Institute of Data

Institute of Data

Intermediate JavaScript

- Methods of Primitive
- Primitive-like Object
- Numbers
- Strings
- Arrays
- Iterables
- Map and Set
- WeakMap and WeakSet
- Destructuring assignment
- Date and time
- JSON

Methods of primitive

A primitive as an object

Other Iteration methods

```
const n = 1.23456;
n.toFixed(2) ; // 1.23, an object wrapper for number with method `toFixed()`

const str = 'hello world';
str.toUpperCase() === (new String(str)).toUpperCase(); // we can explicitly create a wrapper
```

Primitive-like Object

String-like Object

Sometimes, an Object behaves like a string by defining a `toString` method

```
const user = {
  name: 'John',
  toString() {
        return this.name;
  }
                          };
console.log( 'hello   ' + user); // hello John
```

Primitive-like Object

Number-like Object

Sometimes, an Object behaves like a number by defining a `valueOf` method

```
const apple = {
  price: 100,
  valueOf() {
    return this.price;
  }
};
console.log( apple*3 ); //300
```

Primitive-like Object

adaptive number-like/string-like Object

an Object behaves like a number or string by defining a `[Symbol.toPrimitive](hint)` method.

```
const user = {
  name: 'John',
  money: 1000,
  [Symbol.toPrimitive]( hint ) {
    return hint === 'string' ? `{name: '${this.name}'}` : this.money;
  }
};

console.log( user ); // hint: string -> {name: "John"}
console.log( +user ); // hint: number -> 1000
console.log(u ser + 500) ; // hint: default -> 1500
```

Numbers

Decimal Numbers

- Large numbers

```
const billion = 100000000;
const billion = 1_000_000_000;
const billion = 1e9;
```

- Small numbers

```
const microSeconds = 0.000001;
const microSeconds = 1e-6;
```

Numbers

Hexadecimal numbers

Hexadecimal numbers are widely used in JavaScript to represent **colors**, **encode characters**, and for many other things. So naturally, there exists a shorter way to write them: `0x` and then the number.

```
alert( 0xff ); // 255
alert( 0xFF ); // 255 (the same, case insensitive)
```

Numbers

Binary and Octal numbers

Binary and octal numeral systems are rarely used, but also supported using the `0b` and `0o` **prefixes**:

```
const a = 0b11111111; // binary form of 255
const b = 0o377; // octal form of 255

a === b ; // true, the same number 255 at both sides
```

Numbers

Base conversion

The method `num.toString(base)` returns a string representation of num in the numeral system with the given **base**.
The base can vary **from 2 to 36**. By default it is **10**.

- `base=16` is used for **hex colors, character encodings** etc, digits can be **09 or A..F**.
- `base=2` is mostly for **debugging bitwise operations**, digits can be **0 or 1**.
- `base=36` is the maximum, digits can be **0..9 or A..Z**.

```
123456..toString(36) // 2n9c
```

Numbers

Imprecise calculations

Internally, a number is represented in 64-bit format IEEE-754, so there are exactly **64 bits** to store a **number**: **52** of them are used to store the **digits**, **11** of them store the position of the **decimal point** (they are zero for integer numbers), and **1** bit is for the **sign**.

- An Infinity. When the number is too big, it would **overflow the 64-bit** storage.

```
alert(1e500); // Infinity
```

Numbers

Imprecise calculations

- Loss of precision in float point number

```
0.1 + 0.2 ; //0.30000000000000004
```

- Loss of precision when the number of digits reaches 16 or more.

```
console.log( 9999999999999999 );//10000000000000000
```

Numbers

Tests: isFinite and isNaN

- `isNaN(value)` converts its argument to a number and then tests it for being NaN

```
alert( isNaN(NaN) ); // true
alert( isNaN("str") ); // true
alert( NaN === NaN ); // false
```

