---

date: '2019-01-13' publish: 'true' category: 'note' author: 'Maitrik Patel'

title: 'Javascript' description: 'Javascript is taking over the world'

topics: 'tools, development'

source: 'Github'

Reading Tuts

- MDN JS
- JavaScript Information
- You don't know JS
- Resources for Staying on Top of JavaScript
- Eloquent Javascript

Tuts

- Javascript Enlightenment
- JS Video Tuts

Articles and Guideline

- The Basics of Object-Oriented JavaScript
- JavaScript Regular Expression
- JS without JQuery
- Principles of writing consistent, idiomatic JS
- JS Best Practice
- AngularJS Style Guide
- DOM manipulation in vanilla JS
- Javascript variable definitions scope

Tools

- JS Object method explorer
- JavaScript Visualizer

Javascript.info Basic

- JavaScript is a multi-paradigm language, supporting imperative/procedural programming along with OOP (Object-Oriented Programming) with prototypal inheritance.and functional programming.
- JavaScript interprets the line break as an “implicit” semicolon. This is called an automatic semicolon insertion.
- Object Creation Pattern - Encapsulation
- Object Reuse Pattern - Inheritance

```
<!-- works -->
alert('Hello')
alert('World')

<!-- works -->
alert("There will be an error")
[1, 2].forEach(alert)
```

use strict

- To fully enable all features of modern JavaScript, we should start scripts with "use strict".
- The "use strict" directive switches the engine to the “modern” mode, changing the behavior of some built-in features.
- Strict mode is enabled by placing "use strict" at the top of a script or function.

Data-Types

Distinctions between primitives and objects

- A primitive
  - All other types are called “primitive” because their values can contain only a single thing (be it a string or a number or whatever).
  - Is a value of a primitive type.
  - There are 6 primitive types: string, number, 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.

Typeof

- We can call typeof thing to figure this out. Generally, the most useful types are "number," "string," "function," and of course, "object."

```
var someObject = {someProperty: someValue};
console.log( typeof someObject );
```

- We show how to use hasOwnProperty in the last two lines. It returns true or false, based on whether an object has a certain property.
- ObjectName["PropertyName"]

```
var myObj = {
        name : '"lol"'
};
console.log( myObj.hasOwnProperty('name') );
console.log( myObj.hasOwnProperty('nickname') );
```

Type Conversions

- ToString, String(value) – Occurs when we output something. Can be performed with String(value). The conversion to string is usually obvious for primitive values.
- ToNumber, Number(value)– Occurs in math operations. Can be performed with Number(value).

```
undefined       -> NaN
null    -> 0
true / false    -> 1 / 0
string  -> The string is read "as is", whitespaces from both sides are
ignored ->
An empty string becomes 0. An error gives NaN
```

- ToBoolean, Boolean(value) – Occurs in logical operations. Can be performed with Boolean(value).

```
0, null, undefined, NaN, ""     -> false
any other -> value
```

"===" vs "=="

- "===" : same type and have the same value, then === produces true and !== produces false.
- "==" : evil-twins/double-equal operator, however, tries to coerce the values before comparing them
  - Double equals also performs type coercion.
- Falsy values : false, null, undefined, "" (empty string), 0, NaN

Null vs Undefined

- **Null**
  - null is an empty or non-existent value.
  - null must be assigned.
  - when using typeof to test null, it returns object
- **Undefined**
  - Undefined most typically means a variable has been declared, but not defined.

Tables	Undefined	Null
Definition	variable has been declared but not yet been assigned a value	assignment value that means "no value"
Type	Undefined	Object
JSON	Invalid	Valid
Nature	Variable declared but not yet assigned	Represent intentional absence of object value

Tables	Undefined	Null
Check	typeof variableName === “undefined”	variableName === null
Arithmetic	Not-a-number (NaN) error	treated as zero value
Comparison	Equality operator will return true	Identity operator will return false
Identifier	Can be an identifier for a property of global object	Not an identifier for a property of the global object

Comparisons

- Comparison operators return a boolean value.
- Strings are compared letter-by-letter in the “dictionary” order.
- When values of different types are compared, they get converted to numbers (with the exclusion of a strict equality check).
- The values null and undefined equal == each other and do not equal any other value.
- Be careful when using comparisons like > or < with variables that can occasionally be null/undefined. Checking for null/undefined separately is a good idea.

