3 - Data Types

Data Types

Let’s look at the key distinctions between primitives and objects.

A primitive is a value of a primitive type.

There are 7 primitive types: string, number, bigint, boolean, symbol, null and undefined.

An object is capable of storing multiple values as properties.

Can be created with {}, for instance: {name: "John", age: 30}. There are other kinds of objects in JavaScript: functions, for example, are objects.

Numbers

We can define numbers the classic way or using separator while we define a big number:

```
let billion = 1_000_000_000;
```

this would work the same way as writing 1000000000.

To make it shorter, we could also write:

```
let billion = 1e9
```

toString(base)

The method num.toString(base) returns a string representation of num in the numeral system with the given base.

```
let num = 255;

alert( num.toString(16) );  // ff
alert( num.toString(2) );   // 11111111
```

for mathematical operations with numbers, please visit: https://javascript.info/number

Strings

Special characters:

Character	Description
`\n`	New line
`\r`	In Windows text files a combination of two characters `\r\n` represents a new break, while on non-Windows OS it's just `\n`. That's for historical reasons, most Windows software also understands `\n`.
`\'`, `\"`	Quotes
`\\`	Backslash
`\t`	Tab
`\b`, `\f`, `\v`	Backspace, Form Feed, Vertical Tab – kept for compatibility, not used nowadays.
`\xXX`	Unicode character with the given hexadecimal Unicode `XX`, e.g. `'\x7A'` is the same as `'z'`.
`\uXXXX`	A Unicode symbol with the hex code `XXXX` in UTF-16 encoding, for instance `\u00A9` – is a Unicode for the copyright symbol `©`. It must be exactly 4 hex digits.
`\u{X…XXXXXX}` (1 to 6 hex characters)	A Unicode symbol with the given UTF-32 encoding. Some rare characters are encoded with two Unicode symbols, taking 4 bytes. This way we can insert long codes.

String length

The length property of a string returns its length:

```
alert( `My\n`.length ); // 3
```

We can access single characters using [ ];

Strings are immutable objects in Javascript

```
let str = 'Hi';

str[0] = 'h'; // error
alert( str[0] ); // doesn't work
```

Methods `toLowerCase()` and `toUpperCase()` change the case.

Looking for a substring can be done with `str.indexOf(substr, pos)`.

Array

Array

Declarations and methods can be found at: https://javascript.info/array
Arrays in JavaScript can work both as a queue and as a stack. They allow you to add/remove elements, both to/from the beginning or the end.

Loop in an array

```
let fruits = ["Apple", "Orange", "Plum"];

// iterates over array elements
for (let fruit of fruits) {
  alert( fruit );
}
```

Clearing an array

Array length property is writable, so if i set an array length to a number lower that it's size, it gets truncated:

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // truncate to 2 elements
alert( arr ); // [1, 2]

arr.length = 5; // return length back
alert( arr[3] ); // undefined: the values do not return
```

The simplest way to clear the array is using `array.length = 0`

toString : returns a comma separated list of elements

Summary

Array is a special kind of object, suited to storing and managing ordered data items.

The declaration:

```
// square brackets (usual)
let arr = [item1, item2...];

// new Array (exceptionally rare)
let arr = new Array(item1, item2...);
```

The call to new Array(number) creates an array with the given length, but without elements.

- The length property is the array length or, to be precise, its last numeric index plus one. It is auto-adjusted by array methods.
- If we shorten length manually, the array is truncated.

Getting the elements:

- we can get element by its index, like arr[0]
- also we can use at(i) method that allows negative indexes. For negative values of i, it steps back from the end of the array. If i >= 0, it works same as arr[i].
  We can use an array as a deque with the following operations:

`push(...items)` adds items to the end.
`pop()` removes the element from the end and returns it.
`shift()` removes the element from the beginning and returns it.
`unshift(...items)` adds items to the beginning.

To loop over the elements of the array:

- `for (let i=0; i<arr.length; i++)` – works fastest, old-browser-compatible.
- `for (let item of arr)` – the modern syntax for items only,
- `for (let i in arr)` – never use.

To compare arrays, don't use the == operator (as well as >, < and others), as they have no special treatment for arrays. They handle them as any objects, and it's not what we usually want.

Instead you can use for..of loop to compare arrays item-by-item.

Array methods

`push()` adds items to the end,
`pop()` extracts an item from the end,
`shift()` extracts an item from the beginning,
`unshift()` adds items to the beginning.
`delete arr[1]` sets arr[1] to undefined (does not pop the element)

`splice ()`: modifies arr starting from the index start, removes deleteCount elements and then inserts elem1,...,elemN at their place. Returns the array of removed elements.

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

Examples:

