Rest Parameters ...

- Many JavaScript built-in functions support an arbitrary number of arguments
- For instance:
  - Math.max(arg1, arg2, ..., argN) – returns the greatest of the arguments
  - Object.assign(dest, src1, ..., srcN) – copies properties from src1..N into dest
- We can define such functions using three dots ...
  - They literally mean “gather the remaining parameters into an array”

```
function sumAll(...args) { // args is the name for the array
    let sum = 0;

    for (let arg of args)
        sum += arg;
    return sum;
}

alert(sumAll(1)); // 1
alert(sumAll(1, 2)); // 3
alert(sumAll(1, 2, 3)); // 6
```

Rest Parameters …

- We can choose to get the first parameters as variables, and gather only the rest.
- Here the first two arguments go into variables and the rest go into titles array:

```
function showName(firstName, lastName, ...titles) {
    alert(firstName + ' ' + lastName); // Julius Caesar

    // the rest go into titles array
    // i.e. titles = ["Consul", "Imperator"]
    alert(titles[0]); // Consul
    alert(titles[1]); // Imperator
    alert(titles.length); // 2
}

showName("Julius", "Caesar", "Consul", "Imperator");
```

Spread Operator

- We’ve just seen how to get an array from the list of parameters
- But sometimes we need to do exactly the reverse
- For instance, the function **Math.max()** returns the greatest number from a list:

```
alert(Math.max(3, 5, 1)); // 5
```

- Now let’s say we have an array [3, 5, 1]. How do we call Math.max with it?
  - Passing it “as is” won’t work, because Math.max expects a list of numeric arguments
- The *Spread operator* ...arr “expands” an iterable object arr into the list of arguments

```
let arr = [3, 5, 1];
alert(Math.max(...arr)); // 5 (spread turns array into a list of arguments)
```

Spread Operator

- We can combine the spread operator with normal values:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert(Math.max(1, ...arr1, 2, ...arr2, 25)); // 25
```

- Also, the spread operator can be used to merge arrays:

```
let merged = [0, ...arr1, 2, ...arr2];
alert(merged); // 0,1,-2,3,4,2,8,3,-8,1 (0, then arr, then 2, then arr2)
```

- We can use the spread operator with any iterable, not only arrays
  - For instance, we can use it to turn a string into array of characters:

```
let str = "Hello";
alert([...str]); // H,e,l,l,o
```

Additional Array Methods

Method	Description
splice(pos, deleteCount, ...items)	at index pos delete deleteCount elements and insert items
slice(start, end)	creates a new array, copies elements from position start till end (not inclusive) into it
concat(...items)	returns a new array: copies all members of the current one and adds items to it
indexOf/lastIndexOf(item, pos)	look for item starting from position pos, return the index or -1 if not found
includes(value)	returns true if the array has value, otherwise false
find/filter(func)	filter elements through the function, return first/all values that make it return true
sort(func)	sorts the array in-place, then returns it
reverse()	reverses the array in-place, then returns it
split/join	convert a string to array and back
map(func)	creates a new array from results of calling func for every element

Removing Elements from Array

- The **arr.splice(str)** method is a swiss army knife for arrays
- It can do everything: add, remove and insert elements
- The syntax is:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

- It starts from the position index: removes deleteCount elements and then inserts elem1, ..., elemN at their place. Returns the array of removed elements.
- Typically it is used for deletion only:

```
let arr = ["I", "study", "JavaScript"];
arr.splice(1, 1); // from index 1 remove 1 element

alert(arr); // ["I", "JavaScript"]
```

Removing Elements from Array

- The method **arr.slice** is much simpler than similar-looking arr.splice
- The syntax is:

```
arr.slice(start, end)
```

- It returns a new array where it copies all items start index "start" to "end" (not including "end")
  - Both start and end can be negative, in that case position from array end is assumed
  - It works like str.slice, but makes subarrays instead of substrings

```
let arr = ["This", "is", "a", "test"];
alert(arr.slice(1, 3)); // is,a
alert(arr.slice(-2)); // a,test
```

Sorting an Array

- The method arr.sort sorts the array *in place*

