country codes from ISO 3166-1 alpha-2)
- Birthday
- Personnel Id (string, for example 345a-ffG-25KY)
- List of skill topics (for example frontend programming, mathematics, english language, communications etc.)
- Starting date of employment
- Staff email address

Use your classes to create five example student and two example teacher objects.

Exercise 10

Continue previous exercise and create a JS class to store information about courses.

Course should have the following information and necessary methods to set and get the info.

- Name
- Description
- Course Id (number)
- Teacher(s) (can be multiple)

- Registered students

Use the class to create three different courses and add students and teachers to the courses.

Exercise 11

Continue previous exercise and add the necessary methods to

- Get and print out all courses which a given teacher is teaching. (this method should be part of a relevant class)
- Get and print out all courses which a given student is participating (this method should be part of a relevant class)

Exercise 12

Use the dataset from exercise 8. Create a function which returns a random meteorite. Use the function result to print out the meteorite information (print out name, mass and year).

Exercise 13

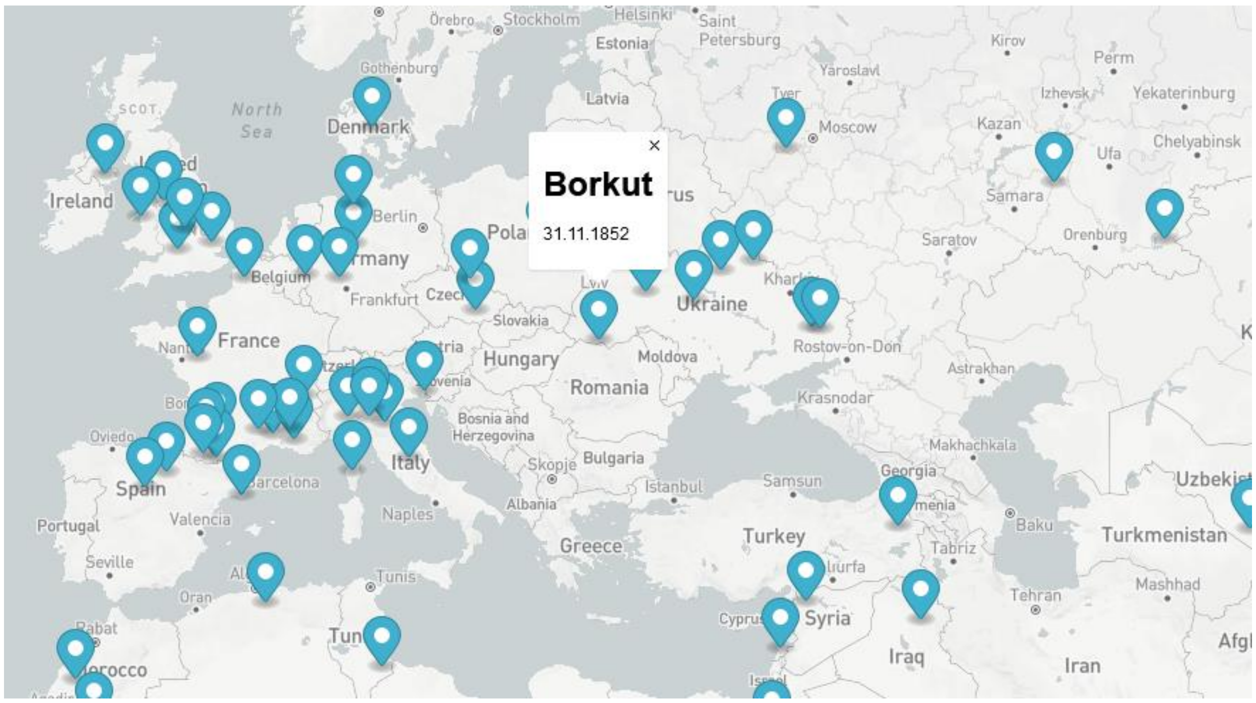
Improve your exercise 12 work by creating a new function, which accepts a parameter to indicate how many random meteorites are selected and then printed out. The new function should use the function you created for exercise 12.

Exercise 14

Use the dataset from exercise 8. Create a function, which returns an array containing information on the five biggest (by mass) meteorites.

Exercise 15

Use the NASA dataset from previous exercise together with Mapbox GL JS API to create a clickable map which looks like below. The map should show all meteorite impact locations on the map with markers and when marker is clicked it should show the name of the meteorite and the date of the impact.



You can use this Mapbox tutorial as a reference: https://docs.mapbox.com/help/tutorials/custom-markers-gl-js/

Creating a custom marker is not needed. You can do this with default markers. Below is an example of how to create a default clickable marker to an existing map.

```
new mapboxgl.Marker()
      .setLngLat([-77.032, 38.913])
      .setPopup(
          new mapboxgl.Popup().setHTML('<h1>Example popup</h1><div>This demonstra
tes clickable marker</div>')
        )
      .addTo(map);
```

Here is an example on how to load the Meteorite dataset and then access the data in the code.

```
fetch('https://data.nasa.gov/resource/y77d-th95.json')
  .then(response => response.json())
  .then(data => {
    // Print the loaded data to console
    console.log(data)
  });
```

Exercise 16

Create two objects, a and b, which both have properties with names

- width
- height
- depth
- weight

Assign random values to each property (how to get random value, see here https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random). All should be greater than zero, but less than 100.

Your task is to create the following functions:

- Function to calculate and print out the total volume of two objects given as parameters. Return single value.
- Function to calculate the sum of widths of the two objects given as parameters. Return single value.
- Function to calculate sum of heights of the two objects given as parameters. Return single value.
- Function to calculate the sum of weights of the two objects given as parameters. Return single value.
- A function to calculate all of the above values and return an object which contains all the calculation results.