"While" vs "For" vs "Do/While"

- FOR loops are great for doing the same task over and over when you know ahead of time how many times you'll have to repeat the loop.

- WHILE loops are ideal when you have to loop, but you don't know ahead of time how many times you'll need to loop.

- Sometimes you want to make sure your loop runs at least one time no matter what. When this is the case, you want a modified while loop called a DO/WHILE loop.

```
#FOR Loop
for( var i = 0; i < 10 ; i++){

}

#WHILE Loop
var myCond = true;

while(myCond){
                console.log("While is here!");
                myCond = false;
}

var myCondi = false;

do{
                console.log("Do/While is here!");
}while(myCondi)
```

Function

- Functions are values. They can be assigned, copied or declared in any place of the code.
- **Function Declaration**
- If the function is declared as a separate statement in the main code flow, that's called a "Function Declaration".
- Function Declarations are processed before the code block is executed. They are visible everywhere in the block.
- a function, declared as a separate statement, in the main code flow.

  ```
  function sum(a, b) {
          return a + b;
  }
  ```

- **Function Expression**
- If the function is created as a part of an expression, it's called a "Function Expression".
- Function Expressions are created when the execution flow reaches them.
- A function, created inside an expression or inside another syntax construct. Here, the function is created at the right side of the "assignment expression"

  ```
  let sum = function(a, b) {
          return a + b;
  };
  ```

- **Callback functions**
- The arguments of ask are called callback functions or just callbacks.

  ```
  // Without CallBack
  function ask(question, yes, no) {
          if (confirm(question)) yes()
          else no();
  }

  function showOk() {
          alert( "You agreed." );
  }

  function showCancel() {
          alert( "You canceled the execution." );
  }
  ```

```
// usage: functions showOk, showCancel are passed as arguments to ask
ask("Do you agree?", showOk, showCancel);

// With CallBack
ask(
        "Do you agree?",
        function() { alert("You agreed."); },
        function() { alert("You canceled the execution."); }
);
```

- **Function Expression vs Function Declaration**
- A Function Expression is created when the execution reaches it and is usable from then on.
- Function Declaration is usable in the whole script/code block.

```
// The Function Declaration sayHi is created when JavaScript is preparing
// to start the script and is visible everywhere in it.
sayHi("John"); // Hello, John

function sayHi(name) {
        alert( `Hello, ${name}` );
}

// A Function Expression won't work

sayHi("John"); // error!

let sayHi = function(name) {  // (*) no magic any more
        alert( `Hello, ${name}` );
};
```

Object

- Let's go back to the analogy of computer languages being like regular spoken languages. In English, you have nouns (which you can think of as "things") and verbs (which you can think of as "actions"). Until now, our nouns (data, such as numbers, strings, or variables) and verbs (functions) have been separate.
- A constructor, as its name suggests, is designed to create and set up multiple instances of an object.
- An object literal on the other hand is one-off, like string and number literals, and used more often as configuration objects or global singletons (e.g. for namespacing).
- **Object literal:**
    - Literal notation creates a single object. Literal notation uses **curly brackets { }** and the object's default properties are defined within the brackets using **property:value** notation.

```
var objectName = {};

var james = {
        job: "programmer",
        married: false,
};

var myObject = {
                iAm : 'an object',
                whatAmI : function(){
                                alert('I am ' + this.iAm);
                }
}
```