```
let arr = ["I", "study", "JavaScript", "right", "now"];
```

```
// remove 3 first elements and replace them with another
arr.splice(0, 3, "Let's", "dance");

alert( arr ) // now ["Let's", "dance", "right", "now"]
```

splice returns the array of removed elements:

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// remove 2 first elements
let removed = arr.splice(0, 2);

alert( removed ); // "I", "study" <-- array of removed elements
```

```
\\insert elements without any removals (deleteCount to 0)
let arr = ["I", "study", "JavaScript"];

// from index 2
// delete 0
// then insert "complex" and "language"
arr.splice(2, 0, "complex", "language");

alert( arr ); // "I", "study", "complex", "language", "JavaScript"
```

negative indexes are allowed to specify position from the end of the array:

```
let arr = [1, 2, 5];

// from index -1 (one step from the end)
// delete 0 elements,
// then insert 3 and 4
arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5
```

`slice()`: eturns a new array copying to it all items from index start to end (not including end). Both start and end can be negative, in that case position from array end is assumed.

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

```
let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s (copy from 1 to 3)

alert( arr.slice(-2) ); // s,t (copy from -2 till the end)
```

`concat()`: create a new array including values from other arrays and additional items, syntax is:

```
arr.concat(arg1, arg2...)
```

any number of parameters works:

```
let arr = [1, 2];

// create an array from: arr and [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4

// create an array from: arr and [3,4] and [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// create an array from: arr and [3,4], then add values 5 and 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

`forEach()`: run a function for every element of the array

```
arr.forEach(function(item, index, array) {
  // ... do something with item
});
```

```
// for each element call alert
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  alert(`${item} is at index ${index} in ${array}`);
});
```

`arr.indexOf(item, from)`: looks for item starting from index from, and returns the index where it was found, otherwise -1.
`arr.includes(item, from)` – looks for `item` starting from index `from`, returns true if found.

`arr.find(fn)`: useful when we have an array of objects, how to find an object with the specific condition?

```
let result = arr.find(function(item, index, array) {
  // if true is returned, item is returned and iteration is stopped
  // for falsy scenario returns undefined
});
```

The function is called for every element of the array. If it returns true, the search is stopped, the item is returned. If nothing found, undefined is returned.

```
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
```

`arr.findIndex()` and `arr.findLastIndex()` return the indexes of the objects if found inside the array, the second scans from right to left.

`find()` method looks for a single (first) element that makes the function return true, we can use `filter()` if there are many:

```
let results = arr.filter(function(item, index, array) {
  // if true item is pushed to results and the iteration continues
  // returns empty array if nothing found
});

//Example:

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
```

`map()`: one of the most useful and used fucntion. Calls the function for each element of the array and returns the array of results.

Syntax is:

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

```
//Transform each element into its length
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
```

```
alert(lengths); // 5,7,6
```

`sort`: sorts the arrays in place, changing its element order. arr itself is modified so the returned value is ignored.

```
let arr = [ 1, 2, 15 ];

// the method reorders the content of arr
arr.sort();

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

The numerical order is not correct because sort converts every element to string by default.
To use our own order we need to supply a function as the argument of `arr.sort()`

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

let arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

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

A comparison function may return any number
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:

```
let arr = [ 1, 2, 15 ];

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

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

\\ or we can use array functions

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

`reverse()`: reverses the order of elements in arr

`split()`: splits the string into an array by the given delimiter `delimit`

```
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 split method has an optional second numeric argument – a limit on the array length. If it is provided, then the extra elements are ignored. In practice it is rarely used though:

```
let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(', ', 2);

alert(arr); // Bilbo, Gandalf
```

Splitting into letters:

```
let str = "test";

alert( str.split('') ); // t,e,s,t
```

`glue()`: it is the opposite of split, creates a string of `arr` items joined by a separator between them:

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];

let str = arr.join(';'); // glue the array into a string using ;

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

`reduce()`: used to calculate a single value based on the array

Maps and Sets

Map is a collection of keyed data items, just like Objects, but they allows keys of any type.