```
let arr = [1, 2, 15];
arr.sort();

alert(arr);  // 1, 15, 2
```

- The order became 1, 15, 2. Incorrect. But why?
- **The items are sorted as strings by default**
- Literally, all elements are converted to strings and then compared
  - So, the lexicographic ordering is applied and indeed "2" > "15“
- This is because an array may contain numbers or strings or any type of elements
- To sort it, we need an *ordering function* that knows how to compare its elements
  - The default is a string order

Sorting an Array

- To use our own sorting order, we need to supply a function of two arguments as the argument of arr.sort()
- The function should work like this:

```
function compare(a, b) {
    if (a > b) return 1;
    if (a == b) return 0;
    if (a < b) return -1;
}
```

- For instance:

```
function compareNumeric(a, b) {
    if (a > b) return 1;
    if (a == b) return 0;
    if (a < b) return -1;
}
arr.sort(compareNumeric);

alert(arr);  // 1, 2, 15
```

Sorting an Array

- Actually, a comparison function is only required to return a positive number to say “greater” and a negative number to say “less”
- That allows to write shorter functions:

```
arr.sort(function (a, b) { return a - b; });

alert(arr);  // 1, 2, 15
```

- Or even shorter using arrow functions:

```
arr.sort((a, b) => a - b);

alert(arr);  // 1, 2, 15
```

Searching in Array

- The methods arr.indexOf(), arr.lastIndexOf() and arr.includes() have the same syntax and do essentially the same as their string counterparts, but operate on items instead of characters

```
let arr = [1, 0, false];

alert(arr.indexOf(0)); // 1
alert(arr.indexOf(false)); // 2
alert(arr.indexOf(null)); // -1

alert(arr.includes(1)); // true
```

  - Note that the methods use === comparison. So, if we look for false, it finds exactly false and not the zero

Searching in Array

- Say we have an array of objects. How do we find an object with a specific condition?
- Here the **arr.find()** method comes in handy
- The syntax is:

```
let result = arr.find(function (item, index, array) {
    // should return true if the item is what we are looking for
});
```

- For example, we have an array of users, each with the fields id and name
- Let’s find the one with id == 1:

```
let users = [
    { id: 1, name: "John" },
    { id: 2, name: "Pete" },
    { id: 3, name: "Mary" }
];

let user = users.find(item => item.id == 1);
alert(user.name); // John
```

Searching in Array

- The find method looks for a single (first) element that makes the function return true
- If there may be many, we can use **arr.filter(fn)**
- The syntax is roughly the same as find, but it returns an array of matching elements:

```
let users = [
    { id: 1, name: "John" },
    { id: 2, name: "Pete" },
    { id: 3, name: "Mary" }
];

// returns array of the first two users
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

Transforming an Array

- The arr.map method is a useful method for transforming an array
- The syntax is:

```
let result = arr.map(function (item, index, array) {
    // returns the new value instead of item
})
```

- It calls the function for each element of the array and returns the array of results
- For instance, here we transform each element into its length:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

Split and Join

- **str.split(delim)** splits the string into an array by the given delimiter delim
- In the example below, we split by a comma followed by space:

```
let names = 'Bilbo, Gandalf, Nazgul';
let arr = names.split(', ');

for (let name of arr) {
    alert(`A message to ${name}.`); // A message to Bilbo (and other names)
}
```

- The call **arr.join(str)** does the reverse to split
- It creates a string of arr items glued by str between them.

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];
let str = arr.join(';');

alert(str); // Bilbo;Gandalf;Nazgul
```

Exercise (14)

- Write the function sortByName(users) that gets an array of objects with property name and sorts it
- For instance:

```
let john = { name: "John", age: 25 };
let adam = { name: "Adam", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [john, adam, mary];

sortByName(arr);

// now: [adam, john, mary]
alert(arr[1].name); // John
```

Exercise (15)

- Let arr be an array
- Create a function unique(arr) that should return an array with unique items of arr
- For instance:

```
function unique(arr) {
    /* your code */
}

let values = ["John", "Harry", "Mary", "Harry", "Beth", "Harry", "Mary", "John"];

alert(unique(values)); // John, Harry, Mary, Beth
```

Exercise (16)

- You have an array of user objects, each one has name, surname and id
- Write the code to create another array from it, of objects with id and fullName, where fullName is generated from name and surname
- For instance:

```
let john = { name: "John", surname: "Smith", id: 1 };
let pete = { name: "Pete", surname: "Hunt", id: 2 };
let mary = { name: "Mary", surname: "Key", id: 3 };

let users = [john, pete, mary];

let usersMapped = /* ... your code ... */

alert(usersMapped[0].id) // 1
alert(usersMapped[0].fullName) // John Smith
```

Iterables

- **Iterables** are objects that can be used in for..of loops (you can "iterate" over them)
- Arrays, strings, and many other built-in Javascript objects are iterables
- Iterables are widely used by the core JavaScript, and many built-in operators and methods rely on them
- Iterables must implement the method named Symbol.iterator (a special built-in symbol just for that)
- The result of obj[Symbol.iterator] is an **iterator**, which handles the iteration process
- An iterator is an object that implements the method next(), which returns an object {done: Boolean, value: any}
  - **done:true** denotes the iteration end
  - **value** is the next value in the sequence

Iterable Example

- Let’s say we have an object, that is not an array, but looks suitable for for..of
- Like a range object that represents an interval of numbers:

```
let range = {
    from: 1,
    to: 5
};
// We want the for..of to work:
// for(let num of range) ... num=1,2,3,4,5
```

- To make the range iterable, we need to add to it a method named Symbol.iterator
  - When for..of starts, it calls that method (or errors if not found)
  - The method must return an *iterator* – an object with the method next()
  - When for..of wants the next value, it calls next() on that object
  - The result of next() must have the form {done: Boolean, value: any}, where done=true means that the iteration is finished, otherwise value must be the new value.

Iterable Example

```
// 1. call to for..of initially calls this
range[Symbol.iterator] = function () {
    // 2. ...it returns the iterator:
    return {
        current: this.from,
        last: this.to,

        // 3. next() is called on each iteration by the for..of loop
        next() {
            // 4. it should return the value as an object {done:.., value :...}
            if (this.current <= this.last) {
                return { done: false, value: this.current++ };
            } else {
                return { done: true };
            }
        }
    }
};

// now it works!
for (let num of range) {
    alert(num); // 1, then 2, 3, 4, 5
}
```

Calling an Iterator Explicitly

- Normally, internals of iterables are hidden from the external code
- There’s a for..of loop, that works, that’s all it needs to know.
- But to understand things better, let’s see how to create an iterator explicitly
- We’ll iterate over a string the same way as for..of, but with direct calls

```
let str = "hello";

// does the same as
// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();
while (true) {
    let result = iterator.next();
    if (result.done) break;
    alert(result.value); // outputs characters one by one
}
```

- That is rarely needed, but gives us more control over the process than for..of. For example, we can split the iteration process: iterate a bit, then stop, do something else, and then resume later.

Array.from

- The method **Array.from()** takes an iterable and makes a "real" Array from it
- Then we can call array methods on it, such as push(), pop(), etc.

```
// assuming that range is taken from the example above
let arr = Array.from(range);
arr.push(6);
alert(arr); // 1,2,3,4,5,6
```

- Here we use Array.from to turn a string into an array of characters:

```
let mystr = '𝒳⍰ ';

// splits mystr into array of characters, taking into account surrogate pairs
let chars = Array.from(mystr);

alert(chars[0]); // 𝒳
alert(chars[1]); // ⍰
alert(chars.length); // 2
```

  - Unlike str.split, it relies on the iterable nature of string and so, just like for..of, correctly works with surrogate pairs

Set

- Set is a collection of values, where each value may occur only once
- Its main methods are:
  - **new Set**(iterable) – creates the set, optionally from an array of values (any iterable will do)
  - **set.add**(value) – adds a value, returns the set itself
  - **set.delete**(value) – removes the value
    - returns true if value existed at the moment of the call, otherwise false
  - **set.has**(value) – returns true if the value exists in the set, otherwise false
  - **set.clear**() – removes everything from the set
  - **set.size** – the elements count

Set Example

- For example, we'd like to store all the users who have visited our site
  - But repeated visits should not lead to duplicates (a visitor must be counted only once)
- Set is just the right thing for that:

```
let set = new Set();
let john = { name: "John" };
let peter = { name: "Peter" };
let mary = { name: "Mary" };

// visits, some users come multiple times
set.add(john);
set.add(peter);
set.add(mary);
set.add(john);
set.add(mary);