- **Object constructor:**
    - When we write **bob = new Object( );** we are using a built-in constructor called Object. This constructor is already defined by the JavaScript language and just makes an object with **no properties or methods.**
    - Constructor notation involves defining an object constructor. And like defining a function, we use the function keyword. You can think of this constructor as a "template" from which you can create multiple objects. To create a new object from a constructor, we use the new keyword.

```
function Object(){
                this.iAm = 'an object';
                this.whatAmI = function(){
                                alert('I am ' + this.iAm);
                };
};

var objectName = new Object();

function Person(job, married) {
                this.job = job;
                this.married = married;
}

var gabby = new Person("student",true);
```

- **Differences between constructor and literal**
    - The constructor object has its properties and methods defined with the keyword 'this' in front of it, whereas the literal version does not.
    - In the constructor object the properties/methods have their 'values' defined after an equal sign '=' whereas in the literal version, they are defined after a colon ':'.
    - The constructor function can have (optional) semi-colons ';' at the end of each property/method declaration whereas in the literal version if you have more than one property or method, they MUST

be separated with a comma ',', and they CANNOT have semi-colons after them, otherwise JavaScript will return an error.

- **Bracket Notation : ObjectName["PropertyName"]**
    - An advantage of bracket notation is that we are not restricted to just using strings that is: no spaces and other limitations.
    - Square brackets notation obj["property"]. Square brackets allow to take the key from a variable, like obj[varWithKey].

```
let user = {};

// set
user["likes birds"] = true;

// get
alert(user["likes birds"]); // true

// delete
delete user["likes birds"];
```

- **Additional operators**
    - To delete a property: delete obj.prop.
    - To check if a property with the given key exists: "key" in obj.
    - To iterate over an object: for (let key in obj) loop.
- **Cloning and merging, Object.assign**
    - A variable stores not the object itself, but its “address in memory”, in other words “a reference” to it.
    - When an object variable is copied – the reference is copied, the object is not duplicated.
    - If we imagine an object as a cabinet, then a variable is a key to it. Copying a variable duplicates the key, but not the cabinet itself.

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

//-----------Copying object---------

let admin = user; // copy the reference
admin.name = "Pete" // Updated to user reference object
alert( user.name ); // Pete as object reference changed

//-----------Clone using loop---------
// Cloning
let user = {
        name: "John",
        age: 30
};

let clone = {}; // the new empty object
for (let key in user) {
        clone[key] = user[key];
}

clone.name = "Pete"; // changed the data in it
alert( user.name ); // John in original object as clone has its own
object reference

//---------Object.Assign------------
let clone = Object.assign({}, user);
```

Garbage collection

- The main concept of memory management in JavaScript is reachability.
- Simply put, “reachable” values are those that are accessible or usable somehow. They are guaranteed to be stored in memory.
- If object is unreachable and Garbage collector will junk the data and free the memory.

Symbols

- “Symbol” value represents a unique identifier.
- Symbols are guaranteed to be unique. Even if we create many symbols with the same description, they are different values.

```
// id is a symbol with the description "id"
let id1 = Symbol("id");
let id2 = Symbol("id");

alert(id1 == id2); // false
```

- Symbols don’t auto-convert to a string
- Symbol in an object literal, we need square brackets.

```
let id = Symbol("id");

let user = {
  name: "John",
  [id]: 123 // not just "id: 123"
};
```

- **Global symbols**

  - Symbols inside the registry are called global symbols. If we want an application-wide symbol, accessible everywhere in the code – that’s what they are for.

```
// read from the global registry
let id = Symbol.for("id"); // if the symbol did not exist, it is
created
```

  - The Symbol.keyFor internally uses the global symbol registry to look up the key for the symbol. So it doesn’t work for non-global symbols. If the symbol is not global, it won’t be able to find it and return undefined.