- `new Map()` – creates the map.
- `map.set(key, value)` – stores the value by the key.
- `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()` – removes everything from the map.
- `map.size` – returns the current element count.

```
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

// remember the regular Object? it would convert keys to string
// Map keeps the type, so these two are different:
alert( map.get(1)   ); // 'num1'
alert( map.get('1') ); // 'str1'

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

Unlike objects, keys are not converted to strings. Any type of key is possible.
Maps shouldn't be used with [] operator, even if it works. This way maps are treated like objects, we should use get() and set() methods.

Maps can use objects as keys:

```
Maps can use objects as keys:

let john = { name: "John" };

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

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

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

This is one of the most useful Map feature

In Objects we cannot use other Objects as key.

Iterate 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's used by default in for..of.

```
let recipeMap = new Map([
  ['cucumber', 500],
  ['tomatoes', 350],
  ['onion',    50]
]);

// 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); // 500, 350, 50
}

// iterate over [key, value] entries
for (let entry of recipeMap) { // the same as of recipeMap.entries()
  alert(entry); // cucumber,500 (and so on)
}
```

Unlike the Objects, maps entries are iterated in the way they were entered.

Object.entries - Create a map from Object

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

let map = new Map(Object.entries(obj));
```

```
alert( map.get('name') ); // John
```

Here, Object.entries returns the array of key/value pairs: [["name","John"], ["age", 30]]. That's what Map needs.

The `Object.fromEntries()` does the inverse, creating an Object from an array of [keys,values].

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

// now prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

If we have data stored in a Map but we need to pass it to a 3rd-party code that expects a plain object:

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // or
let obj = Object.fromEntries(map);
// make a plain object (*)

// done!
// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2
```

A call to map.entries() returns an iterable
of key/value pairs, exactly in the format for Object.fromEntries.

A Set is a special data type collection where each value may occur only once:

```
new Set(iterable) – creates the set, and if an iterable object is provided
(usually an array), copies values from it into the set.
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 – is the elements count.
```

Repeated calls os set.add(value) with the same value don't do anything. That's the reason why each value appears in a Set only once.

```
let set = new Set();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// visits, some users come multiple times
set.add(john);
set.add(pete);
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 Pete and Mary)
}
```

Set is much better organized internally for uniqueness checks unlike arrays.

Iteration over Set

```
let set = new Set(["oranges", "apples", "bananas"]);

for (let value of set) alert(value);

// the same with forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

Note the funny thing. The callback function passed in forEach has 3 arguments: a value, then the same value valueAgain, and then the target object. Indeed, the same value appears in the arguments twice.

That’s for compatibility with Map where the callback passed forEach has three arguments. Looks a bit strange, for sure. But may help to replace Map with Set in certain cases with ease, and vice versa. The

same methods Maps has for iterators are supported:

- set.keys() – returns an iterable object for values,
- set.values() – same as set.keys(), for compatibility with Map,
- set.entries() – returns an iterable object for entries [value, value], exists for compatibility with Map.

Summary

Map – is a collection of keyed values.

The differences from a regular Object:

- Any keys, objects can be keys.
- Additional convenient methods, the size property.

Set – is a collection of unique values.

Iteration over Map and Set is always in the insertion order, so we can’t say that these collections are unordered, but we can’t reorder elements or directly get an element by its number.

WeakMaps and WeakSets

If we use an object as the key in a regular Map, then while the Map exists, that object exists as well. It occupies memory and may not be garbage collected. Differently from normal Maps, WeakMaps does not prevent garbage-collection of key objects.

The first difference between Map and WeakMap is that keys must be objects, not primitive values:

```
let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // works fine (object key)

// can't use a string as the key
weakMap.set("test", "Whoops"); // Error, because "test" is not an object
```

Now, if we use an object as the key in it, and there are no other references to that object – it will be removed from memory (and from the map) automatically.