// set keeps only unique values
alert(set.size); // 3

for (let user of set) {
    alert(user.name); // John (then Peter and Mary)
}
```

Exercise (17)

- Let arr be an array
- Create a function unique(arr) that should return an array with unique items of arr
- Use set to make the function more efficient
- For instance:

```
function unique(arr) {
    /* your code */
}

let values = ["John", "Harry", "Mary", "Harry", "Beth", "Harry", "Mary", "John"];

alert(unique(values)); // John, Harry, Mary, Beth
```

Exercise (18)

- Write a function subArrayZero(arr) that gets an array and returns whether it contains a contiguous subarray whose sum is equal to 0
  - Your function should go over the array elements only once

```
function subArrayZero(arr) {
    // your code
}

alert(subArrayZero([-5, 12, 4, -7, 2, 1, 8])); // true, 4 + (-7) + 2 + 1 = 0
alert(subArrayZero([3, -2, -6, 2, 1, -2])); // false
```

Map

- Map is a collection of keyed data items, just like an Object
- The main difference is that Map allows keys of any type
  - Objects can also be keys
- The main methods are:
  - **new Map()** – creates the map.
  - **map.set**(key, value) – stores the value by the key and returns the map
  - **map.get**(key) – returns the value by the key, undefined if key doesn't exist in map
  - **map.has**(key) – returns true if the key exists, false otherwise
  - **map.delete**(key) – removes the value by the key
  - **map.clear**() – clears the map
  - **map.size** – returns the current element count

Map Examples

```
let map = new Map();
map.set('1', 'str1');   // a string key
map.set(1, 'num1');     // a numeric key
map.set(true, 'bool1'); // a boolean key

// Map keeps the key type (unlike Object), so these two are different:
alert(map.get(1)); // 'num1'
alert(map.get('1')); // 'str1'

alert(map.size); // 3
```

```
// Using objects as keys
let user = { name: "John" };

// for every user, let's store his visits count
let visitsCountMap = new Map();

// john is the key for the map
visitsCountMap.set(user, 123);

alert(visitsCountMap.get(john)); // 123
```

Map From Object

- When a Map is created, we can pass an array (or another iterable) with key-value pairs, like this:

```
let map = new Map([
    ['1', 'str1'],
    [1, 'num1'],
    [true, 'bool1']
]);
```

- There is a built-in method Object.entries(obj) that returns an array of key/value pairs for an object exactly in that format
- So we can initialize a map from an object like this:

```
let map = new Map(Object.entries({
    name: "John",
    age: 30
}));
```

Iteration over Maps

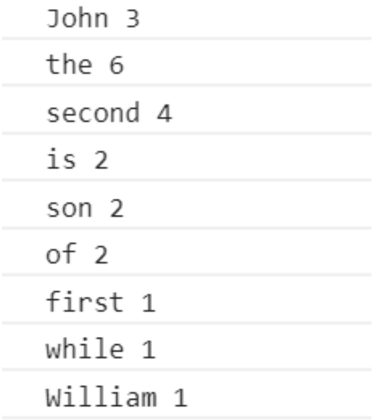
- For looping over a map, there are 3 methods:
  - **map.keys()** – returns an iterable for keys
  - **map.values()** – returns an iterable for values
  - **map.entries()** – returns an iterable for entries [key, value]
    - It is used by default in for..of

```
let recipeMap = new Map([
    ['cucumber', 10],
    ['tomatoes', 15],
    ['onion', 3]
]);

// iterate over keys (vegetables)
for (let vegetable of recipeMap.keys()) {
    alert(vegetable); // cucumber, tomatoes, onion
}

// iterate over values (amounts)
for (let amount of recipeMap.values()) {
    alert(amount); // 10, 15, 3
}

// iterate over [key, value] entries
for (let entry of recipeMap) { // the same as of
recipeMap.entries()
    alert(entry); // cucumber,10 (and so on)
}
```

Exercise (19)

- Create a function countWords(sentence) that gets a sentence and prints to the console the number of occurrences of each word in the sentence
- For instance:

```
function countWords(sentence) {
    // your code
}

let sentence = "John the second is the son of John the first,
while the second son of John the second is William the
second.";
countWords(sentence);
```