```
alert( Symbol.keyFor(Symbol.for("name")) ); // name, global symbol
alert( Symbol.keyFor(Symbol("name2")) ); // undefined, the argument
isn't a global symbol
```

Methods

- Functions that are stored in object properties are called “methods”.
- Methods allow objects to “act” like object.doSomething().
- Methods can reference the object as "this".

The "this" Keyword

- The keyword "this" acts as a placeholder, and will refer to whichever object called that method when the method is actually used.
- When a function is declared, it may use "this", but that "this" has no value until the function is called.

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

  sayHi() {
    alert(this.name);
  }
};

user.sayHi(); // John
```

- That function can be copied between objects.
- When a function is called in the “method” syntax: object.method(), the value of "this" during the call is object.
- "this" is undefined in strict mode.
- If we try to access this.name, there will be an error.

```
function sayHi() {
  alert(this);
}

sayHi(); // undefined
```

- In non-strict mode the value of "this" in such case will be the global object (window in a browser, we'll get to it later in the chapter Global object). This is a historical behavior that "use strict" fixes.
- NOTE: Arrow functions are special: they have no this. When this is accessed inside an arrow function, it is taken from outside.

Object to primitive conversion

- The object-to-primitive conversion is called automatically by many built-in functions and operators that expect a primitive as a value.
- There are 3 types (hints) of it:
  - "string" (for alert and other string conversions)
  - "number" (for maths)
  - "default" (few operators)
- The specification describes explicitly which operator uses which hint. There are very few operators that "don't know what to expect" and use the "default" hint. Usually for built-in objects "default" hint is handled the same way as "number", so in practice the last two are often merged together.
- The conversion algorithm is:
- Call obj[Symbol.toPrimitive] (hint) if the method exists,
- Otherwise if hint is "string"
  - try obj.toString() and obj.valueOf(), whatever exists.
- Otherwise if hint is "number" or "default"
  - try obj.valueOf() and obj.toString(), whatever exists.

Data Types

Number

- All numbers in JavaScript are stored in 64-bit format IEEE-754, also known as "double precision floating point numbers".
- Append "e" with the zeroes count to the number. Like: 123e6 is 123 with 6 zeroes.
- A negative number after "e" causes the number to be divided by 1 with given zeroes. That's for one-millionth or such.

- For different numeral systems:
  - Can write numbers directly in hex (0x), octal (0o) and binary (0b) systems
  - parseInt(str, base) parses an integer from any numeral system with base: 2 ≤ base ≤ 36.
  - num.toString(base) converts a number to a string in the numeral system with the given base.
- For converting values like 12pt and 100px to a number:
  - Use parseInt/parseFloat for the “soft” conversion, which reads a number from a string and then returns the value they could read before the error.
- For fractions:
  - Round using Math.floor, Math.ceil, Math.trunc, Math.round or num.toFixed(precision).
  - Make sure to remember there’s a loss of precision when working with fractions.
- More mathematical functions:
  - See the Math object when you need them. The library is very small, but can cover basic needs.

Strings

- There are 3 types of quotes. Backticks allow a string to span multiple lines and embed expressions.
- Strings in JavaScript are encoded using UTF-16.
- To get a character, use "[array]"
- To get a substring, use: slice or substring.
- To lowercase/uppercase a string, use: toLowerCase/toUpperCase.
- To look for a substring, use: indexOf, or includes/startsWith/endsWith for simple checks.
- To compare strings according to the language, use: localeCompare, otherwise they are compared by character codes.
- There are several other helpful methods in strings:
  - str.trim() – removes (“trims”) spaces from the beginning and end of the string.
  - str.repeat(n) – repeats the string n times.

Array

- Array is a special kind of object, suited to storing and managing ordered data items.
- The call to new Array(number) creates an array with the given length, but without elements.

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

// new Array (exceptionally rare)
let arr = new Array(item1, item2...);
let arr = new Array(2); // will it create an array of [2] ?
alert( arr[0] ); // undefined! no elements.
alert( arr.length ); // length 2
```

- Array implement only toString conversion

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

- Loop in 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.
```

- We can use an array as a deque with the following operations:

> sort(), reverse() and splice() modify the array but slice(), concat() and map() create a new array.