```
let john = { name: "John" };

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // overwrite the reference

// john is removed from memory!
```

WeakMap does not support iteration and methods keys(), values(), entries(), so there's no way to get all keys or values from it.

WeakMap has only the following methods:

- weakMap.get(key)
- weakMap.set(key, value)
- weakMap.delete(key)
- weakMap.has(key)

Why such a limitation? That's for technical reasons. If an object has lost all other references (like john in the code above), then it is to be garbage-collected automatically. But technically it's not exactly specified when the cleanup happens.

The JavaScript engine decides that. It may choose to perform the memory cleanup immediately or to wait and do the cleaning later when more deletions happen. So, technically, the current element count of a WeakMap is not known. The engine may have cleaned it up or not, or did it partially. For that reason, methods that access all keys/values are not supported.

What are WeakMaps used for?

If we're working with an object that "belongs" to another code, maybe even a third-party library, and would like to store some data associated with it, that should only exist while the object is alive – then WeakMap is exactly what's needed.

We put the data to a WeakMap, using the object as the key, and when the object is garbage collected, that data will automatically disappear as well.

```
weakMap.set(john, "secret documents");
// if john dies, secret documents will be destroyed automatically
```

For instance, we have code that keeps a visit count for users. The information is stored in a map: a user object is the key and the visit count is the value. When a user leaves (its object gets garbage collected), we don't want to store their visit count anymore.

Here's an example of a counting function with Map:

```
// 📁 visitsCount.js
let visitsCountMap = new Map(); // map: user => visits count

// increase the visits count
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}

// somewhere else in the code:

// 📁 main.js
let john = { name: "John" };

countUser(john); // count his visits

// later john leaves us
john = null;
```

Now, john object should be garbage collected, but remains in memory, as it's a key in visitsCountMap.

We need to clean visitsCountMap when we remove users, otherwise it will grow in memory indefinitely. Such cleaning can become a tedious task in complex architectures.

We can avoid it by switching to WeakMap instead:

```
// 📁 visitsCount.js
let visitsCountMap = new WeakMap(); // weakmap: user => visits count

// increase the visits count
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

Now we don't have to clean visitsCountMap. After john object becomes unreachable, by all means except as a key of WeakMap, it gets removed from memory, along with the information by that key from WeakMap.

Another use-case scenario is caching, doing it with Maps is not optimal:

```
// 📁 cache.js
let cache = new Map();

// calculate and remember the result
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* calculations of the result for */ obj;

    cache.set(obj, result);
  }

  return cache.get(obj);
}

// Now we use process() in another file:

// 📁 main.js
let obj = {/* let's say we have an object */};

let result1 = process(obj); // calculated

// ...later, from another place of the code...
let result2 = process(obj); // remembered result taken from cache
```

```
// ...later, when the object is not needed any more:
obj = null;

alert(cache.size); // 1 (Ouch! The object is still in cache, taking memory!)
```

For multiple calls of process(obj) with the same object, it only calculates the result the first time, and then just takes it from cache. The downside is that we need to clean cache when the object is not needed any more.

If we replace Map with WeakMap, then this problem disappears. The cached result will be removed from memory automatically after the object gets garbage collected.

```
// 📁 cache.js
let cache = new WeakMap();

// calculate and remember the result
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* calculate the result for */ obj;

    cache.set(obj, result);
  }

  return cache.get(obj);
}

// 📁 main.js
let obj = {/* some object */};

let result1 = process(obj);
let result2 = process(obj);

// ...later, when the object is not needed any more:
obj = null;

// Can't get cache.size, as it's a WeakMap,
// but it's 0 or soon be 0
// When obj gets garbage collected, cached data will be removed as well
```

The most notable limitation of WeakMap and WeakSet is the absence of iterations, and the inability to get all current content. That may appear inconvenient, but does not prevent WeakMap/WeakSet from doing their main job – **be an "additional" storage of data for objects which are stored/managed at another place.**

Objects.keys,values,entries

We can use the following methods with objects:

- Object.keys(obj) – returns an array of keys.
- Object.values(obj) – returns an array of values.
- Object.entries(obj) – returns an array of [key, value] pairs.