- `isFinite(value)` converts its argument to a number and returns true if it is a regular number, not NaN/Infinity/-Infinity.

```
alert( isFinite("15") ); // true
alert( isFinite("str") ); // false, because a special value: NaN
alert( isFinite(Infinity) ); // false, because a special value: Infinity
```

Numbers

ParseInt and parseFloat(soft conversion)

Numeric conversion using a plus `+` or `Number()` is strict. If a value is not exactly a number, it fails:

```
console.log( +"100px" ); // NaN
```

`parseInt` and `parseFloat` 'read' a number from a string until they can't. In case of an error, the gathered number is returned. The function parseInt returns an integer, whilst `parseFloat` will return a floating-point number.

```
console.log(parseInt('100px'));// 100
console.log(parseFloat('12.5em'));// 12.5
console.log(parseFloat('12.3.4')); //12.3
console.log(parseInt('a123')); // NaN
```

Numbers

parseInt with radix

The `parseInt()` function has an optional **second parameter**. It specifies the base of the numeral system, so parseInt can also parse strings of **hex numbers, binary numbers** and **so on**.

```
console.log(parseInt('0xff', 16)); // 255
console.log( parseInt('ff', 16) ); // 255, without 0x also works
console.log(parseInt('111', 2) ); //7
console.log(parseInt('2n9c', 36)); //123456
```

Strings

Definition

The textual data is stored as strings, The internal format for strings is always UTF-16, it is not tied to the page encoding.

Special characters

It is possible to create multiline strings with single and double quotes by using a so-called **newline character**, written as `\n`, which denotes a **line break**.

```
const guestList = "Guests:\n * John\n * Pete\n * Mary";

console.log(guestList); // a multiline list of guests
```

Strings

Other special characters

Character	Description
\r	Carriage return: \r\n to represent a line break in windows
\', \"	Quotes
\\	Backslash
\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. e.g `\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. e.g `\u{1F60D}` is a smiling face symbol 😍

Strings

Comparing strings

All strings are encoded using UTF-16. That means each character has a corresponding **numeric code**. when 2 strings are compared, under the hood, Javascript converts them into the **numeric codes** and compare them mathematically.

- `str.codePointAt(pos)`, returns the **numeric code** for the character at position `pos`

```
'Z'.codePointAt(0); // 90
```

- `String.fromCodePoint(code)`, creates a character by its **numeric code**.

```
String.fromCodePoint(90); // Z
```

Strings

Surrogate pairs

All frequently used characters have **2-byte codes**. Letters in most **european languages, numbers, and even most hieroglyphs**, have a 2-byte representation.

But 2 bytes only allow **65536 (2**16)** combinations which is not enough for every possible symbol. So rare symbols are encoded with **a pair of 2-byte characters** called **a surrogate pair**.

```
'𝒳'.length ; // 2, MATHEMATICAL SCRIPT CAPITAL X
'😂'.length ; // 2, FACE WITH TEARS OF JOY
'鮭'.length ; // 2, a rare Chinese hieroglyph
```

Array

Definition

A data structure to store an **ordered collection**.

Declaration

To create an array

```
const arr = new Array(1,2,3); // constructor method
const arr = [1, 2, 3]; // array literal
```

Array

Queue data structure, FIFO (First-In-First-Out)

- `push` appends an element to the end.
- `shift` get an element from the beginning, advancing the queue, so that the 2nd element becomes the 1st.

Stack data structure, LIFO (Last-In-First-Out)

- `push` adds an element to the end.
- `pop` takes an element from the end.