- **To add/remove elements:**
  - 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.
  - splice(position, deleteCount, ...items) – at index position delete deleteCount elements and insert items.
  - slice(start, end) – creates a new array, copies elements from position start till end (not inclusive) into it.
  - forEach(func(item, index, array){}) – calls func for every element, does not return anything.
  - concat(...items) – returns a new array: copies all members of the current one and adds items to it. If any of items is an array, then its elements are taken.
- **To search among elements:**
  - 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.
  - findIndex is like find, but returns the index instead of a value.
  - To iterate over elements:
  - forEach(func) – calls func for every element, does not return anything.
- **To transform the array:**

- map(func) – creates a new array from results of calling func for every element.
- 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.
- reduce(func, initial) – calculate a single value over the array by calling func for each element and passing an intermediate result between the calls.

```
let value = arr.reduce(function(previousValue, item, index, array) {
        // ...
}, initial);
```

- **Additionally:**
    - Array.isArray(arr) checks arr for being an array.

Iterables

- Iterable objects is a generalization of arrays. That's a concept that allows to make any object useable in a for..of loop.
- Objects that can be used in for..of are called iterable.
- Technically, iterables must implement the method named Symbol.iterator.
    - The result of obj[Symbol.iterator] is called an iterator. It handles the further iteration process.
    - An iterator must have the method named next() that returns an object {done: Boolean, value: any}, here done:true denotes the iteration end, otherwise the value is the next value.
- The Symbol.iterator method is called automatically by for..of, but we also can do it directly.
- Built-in iterables like strings or arrays, also implement Symbol.iterator.
- String iterator knows about surrogate pairs.
- Array.from(obj[, mapFn, thisArg]) makes a real Array of an iterable or array-like obj, and we can then use array methods on it. The optional arguments mapFn and thisArg allow us to apply a function to each item.

Map, Set, WeakMap and WeakSet

- **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.
    - unlike objects, keys are not converted to strings. Any type of key is possible.
    - 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() – clears 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
```

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

```
// array of [key, value] pairs
let map = new Map([
  ['1',  'str1'],
  [1,    'num1'],
  [true, 'bool1']
]);
```

- **Iteration over Map**
    - 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)
}
```

- **SET**
  - A 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 – is the elements count.

    ```
    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)
    }
    ```

- 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);
});
```

- 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**
    - WeakMap is a variant of Map that allows only objects as keys and removes them once they become inaccessible by other means.
    - It does not support operations on the structure as a whole: no size, no clear(), no iterations.
- **WeakSet**
    - WeakSet is a variant of Set that only stores objects and removes them once they become inaccessible by other means.
- WeakSet/WeakMap is a special kind of Set/Map that does not prevent JavaScript from removing its items from memory.
- Also does not support size/clear() and iterations.
- WeakMap and WeakSet are used as “secondary” data structures in addition to the “main” object storage.
- Once the object is removed from the main storage, if it is only found in the WeakMap/WeakSet, it will be cleaned up automatically.

Destructuring assignment

- Destructuring assignment is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables, as sometimes they are more convenient.

```
// let [item1 = default, item2, ...rest] = array
// we have an array with the name and surname
let arr = ["Ilya", "Kantor"]

// Before
let firstName = arr[0];
let surname = arr[1];

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

alert(firstName); // Ilya
alert(surname);  // Kantor
```

- **Object destructuring**

The destructuring assignment also works with objects.