```
John 3
the 6
second 4
is 2
son 2
of 2
first 1
while 1
William 1
```

Exercise (20)

- Anagrams are words that have the same number of same letters, but in different order
- For instance:
  - nap - pan
  - ear - are - era
  - cheaters - hectares – teachers
- Write a function aclean(arr) that returns an array cleaned from anagrams
- For instance:

```
let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];

alert(aclean(arr)); // "nap,teachers,ear" or "PAN,cheaters,era"
```

  - From every anagram group should remain only one word, no matter which one

Destructuring Assignment

- Destructuring assignment allows for instantly “unpacking” arrays or objects into a bunch of variables, as sometimes they are more convenient
- Destructuring also works great with complex functions that have many parameters
- An example of how an array is destructured into variables:

```
// we have an array with first name and last name
let arr = ["Roi", "Yehoshua"];

// destructuring assignment
let [firstName, lastName] = arr;

// a shorter way for writing:
// let firstName = arr[0];
// let lastName = arr[1];

alert(firstName); // Roi
alert(lastName);  // Yehoshua
```

Destructuring Assignment

- Unwanted elements of the array can be thrown away via an extra comma:

```
// skipping the first and second elements, the third one is assigned to title,
// and the rest are also skipped
let [, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(title); // Consul
```

- We can use destrucutring assignment with any iterable, not only arrays:

```
let [a, b, c] = "abc"; // ["a", "b", "c"]
let [one, two, three] = new Set([1, 2, 3]);
```

- We can assign to anything at the left side, e.g., an object property:

```
let user = {};
[user.firstName, user.lastName] = "John Smith".split(' ');

alert(user.firstName); // John
```

Destructuring Assignment

- We can use destructuring to loop over keys-and-values of a map:

```
let countryCodes = new Map();
countryCodes.set("US", "United States");
countryCodes.set("FR", "France");
countryCodes.set("IL", "Israel");

for (let [key, value] of countryCodes.entries()) {
    alert(`${key}:${value}`); // US: United States, FR: France, IL: Israel
}
```

Object Destructuring

- The destructuring assignment also works with objects
- The basic syntax is:

```
let {var1, var2} = {var1:…, var2…}
```

- For example:

```
let options = {
    title: "Menu",
    width: 100,
    height: 200
};
let { title, width, height } = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

  - The properties options.title, options.width and options.height are assigned to the corresponding variables. The order of the variables on the left side does not matter.

Object Destructuring

- If we want to assign a property to a variable with another name, e.g., options.width to go into the variable named w, then we can set it using a colon:

```
let options = {
    title: "Menu",
    width: 100,
    height: 200
};

// { sourceProperty: targetVariable }
let { width: w, height: h, title } = options;

// width -> w
// height -> h
// title -> title

alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
```

Object Destructuring

- For potentially missing properties we can set default values using "=", like this:

```
let options = {
    title: "Menu"
};

let { width = 100, height = 200, title } = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

- Just like with arrays or function parameters, default values can be any expressions or even function calls. They will be evaluated if the value is not provided.

Object Destructuring

- We can use existing variables on the left side of the destructuring assignment
- But there’s a catch:

```
let title, width, height;

// error in this line
{ title, width, height } = { title: "Menu", width: 200, height: 100 };
```

- The problem is that JavaScript treats {...} as a code block
- To show JavaScript that it’s not a code block, we need to wrap the whole assignment in brackets (...):

```
let title, width, height;

// okay now
({ title, width, height } = { title: "Menu", width: 200, height: 100 });

alert(title); // Menu
```

Smart Function Parameters

- There are times when a function has many parameters, most of which are optional
- Imagine a function that creates a menu. It may have a width, a height, a title, items list and so on.
- Here’s a bad way to write such function:

```
function showMenu(title = "Untitled", width = 200, height = 100, items = []) {
    // ...
}
```

- The problem is how to remember the order of arguments, and also how to call such a function when most parameters are ok by default. Like this?