Array

Methods that work with the end of the array:

- `pop`, extracts the last element of the array and returns it:

```
const fruits = ["Apple", "Orange", "Pear"];
fruits.pop(); // remove "Pear"
alert( fruits ); // Apple, Orange
```

- `push`, append the element of the array:

```
const fruits = ["Apple", "Orange"];
fruits.push("Pear");
alert( fruits ); // Apple, Orange, Pear
```

Array

Methods that work with the beginning of the array:

- `shift`, extracts the first element of the array and returns it.

```
const fruits = ["Apple", "Orange", "Pear"];
fruits.shift() ; // remove Apple
alert( fruits ); // Orange, Pear
```

- `unshift`, add the element to the beginning of the array.

```
const fruits = ["Orange", "Pear"];
fruits.unshift('Apple');
alert( fruits ); // Apple, Orange, Pear
```

Arrays

Internals

An array is a special kind of object.The engine tries to store its elements in the contiguous memory area, one after another.

```
const fruits = ["Banana"]

const arr = fruits; // copy by reference (two variables reference the same array)

alert( arr === fruits ); // true

arr.push("Pear"); // modify the array by reference

alert( fruits ); // Banana, Pear - 2 items now
```

Array

Multidimensional arrays

Arrays can have items that are also arrays. We can use it for **multidimensional** arrays, for example to **store matrices**.

```
const matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

alert( matrix[1][1] ); // 5, the central element
```

Array

`toString` method

Arrays have their own implementation of `toString` method that returns a **comma-separated** list of elements.

```
const arr = [1, 2, 3];

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

Arrays do not have `Symbol.toPrimitive`, neither a viable `valueOf`, they implement **only** `toString` conversion, so empty array `[]` becomes an **empty string**.

```
alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
```

Array

`splice` Method

- Syntax

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

- It can **insert, remove and replace** items in array. It modifies `arr` **starting** from the **index** `start`, **removes** `deleteCount` items and then **inserts** `elem1, ..., elemN` at their place. Returns the array of removed elements.

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

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

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

Array

`slice` Method

- Syntax

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

- It returns 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.

```
const 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)
```

Array

`concat` Method

- Syntax

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

- The method `arr.concat` creates a **new array** that includes values from other arrays and additional items. It accepts any number of arguments: either **arrays** or **values**.

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

alert( arr.concat([3, 4]) ); // 1,2,3,4
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

Array

`forEach` Method

- Syntax

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

- The `arr.forEach` method allows to run a **function** for **every element** of the array.

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

Array

`indexOf` Method

- Syntax

```
arr.indexOf(item, from)
```

- It looks for `item` **starting** from index `from`, and returns the **index** where it was found, otherwise `-1`.

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

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

Array

`lastIndexOf` Method

- Syntax

```
arr.lastIndexOf(item, from)
```

- the same as `indexOf`, but it looks from right to left.

```
const arr = [1, 0, 1];

arr.lastIndexOf(1); // 2 instead of 0
```

Array

`find` Method

- Syntax

```
const 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 elements of the array, one after another, `item` is the element, `index` is its index, `array` is the **array** itself.

```
const users = [
  {id: 1, name: "John"},
  {id: 3, name: "Mary"}
];
const user = users.find(item => item.id == 1);
alert(user.name); // John
```

Array

`filter` Method

- Syntax

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

- It is similar to `find`, but `filter` returns **an array** of **all** matching elements.

```
const users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];
const someUsers = users.filter(item => item.id < 3);
alert(someUsers.length); // 2
```

Array

`map` Method

- Syntax

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

- It **transforms** the array by calling the **function for each element** of the array and returning the **new array of results**.

```
const lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);

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

Array

`sort` Method

- Syntax

```
arr.sort(function compareFn(firstEl, secondEl) { ... });
```

- It sorts the array **in place**, changing its **element order**. It also returns the sorted array, but the returned value is usually **ignored**, as `arr` itself is **modified**.
- If `compareFn` is omitted, the array elements are converted to strings, then sorted according to each character's **Unicode code point value**.