```
// let {prop : varName = default, ...} = object
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

let {title, width, height} = options;

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

// chnage name
// assign value
// change order
// function({
//   incomingProperty: parameterName = defaultValue
//.})

let options = {
  title: "Menu"
};

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

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

- Smart function parameters

```
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);
```

Date and time

- Use it to store creation/modification times, to measure time, or just to print out the current date.
- Months are counted from zero (yes, January is a zero month).
- Days of week in getDay() are also counted from zero (that's Sunday).

```
new Date(year, month, date, hours, minutes, seconds, ms)
```

- Access date components
    - getFullYear()
    - getMonth()
    - getDate()
    - getHours()
    - getMinutes()
    - getSeconds()
    - getMilliseconds()
    - getTime()
    - getTime()
    - getTimezoneOffset()
- Setting date components
    - setFullYear(year [, month, date])
    - setMonth(month [, date])
    - setDate(date)
    - setHours(hour [, min, sec, ms])
    - setMinutes(min [, sec, ms])
    - setSeconds(sec [, ms])
    - setMilliseconds(ms)
    - setTime(milliseconds)
- Date.now()
    - 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 and doesn't put pressure on garbage collection.

JSON

- The JSON (JavaScript Object Notation) is a general format to represent values and objects.
- JavaScript provides methods: - JSON.stringify to convert objects into JSON. - JSON.parse to convert JSON back into an object.
- **JSON.stringify**
    - JSON is data-only cross-language specification, so some JavaScript-specific object properties are skipped by JSON.stringify.
    - Namely:
        - Function properties (methods).
        - Symbolic properties.
        - Properties that store undefined.

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

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

    - The important limitation: there must be no 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
```

    - Excluding and transforming: replacer
        - `let json = JSON.stringify(value[, replacer[, space]])`
        - JSON.stringify is used with the first argument only. But if we need to fine-tune the replacement process, like to filter out circular references, we can use the second argument of JSON.stringify.

```
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, ['title', 'participants', 'place',
'name', 'number']) );
/*
{
        "title":"Conference",
        "participants":[{"name":"John"},{"name":"Alice"}],
        "place":{"number":23}
}
*/
```

- **JSON.parse**

```
//Syntax : let value = JSON.parse(str[, reviver]);
let schedule = `{
        "meetups": [
                {"title":"Conference","date":"2017-11-
30T12:00:00.000Z"},
                {"title":"Birthday","date":"2017-04-18T12:00:00.000Z"}
        ]
}`;
// JSON.parse(str[, reviver]);
// JSON.parse(schedule, function());
schedule = JSON.parse(schedule, function(key, value) {
        if (key == 'date') return new Date(value);
        return value;
});

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

- **Custom “toJSON”**
    - Like toString for string conversion, an object may provide method toJSON for to-JSON conversion. JSON.stringify automatically calls it if available.

```
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
        }
*/
```

Recursion

- Recursion is a programming term that means a “self-calling” function. Such functions can be used to solve certain tasks in elegant ways.
- When a function calls itself, that’s called a recursion step. The basis of recursion is function arguments that make the task so simple that the function does not make further calls.
- A recursively-defined data structure is a data structure that can be defined using itself.
  - For instance, the linked list can be defined as a data structure consisting of an object referencing a list (or null).
  - `list = { value, next -> list }`
  - Linked list

  ```
          let list = { value: 1 };
          list.next = { value: 2 };
          list.next.next = { value: 3 };
          list.next.next.next = { value: 4 };
  ```

  - Trees like HTML elements tree or the department tree from this chapter are also naturally recursive: they branch and every branch can have other branches.

Rest Parameters "..."

- A function can be called with any number of arguments, no matter how it is defined.
- The rest parameters must be at the end

```
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");
```

- **The “arguments” variable**
  - There is also a special array-like object named arguments that contains all arguments by their index.
  - arguments is both array-like and iterable, it’s not an array. It does not support array methods, so we can’t call arguments.map(...) for example.
  - In old times, rest parameters did not exist in the language, and using arguments was the only way to get all arguments of the function, no matter their total number.
  - All arguments of a function call are also available in “old-style” arguments: array-like iterable object.
  - Arrow functions do not have "arguments"

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );

  // it's iterable
  // for(let arg of arguments) alert(arg);
}

// shows: 2, Julius, Caesar
showName("Julius", "Caesar");