This methods are different from maps

	Map	Object
Call syntax	`map.keys()`	`Object.keys(obj)`, but not `obj.keys()`
Returns	iterable	"real" Array

example of using Object.values to loop over property values:

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

// loop over values
for (let value of Object.values(user)) {
  alert(value); // John, then 30
}
```

As Objects lack many methods that exist for arrays (map, filter), we can transform the objects using Objects.entries and Object.fromEntries:

- Use Object.entries(obj) to get an array of key/value pairs.
- Use array methods on that array like map().
- Uso Object.fromEntries(array) to turn in back into an object

Here's an example of how we could double the prices of items inside an object:

```
let prices = {
  banana: 1,
  orange: 2,
  meat: 4,
};

let doublePrices = Object.fromEntries(
  // convert prices to array, map each key/value pair into another pair
  // and then fromEntries gives back the object
```

```
  Object.entries(prices).map(entry => [entry[0], entry[1] * 2])
);

alert(doublePrices.meat); // 8
```

Destructuring assignment

It is a special syntax that allows us to unpack arrays or obkects into a bunch of variables, as something that's more convenient. Works also with complex functions with a lot of parameters. A basic example could be:

```
// we have an array with the name and surname
let arr = ["John", "Smith"]

// destructuring assignment
// sets firstName = arr[0]
// and surname = arr[1]
let [firstName, surname] = arr;

alert(firstName); // John
alert(surname);  // Smith
----------------------------------------------------------------
// we can combine it with split method:
let [firstName, surname] = "John Smith".split(' ');
alert(firstName); // John
alert(surname);  // Smith

// The original array itself is not modified.
----------------------------------------------------------------
// We can throw away unwanted elements  with commas:

// second element is not needed
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman
Republic"];

alert( title ); // Consul
----------------------------------------------------------------
// It works with any iterable on the right side
let [a, b, c] = "abc"; // ["a", "b", "c"]
let [one, two, three] = new Set([1, 2, 3]);
----------------------------------------------------------------
// Assign anything on the left side
let user = {};
[user.name, user.surname] = "John Smith".split(' ');

alert(user.name); // John
alert(user.surname); // Smith
----------------------------------------------------------------
```

```
// We can loop with .entries()

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

// loop over keys-and-values
for (let [key, value] of Object.entries(user)) {
  alert(`${key}:${value}`); // name:John, then age:30
}
-----------------------------------------------------------------
// with a Map is a bit easier:

let user = new Map();
user.set("name", "John");
user.set("age", "30");

// Map iterates as [key, value] pairs, very convenient for destructuring
for (let [key, value] of user) {
  alert(`${key}:${value}`); // name:John, then age:30
}
-----------------------------------------------------------------
// Swap variables trick

let guest = "Jane";
let admin = "Pete";

// Let's swap the values: make guest=Pete, admin=Jane
[guest, admin] = [admin, guest];

alert(`${guest} ${admin}`); // Pete Jane (successfully swapped!)
-----------------------------------------------------------------
// Taking the rest
let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman
Republic"];

// rest is array of items, starting from the 3rd one
alert(rest[0]); // Consul
alert(rest[1]); // of the Roman Republic
alert(rest.length); // 2

// The value of rest is the array of the remaining array elements. Any name
```

```
can be used instead of rest, but in needs to have three dots and be in the last position
----------------------------------------------------------------
// Absent value are considered undefined
let [firstName, surname] = [];
alert(firstName); // undefined
alert(surname); // undefined
// defined default values
let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name);    // Julius (from array)
alert(surname); // Anonymous (default used)
```

Object destructuring

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

let {title, width, height} = options;

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

// In this case, order does not matter, this would work too:
// changed the order in let {...}
let {height, width, title} = { title: "Menu", height: 200, width: 100 }

---------------------------------------------------------------------
// If we want to redefine the name of the variable in which values are goint
(e.g width->w):
// { sourceProperty: targetVariable }
let {width: w, height: h, title} = options;
// {what: goes where}
---------------------------------------------------------------------
//We can extract only what we need from complex objects:
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

// only extract title as a variable
let { title } = options;

alert(title); // Menu
---------------------------------------------------------------------
//Taking the rest works also in this case:

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