```
const arr = [ 1, 2, 15 ];
arr.sort();//1, 15, 2, The items are sorted as strings by default.
```

Array

`sort` Method

- With **Ordering function**, It will walk the array, compare its elements using the provided function: `compareFn` and **reorder** them, all we need is to provide the **implementation** of `compareFn` which does the comparison.

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

arr.sort(function(a, b) { return a - b; });// 1, 2, 15
```

Array

`reverse` Method

- Syntax

```
arr.reverse();
```

- It **reverses the order** of elements in `arr`, **mutates** the `arr`, and returns a **reference** to the `arr`.

```
const a = [1, 2, 3];

a.reverse();

console.log(a); // [3, 2, 1]
```

Array

`join` Method

- Syntax

```
arr.join(separator)
```

- It creates a string of `arr` items joined by `separator` between them. when `separator` is omitted, the `arr` elements are separated with a **comma** (`,`)

```
const arr = ['Wind', 'Water', 'Fire'];

arr.join(';'); // 'Wind;Water;Fire'
arr.join();    // 'Wind,Water,Fire'
```

Array

`reduce` Method

- Syntax

```
const value = arr.reduce(function(accumulator, item, index, array) {
  // ...
}, [initial]);
```

- when `function` is applied, the **result** of the **previous function call** is passed to the next one as the **first argument**, which is the accumulator that stores the **combined result** of all previous executions. And at the end it becomes the **result** of `reduce`.

```
[1, 2, 3, 4].reduce((sum, current) => sum + current, 0); //10
```

Array

Other Iteration methods

```
const arrayLikeObj  = { 0: 1, 1:4, 2:8, length: 3};
Array.prototype.reduce.call(arrayLikeObj, (acc, curr) => (acc + curr)); //  13

function foo(a, b, c) {
  const s = Array.prototype.join.call(arguments); //arguments is an array-like object
  console.log(s); // '1,a,true'
}
foo(1, 'a', true);
```

Iterables

***Iterable* objects** are a **generalization of arrays**, which allows us to make any object usable in a `for..of` loop.

`Symbol.iterator`

- The symbol specifies the **default iterator** for an object. It can also be **customized** in the way how we would like to implement the **iterator method** (`[Symbol.iterator]`) and use it `for .. of` loop.
- Note: **String, Array, TypedArray, Map, and Set** are all **built-in iterables**, so they can be used in `for .. of`` loop.

Iterables

`[Symbol.iterable]` implementation

- With plain function

```
const range = {
  from: 1,
  to: 5,
  [Symbol.iterator]() {
    this.current = this.from;
    return this;
  },
  next() {
    if (this.current <= this.to) {
      return { done: false, value: this.current++ };
    } else {
      return { done: true };
    }
  }
};

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

Iterables

`[Symbol.iterable]` implementation

- With generator function

```
const range = {
  from: 1,
  to: 5
};

range[Symbol.iterator] = function* () {
  for ( let i = this.from; i <= this.to; i++) {
    yield i;
  }
};

for ( const num of range ) {
  console.log(num);  // 1, 2, 3, 4, 5
}
```

Array

Static Method `Array.from`

- Syntax

```
Array.from(arrayLike, mapFn, thisArg)
Array.from(iterable, mapFn, thisArg)
```

- Conversion from **Array-like object** or **iterable** to **Array**

```
Array.from('foo'); // [ "f", "o", "o" ]
Array.from(new Set(['foo', 'bar', 'baz', 'foo'])); //[ "foo", "bar", "baz" ]
Array.from(new Map([[1, 2], [2, 4], [4, 8]])); //// [[1, 2], [2, 4], [4, 8]]

// Create an array based on a property of DOM Elements
const images = document.getElementsByTagName('img');
const sources = Array.from(images, image => image.src);
const insecureSources = sources.filter(link => link.startsWith('http://'));

function f() {
  return Array.from(arguments);
}
f(1, 2, 3);// [ 1, 2, 3 ]
```

Map

`Map` is a **collection** of **keyed data items**, just like an **Object**. But the main difference is that Map allows keys of **any type**.

Methods and properties

- `new Map()`: creates the map.
- `map.set(key, value)`: stores the value by the key, and returns map itself.
- `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**.