```
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"]);
```

- That’s ugly, and becomes unreadable when we deal with more parameters

Smart Function Parameters

- Destructuring comes to the rescue!
- We can pass parameters as an object, and the function immediately destructurizes them into variables:

```
// we pass object to function
let options = {
    title: "My menu",
    items: ["Item1", "Item2"]
};

// ...and it immediately expands it to variables
function showMenu({ title = "Untitled", width = 200, height = 100, items = [] }) {
    // title, items – taken from options, width, height – defaults used
    alert(`${title} ${width} ${height}`); // My Menu 200 100
    alert(items); // Item1, Item2
}

showMenu(options);
```

Smart Function Parameters

- We can also use more complex destructuring with nested objects and colon mappings:

```
let options = {
    title: "My menu",
    items: ["Item1", "Item2"]
};

function showMenu({
    title = "Untitled",
    width: w = 100,  // width goes to w
    height: h = 200, // height goes to h
    items: [item1, item2] // items first element goes to item1, second to item2
}) {
    alert(`${title} ${w} ${h}`); // My Menu 100 200
    alert(item1); // Item1
    alert(item2); // Item2
}

showMenu(options);
```

Exercise (21)

- We have an object:

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

- Write the destructuring assignment that reads:
  - name property into the variable name
  - years property into the variable age
  - isAdmin property into the variable isAdmin (false if absent)
- The values after the assignment should be:

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

// your code to the left side:
// ... = user;

alert(name); // John
alert(age); // 30
alert(isAdmin); // false
```

Date and Time

- Let’s meet a new built-in object: **Date**
- It stores the date, time and provides methods for date/time management
- For instance, we can use it to measure time, or just to print out the current date
- To create a new Date object call new Date() with one of the following arguments:
  - **new Date()** - creates a Date object for the current date and time
  - **new Date(milliseconds)** - creates a Date object with the time equal to number of milliseconds passed after the Jan 1st of 1970 UTC+0 (this is called a timestamp)
  - **new Date(datestring)** - reads the date from a string
  - **new Date(year, month, date, hours, minutes, seconds, ms)** - creates the date with the given components in the local time zone
    - The year must have 4 digits: 2013 is okay, 98 is not
    - The month count starts with 0 (Jan), up to 11 (Dec)
    - The date parameter is actually the day of month, if absent then 1 is assumed
    - If hours/minutes/seconds/ms is absent, they are assumed to be equal 0

Date Creation Example

```
let now = new Date();
alert(now); // shows current date/time

// 0 means 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert(Jan01_1970);

let date = new Date("2018-05-25");
alert(date); // Fri May 25 2018 ...

let date2 = new Date(2011, 0, 1, 2, 3, 4, 567);
alert(date2); // 1.01.2011, 02:03:04.567

new Date(2011, 0, 1); // 1 Jan 2011, 00:00:00
```

Access Date Components

- There are many methods to access the year, month and so on from the Date object:
  - **getFullYear()** - get the year (4 digits)
  - **getMonth(**) - get the month, **from 0 to 11**
  - **getDate()** - get the day of month, from 1 to 31 (the method name may look strange)
  - **getHours(), getMinutes(), getSeconds(), getMilliseconds()** - get the corresponding time components
  - **getDay()** - get the day of week, from 0 (Sunday) to 6 (Saturday)
- All the methods above return the components relative to the local time zone
- There are also their UTC-counterparts, that return day, month, year and so on for the time zone UTC+0: getUTCFullYear(), getUTCMonth(), getUTCDay()

Access Date Components

```javascript
let currDay = now.getDate();
let currMonth = now.getMonth() + 1;
let currYear = now.getFullYear();
alert(`${currDay}/${currMonth}/${currYear}`); // 25/5/2018

// the hour in your current time zone
alert(now.getHours());

// the hour in UTC+0 time zone (London time without daylight savings)
alert(now.getUTCHours());
```

Measuring Time Difference

- Dates can be subtracted, the result is their difference in ms
- However, if we only want to measure the difference, we don’t need the Date object
- There’s a special method **Date.now()** that returns the current timestamp
  - It is semantically equivalent to new Date().getTime(), but it doesn’t create an intermediate Date object, so it’s faster
- For instance:

```
let start = Date.now(); // milliseconds count from 1 Jan 1970

// do the job
for (let i = 0; i < 100000; i++) {
    let doSomething = i * i * i;
}

let end = Date.now(); // done
alert(`The loop took ${end - start} ms`); // subtract numbers, not dates
```

Exercise (22)

- Create a function getSecondsToTomorrow() that returns the number of seconds till tomorrow
- For instance, if now is 23:00, then:

```
getSecondsToTomorrow() == 3600
```