```
};

// title = property named title
// rest = object with the rest of properties
let {title, ...rest} = options;

// now title="Menu", rest={height: 200, width: 100}
alert(rest.height);  // 200
alert(rest.width);   // 100
----------------------------------------------------------------
//Nested assignments:
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

// destructuring assignment split in multiple lines for clarity
let {
  size: { // put size here
    width,
    height
  },
  items: [item1, item2], // assign items here
  title = "Menu" // not present in the object (default value is used)
} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
alert(item1);  // Cake
alert(item2);  // Donut
// All properties of options object except extra that is absent in the left
part, are assigned to corresponding variables.
```

Date and Time

To declare a new data object with current date and time:

```
let now = new Date();
alert( now ); // shows current date/time
//Creating a datetime object specifing the milliseconds passed since
01/01/1970
// 0 means 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );

// now add 24 hours, get 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
let Dec31_1969 = new Date(-24 * 3600 * 1000);
alert( Dec31_1969 );
// We can create a date passing:
new Date(year, month, date, hours, minutes, seconds, ms)
Create the date with the given components in the local time zone. Only the
first two arguments are obligatory.
```

The year should have 4 digits. For compatibility, 2 digits are also accepted and considered 19xx, e.g. 98 is the same as 1998 here, but always using 4 digits is strongly encouraged.
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.

we can retrieve the info using:

-getFullYear(), getMonth(),getDate() etc...

This methods work with local timezones, but also the UTC version exists:

```
// current date
let date = new Date();

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

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

All these methods also has the Setters().

Out of range components are distributed automatically:
Let's say we need to increase the date "28 Feb 2016" by 2 days. It may be "2 Mar" or "1 Mar" in case of a leap-year. We don't need to think about it. Just add 2 days. The Date object will do the rest:

```
let date = new Date(2016, 1, 28);
date.setDate(date.getDate() + 2);

alert( date ); // 1 Mar 2016

// more ocasionally

let date = new Date(2016, 0, 2); // 2 Jan 2016

date.setDate(1); // set day 1 of month
alert( date );

date.setDate(0); // min day is 1, so the last day of the previous month is assumed
alert( date ); // 31 Dec 2015
```

When Date object is converted to a number, it becomes the timestamp (milliseconds from 1/1/1970):

```
let date = new Date();
alert(+date); // the number of milliseconds, same as date.getTime()

// This is handy when we want to do time mesurements

let start = new Date(); // start measuring time

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

let end = new Date(); // end measuring time

alert( `The loop took ${end - start} ms` );

// But if we only need time, there is a better funcion -> Date.now()

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
```

Using Date.now() is much faster but it only matters in case of high performance necessities.

Parsing dates from strings

Date.parse from a string
The method Date.parse(str) can read a date from a string.

The string format should be: YYYY-MM-DDTHH:mm:ss.sssZ, where:

- YYYY-MM-DD – is the date: year-month-day.
- The character "T" is used as the delimiter.
- HH:mm:ss.sss – is the time: hours, minutes, seconds and milliseconds.
- The optional 'Z' part denotes the time zone in the format +-hh:mm. A single letter Z would mean UTC+0.
- Shorter variants are also possible, like YYYY-MM-DD or YYYY-MM or even YYYY.

The call to Date.parse(str) parses the string in the given format and returns the timestamp (number of milliseconds from 1 Jan 1970 UTC+0). If the format is invalid, returns NaN.