```
const 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.set({ name: "John" }, 123) // an object key
alert( map.get(1)   ); // 'num1'
alert( map.get('1') ); // 'str1'
alert( map.size ); // 4
```

Map

Iteration over Map

- `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`.

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

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

for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}

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

Map

Conversions with Object

- `Object.fromEntries` create an **Object** from `Map.entries()`.

```
const priceMap = new Map([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

const prices = Object.fromEntries(priceMap.entries());

console.log(prices);// { banana: 1, orange: 2, meat: 4 }
```

Map

Conversions with Object

- `Object.entries`
  create a **Map** from an **Object**.

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

const map = new Map(Object.entries(obj));

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

Set

A `Set` is a special type collection: **"set of values"** (without keys), where each value may occur only once.

Methods

- `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.

```
const visitors = new Set();
const john = { name: "John" };
const pete = { name: "Pete" };
const mary = { name: "Mary" };

visitors.add(john);
visitors.add(pete);
visitors.add(mary);
visitors.add(john);
visitors.add(mary);
alert( visitors.size ); // 3
```

Set

Iteration over `Set`

- **Loop over** a **set** either with `for..of` or using `forEach`.

```
const set = new Set(["oranges", "apples", "bananas"]);
for (let value of set) alert(value);

set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

- Methods for iterators
  - `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.

WeakMap

- The keys must be objects, not primitive values

```
const weakMap = new WeakMap();
const obj = {};

weakMap.set(obj, "ok"); // works fine (object key)
weakMap.set("test", "Whoops"); // Error, because "test" is not an object
```

- If there are no other **references** to that **object key**, it will be removed from memory (and from the map) **automatically**.

```
const john = { name: "John" };
const weakMap = new WeakMap();

weakMap.set(john, "...");

john = null; // overwrite the reference, john is removed from memory!
```

WeakMap

Methods

`WeakMap` does not support **iteration** and methods **keys(), values(), entries()**, so there is 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)`

WeakMap

Use Case: additional data

When working with an object that **"belongs"** to another code,eg. a 3rd-party library, and would like to **store some data associated with it**, that should only **exist** while the object is **alive**.

So, we put such data to a `WeakMap`, using the **object as the key**, and when the object is **garbage collected**, such data will automatically be **destroyed**.

WeakMap

Use Case: additional data

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

export default function countUser(user) {
  const count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

```
// 📁 main.js
import countUser from './visitsCount.js';

const john = { name: "John" };
countUser(john); // count his visits
john = null;// later john leaves us
```

WeakMap

Use case:caching

We can store **cached results** or **"memoized result"** from a function, so that future calls on the **same object** can **reuse** it.

- Build a caching util

```
// 📁 cache.js
let cache = new WeakMap();
// calculate and remember the result
export default function process(obj) {
  if (!cache.has(obj)) {
    const result = /* calculate the result for */ obj;
    cache.set(obj, result);
  }
  return cache.get(obj);
}
```

WeakMap

Use case: caching

- consume the **caching** util

```
// 📁 main.js
import process from './cache.js';

const obj = {/* some object */};

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

// ...later, when the object is not needed any more:
obj = null;
// When obj gets garbage collected, cached data will be removed as well
```

Destructuring assignment

Destructuring assignment is a **special syntax** that allows us to **"unpack"** arrays or objects into a **bunch of variables**.

Array destructuring

- It **"destructures"** by **copying** items into variables.

```
const arr = ["John", "Smith"]
const [firstName, surname] = arr;
alert(firstName); // John
alert(surname);  // Smith
```

- It **ignores** elements using **commas**.

```
const [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
console.log( title ); // Consul
```

Destructuring assignment

Array destructing

- It works with any **iterable** on the **right-side**.

```
const [ a, b, c ] = "abc"; // ["a", "b", "c"]

const [ one, two, three ] = new Set( [ 1, 2, 3 ] );
const [ [ type, quantity ] ] = new Map([ [ 'apple', 50 ] ] );
```

- It **assigns** to anything at the **left-side**.

```
const user = {};

[ user.name, user.surname ] = "John Smith".split(' ');

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

Destructuring assignment

Array destructing

- Looping with `.entries()`

```
const user = {
  name: "John",
  age: 30
};
for (const [key, value] of Object.entries(user)) {
  alert(`${key}:${value}`); // name:John, then age:30
}
```

- Swap **variables** trick.