- Note that the function should work at any day

Scheduling: setTimeout and setInterval

- We may decide to execute a function not right now, but at a certain time later
- That’s called “scheduling a call”
- There are two methods for it:
  - **setTimeout()** allows to run a function once after the interval of time
  - **setInterval()** allows to run a function regularly with the interval between the runs
- These methods are supported in all browsers and Node.JS

setTimeout

- The syntax:

```
let timerId = setTimeout(func|code, delay[, arg1, arg2...])
```

  - **func|code** – a function or a string of code to execute. Usually, that’s a function.
  - **delay** - the delay before run, in milliseconds (1000 ms = 1 second)
  - **arg1, arg2…** - arguments for the function

- For instance, this code calls sayHi() after one second:

```
function sayHi() {
    alert('Hello');
}
setTimeout(sayHi, 1000);
```

  - You can also use an arrow function:

```
setTimeout(() => alert('Hello'), 1000);
```

setTimeout

- Example for passing arguments to the schedules function:

```
function sayHi(phrase, who) {
    alert(phrase + ', ' + who);
}

setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

- Novice developers sometimes make a mistake by adding () after the function:

```
// wrong!
setTimeout(sayHi(), 1000);
```

  - That doesn’t work, because setTimeout expects a reference to function, and here sayHi() runs the function, and the *result of its execution* is passed to setTimeout
  - In our case the result of sayHi() is undefined (the function returns nothing), so nothing is scheduled

Canceling with clearTimeout

- A call to setTimeout returns a “timer identifier” **timerId**, that we can use to cancel the execution
- The syntax to cancel:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

- In the code below, we schedule the function and then cancel it
- As a result, nothing happens:

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier

clearTimeout(timerId);
```

setInterval

- The **setInterval** method has the same syntax as setTimeout:

```
let timerId = setInterval(func|code, delay[, arg1, arg2...])
```

  - All arguments have the same meaning
  - But unlike setTimeout it runs the function not only once, but regularly after the given interval of time
- To stop further calls, you can call **clearInterval**(timerId)
- The following example shows a message every 2 seconds, and stops after 5 seconds:

```
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);

// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

  - In Chrome, Opera and Safari the internal timer becomes “frozen” while showing alert/prompt
  - So if you run the code above and don’t dismiss the alert window after some time, then the next alert will be shown after 2 more seconds (timer did not tick during the alert)

setTimeout(…,0)

- There’s a special use case: setTimeout(func, 0)
- This schedules the execution of func as soon as possible
- But scheduler will invoke it only after the current code is complete
- So the function is scheduled to run “right after” the current, i.e., *asynchronously*.
- For instance, this outputs “Hello”, then immediately “World”:

```
setTimeout(() => alert("World"), 0);

alert("Hello");
```

  - The first line “puts the call into calendar after 0ms”. But the scheduler will only “check the calendar” after the current code is complete, so "Hello" is first, and "World" – after it.

Splitting CPU-Hungry Tasks

- There’s a trick to split CPU-hungry tasks using setTimeout
- Let’s take a simpler example for consideration
- We have a function to count from 1 to 2000000000:

```
let i = 0;
let start = Date.now();

function count() {
    // do a heavy job
    for (let j = 0; j < 2e9; j++) {
        i++;
    }
    alert("Done in " + (Date.now() - start) + 'ms');
}
count();
```

- If you run it, the CPU will hang - the whole JavaScript actually is paused, no other actions work until it finishes

Splitting CPU-Hungry Tasks

- Let’s split the job using the nested setTimeout:

```
let i = 0;
let start = Date.now();

function count() {
    // do a piece of the heavy job
    do {
        i++;
    } while (i % 1e6 != 0);

    if (i == 1e9) {
        alert("Done in " + (Date.now() - start) + 'ms');
    } else {
        setTimeout(count, 0); // schedule the new call
    }
}
count();
```

- Now the browser UI is fully functional during the “counting” process
  - Pauses between count executions provide just enough “breath” for the JavaScript engine to do something else, to react to other user actions

Exercise (23)

- Write a function printNumbers(from, to) that outputs a number every second, starting from **from** and ending with **to**
- Make two variants of the solution:
  - Using setInterval()
  - Using setTimeout()