For instance:

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert(ms); // 1327611110417  (timestamp)
```

We can instantly create a new Date object from the timestamp:

```
let date = new Date( Date.parse('2012-01-26T13:51:50.417-07:00') );

alert(date);
```

Date and time in JavaScript are represented with the Date object. We can't create "only date" or "only time": Date objects always carry both.
Months are counted from zero (yes, January is a zero month).
Days of week in getDay() are also counted from zero (that's Sunday).
Date auto-corrects itself when out-of-range components are set. Good for adding/subtracting days/months/hours.
Dates can be subtracted, giving their difference in milliseconds. That's because a Date becomes the

timestamp when converted to a number.

Use Date.now() to get the current timestamp fast.

Note that unlike many other systems, timestamps in JavaScript are in milliseconds, not in seconds.

Json

Json (JabaScript Object Notation) is a general formato to represent values and objects. IJavascript provides two methods to work with JSON:

- JSON.stringify to convert objects into JSON.
- JSON.parse to convert JSON back into an object.

The resulting json string is called a JSON-encoded or serialized or stringified or marshalled object. We are ready to send it over the wire or put into a plain data store.

JSON is data-only language-independent specification, so some JavaScript-specific object properties are skipped by JSON.stringify.

- Function properties (methods).
- Symbolic keys and values.
- Properties that store undefined

```
let user = {
  sayHi() { // ignored
    alert("Hello");
  },
  [Symbol("id")]: 123, // ignored
  something: undefined // ignored
};

alert( JSON.stringify(user) ); // {} (empty object)
```

There must not be circular references:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: ["john", "ann"]
};

meetup.place = room;       // meetup references room
room.occupiedBy = meetup; // room references meetup

JSON.stringify(meetup); // Error: Converting circular structure to JSON
```

We can pass multiple arguments to stringify, but mostly it is used with just one argument. If we need to fine-tune the conversion process, we can use:

```
let json = JSON.stringify(value[, replacer, space])
```

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "John"}, {name: "Alice"}],
  place: room // meetup references room
};

room.occupiedBy = meetup; // room references meetup
//If we pass an array of properties, only these will be encoded.
alert( JSON.stringify(meetup, ['title', 'participants']) );
// {"title":"Conference","participants":[{},{}]}
```

Fortunately, we can use a function instead of an array as the replacer.

The function will be called for every (key, value) pair and should return the “replaced” value, which will be used instead of the original one. Or undefined if the value is to be skipped.

In our case, we can return value “as is” for everything except occupiedBy. To ignore occupiedBy, the code below returns undefined:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "John"}, {name: "Alice"}],
  place: room // meetup references room
};

room.occupiedBy = meetup; // room references meetup

alert( JSON.stringify(meetup, function replacer(key, value) {
  alert(`${key}: ${value}`);
  return (key == 'occupiedBy') ? undefined : value;
}));
```

```
/* key:value pairs that come to replacer:
:             [object Object]
title:        Conference
participants: [object Object],[object Object]
0:            [object Object]
name:         John
1:            [object Object]
name:         Alice
place:        [object Object]
number:       23
occupiedBy: [object Object]
*/
```

Please note that replacer function gets every key/value pair including nested objects and array items. It is applied recursively. The value of this inside replacer is the object that contains the current property.

The first call is special. It is made using a special “wrapper object”: {"": meetup}. In other words, the first (key, value) pair has an empty key, and the value is the target object as a whole. That’s why the first line is ":[object Object]" in the example above.

The idea is to provide as much power for replacer as possible: it has a chance to analyze and replace/skip even the whole object if necessary.

toJSON ovverride

```
let room = {
  number: 23,
  toJSON() {
    return this.number;
  }
};

let meetup = {
  title: "Conference",
  room
};

alert( JSON.stringify(room) ); // 23

alert( JSON.stringify(meetup) );
/*
  {
    "title":"Conference",
```

```
    "room": 23
  }
*/
```

##JSON.parse

```
let value = JSON.parse(str, [reviver]);
```

The reviver is a function that can be passed to the parse when we have to deal with different kind of data, e.g:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert( meetup.date.getDate() ); // Error!
```

in this case date is a string and is not converted in a Date by the parse funtcion, what we can do is:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( meetup.date.getDate() ); // now works!

//This also works with nested objects:
let schedule = `{
  "meetups": [
    {"title":"Conference","date":"2017-11-30T12:00:00.000Z"},
    {"title":"Birthday","date":"2017-04-18T12:00:00.000Z"}
  ]
}`;

schedule = JSON.parse(schedule, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( schedule.meetups[1].date.getDate() ); // works!
```