```
const guest = "Jane", admin = "Pete";
[guest, admin] = [admin, guest];
```

Destructuring assignment

The array rest `...`

Usually, if the array is longer than the list at the left, the **"extra"** items are **omitted**.If we'd like also to gather all that follows, we can add one more parameter that gets **'the rest'** using three dots `...`

```
const [ name1, name2, ...titles ] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
console.log( titles.length );// 2
console.log( titles ) ;// ["Consul", "of the Roman Republic"]
```

Default values

```
const [ name = "Guest", surname = "Anonymous" ] = ["Julius"];
alert(name);    // Julius (from array)
alert(surname); // Anonymous (default used)
```

Destructuring assignment

Object destructuring

- An **existing** object at the **right side** is to be split into variables. The left side contains an **object-like "pattern"** for **corresponding properties**.

```
// changed the order in let {...}
const { height, width, title } = { title: "Menu", height: 200, width: 100 };
```

- Assign a **property** to a variable with **another name**.

```
const options = {
  width: 100,
  height: 200
};
const { width: w, height: h } = options;
```

Destructuring assignment

Object destructuring

- Assign default value to **missing property**

```
const { width: w = 100, height: h = 200, title }  = { title: "menu" };
```

- **Nested** destructing

```
const options = {
  size: { width: 100,  height: 200 },
  items: [ "Cake", "Donut" ],
};
const {
  size: { width, height },
  items: [ item1, item2 ], // assign items here
} = options;
```

Destructuring assignment

Object destructuring

- Smart function parameters

There are times when a function has **many parameters**, most of which are optional.We can pass **parameters as an object**, and the function immediately **destructurizes them into variables**.

```
const options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};
function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {
  alert( `${title} ${width} ${height}` ); // My Menu 200 100
  alert( items ); // Item1, Item2
}
showMenu(options);
```

Destructuring assignment

Object destructuring

- The rest pattern `...`.

It just like we did with arrays when we have more variable than we need.

```
const options = {
  title: "Menu",
  height: 200,
  width: 100
};
const { title, ...rest } = options;

// now title="Menu", rest = { height: 200, width: 100 }
alert(rest.height);  // 200
alert(rest.width);   // 100
```

Date and time

Creation

- Without arguments `new Date()`, It creates a `Date` for the current date and time.

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

- With arguments `new Date(milliseconds)`, It create a **Date object** with the time equal to number of **milliseconds** (1/1000 of a second) passed after the **Jan 1st of 1970 UTC+0**.

```
const Jan01_1970 = new Date(0); // 0 means 01.01.1970 UTC+0

const Jan02_1970 = new Date(24 * 3600 * 1000); // now add 24 hours, get 02.01.1970 UTC+0
```

Date and time

Create

- `new Date(datestring)`, it parses a string into a **Date object**, which is the same as `Date.parse`

```
const date = new Date("2017-01-26"); // The time is not set, so it's assumed to be midnight GMT and
```

- `new Date(year, month, date, hours, minutes, seconds, ms)`, it create the date with the given components in the local time zone. Only the first **two arguments are obligatory**.
  - 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

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

Date and time

Get date components

- `getFullYear()`: Get the year (`4` digits).
- `getMonth()`: Get the month, from **0 to 11**.
- `getDate()`: Get the day of month, from **1 to 31**.
- `getHours(), getMinutes(), getSeconds(), getMilliseconds()`: Get the corresponding time components.
- `getDay()`: Get the day of week, from **0 (Sunday) to 6 (Saturday)**. The first day is always Sunday.
- `getTime()`: Returns the timestamp for the date, a number of milliseconds passed from the **January 1st of 1970 UTC+0**.
- `getTimezoneOffset()`: Returns the difference between **UTC** and the **local time zone**, **in minutes**.

Date and time

Set date components

- `setFullYear(year, [month], [date])` Doc
- `setMonth(month, [date])` Doc
- `setDate(date)` Doc
- `setHours(hour, [min], [sec], [ms])` Doc
- `setMinutes(min, [sec], [ms])` Doc
- `setSeconds(sec, [ms])` Doc
- `setMilliseconds(ms)` Doc
- `setTime(milliseconds)` Doc

Date and time

Convert `Date` to `Number`

- `+new Date()`
- `new Date().getTime()`
- `Date.now()` // better JS performance

Convert string to `Date` with `Date.parse(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 **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`.

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

JSON

The JSON (JavaScript Object Notation) is a general format to represent values and objects. It is described as in RFC 4627 standard.

- `JSON.stringify` method to convert **objects** into **JSON**.

```
const student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js'],
  wife: null
};

JSON.stringify(student); //'{"name":"John","age":30,"isAdmin":false,"courses":["html","css","js"],"wife":null}'
```

JSON

- **JSON** is **data-only** language-independent specification, so some **JavaScript-specific object properties** are **skipped** by `JSON.stringify`. namely:
  - Function properties (methods).
  - Symbolic keys and values.
  - Properties that store undefined.

```
const user = {
  sayHi() { // ignored
    alert("Hello");
  },
  [Symbol("id")]: 123, // ignored
  something: undefined // ignored
};
alert( JSON.stringify(user) ); // {} (empty object)
```

JSON

- No circular references with `JSON.stringify`.

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

const 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
```

JSON

Excluding and transforming: `replacer`

- The second argument in `JSON.stringify(value[, replacer, space])` is **Array of properties to encode** or **a mapping function function(key, value)**.
- `replacer` use case: **filter out** circular references

```
const room = {
  number: 23
};
const meetup = {
  title: "Conference",
  participants: [{name: "John"}, {name: "Alice"}],
  place: room // meetup references room
};

room.occupiedBy = meetup; // room references meetup
alert( JSON.stringify(meetup, ['title', 'participants']) );// {"title":"Conference","participants":[{},{}]}
```

JSON

- `replacer` use case: **'retain' JS specific properties** except `Symbol`

```
const user = {
  sayHi() { alert("Hello");},
  [Symbol("id")]: 123,
  something: undefined
};
JSON.stringify(user, function(key, value) {
  if( typeof value === 'function') {
    return value.toString();
  }
  if( typeof value === 'undefined' ) {
    return ''; // convert undefined value to empty string
  }
  return value;
});
```

JSON

`JSON.stringify` Formatting: space

- The **third** argument of `JSON.stringify(value, replacer, space)` is the **number of spaces** to use for **pretty** formatting.

```
const user = {
  name: "John",
  age: 25
};
alert(JSON.stringify(user, null, 2));
/* two-space indents:
{
  "name": "John",
  "age": 25
}
*/
```

JSON

`JSON.stringify`: custom "toJSON"

Like `toString` for **string conversion**, an object may provide method `toJSON` for **to-JSON conversion**. `JSON.stringify` automatically calls it whenever **available**.

```
const room = {
  number: 23,
  toJSON() { return this.number; }
};
const meetup = {
  title: "Conference",
  room
};
alert( JSON.stringify(room) ); // 23
alert( JSON.stringify(meetup) ); // '{"title":"Conference","room":23}'
```

JSON

`JSON.parse` method

- Syntax

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

- `str` is the **JSON string** to be **decoded**, `reviver` is an **optional function(key, value)** and used to **transform** the value.

```
let schedule = `{
  "meetups": [
    {"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[0].date.getDate() ); //18
```

Questions?