// shows: 1, Ilya, undefined (no second argument)
showName("Ilya");
```

Spread operator

- ...Spread Operator
- The spread operator internally uses iterators to gather elements, the same way as for..of does.
- there’s a subtle difference between Array.from(obj) and [...obj]:
  - Array.from operates on both array-likes and iterables.
  - The spread operator operates only on iterables.
- the task of turning something into an array, Array.from tends to be more universal.

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];
let str = "Hello";

let merged = [0, ...arr, 2, ...arr2];
alert( [...str] ); // H,e,l,l,o
alert(merged); // 0,3,5,1,2,8,9,15 (0, then arr, then 2, then arr2)
```

Rest parameters vs spread operator.

- There's an easy way to distinguish between them:
  - When ... is at the end of function parameters, it's "rest parameters" and gathers the rest of the list of arguments into an array.
  - When ... occurs in a function call or alike, it's called a "spread operator" and expands an array into a list.
- Use patterns:
  - Rest parameters are used to create functions that accept any number of arguments.
  - The spread operator is used to pass an array to functions that normally require a list of many arguments.
- Together they help to travel between a list and an array of parameters with ease.

JS Closures

- **Lexical Environment**
  - In JavaScript, every running function, code block {...}, and the script as a whole have an internal (hidden) associated object known as the Lexical Environment.
  - "Lexical Environment" is a specification object. We can't get this object in our code and manipulate it directly.
  - A Lexical Environment is created when a code block runs and contains block-local variables. Example : If, For, While and {...}
  - The Lexical Environment object consists of two parts:
    1. Environment Record – an object that stores all local variables as its properties (and some other information like the value of this).
    2. A reference to the outer lexical environment, the one associated with the outer code.
  - A "variable" is just a property of the special internal object, associated with the currently executing block/function/script Environment Record.
  - Working with variables is actually working with the properties of that object.
  - "To get or change a variable" means "to get or change a property of that object.

  - When the code wants to access a variable – the inner Lexical Environment is searched first, then the outer one, then the more outer one and so on until the global one.
  - A function gets outer variables as they are now; it uses the most recent values.
  - One call – one Lexical Environment : a new function Lexical Environment is created each time a function runs.
  - Lexical Environment is cleaned up and deleted after the function run

```
//----No Clousre----
//----Access outer Lexical Envrionment Object variable---
let name = "John";

function sayHi() {
        alert("Hi, " + name);
}

name = "Pete"; // (*)

sayHi(); // Pete
```

- **Closures**
  - A closure is a function that remembers its outer variables and can access them.
  - All functions in JavaScript are closures
  - A closure is an inner function that has access to the outer (enclosing) function's variables—scope chain.
  - The closure has three scope chains:
    1. it has access to its own scope (variables defined between its curly brackets),
    2. it has access to the outer function's variables, and
    3. it has access to the global scope variables.

```
function makeWorker() {
        let name = "Pete";
        return function() {
                alert(name);
        };
}
let name = "John";
// create a function
let work = makeWorker();
// call it
work(); // Pete

// ----- Example Two ------

let count = 100;
function makeCounter() {
        let count = 0;
        return function() {
                return count++; // has access to the outer counter
        };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2

// ----- Example Three ------

let c = 4
const addX = x => n => n + x
const addThree = addX(3)
let d = addThree(c)
console.log('example partial application', d) // example partial
application 7

//---Real life example------

function makeSizer(size) {
        return function() {
                document.body.style.fontSize = size + 'px';
        };
}

var size12 = makeSizer(12);
var size14 = makeSizer(14);
var size16 = makeSizer(16);

document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;

//--- html code
<a href="#" id="size-12">12</a>
<a href="#" id="size-14">14</a>
<a href="#" id="size-16">16</a>
```

- **Immediately-invoked function expressions - IIFE**
  - The Function Expression is wrapped with parenthesis (function {...})
  - Parentheses around the function is a trick to show JavaScript that the function is created in the context of another expression, and hence it's a Function Expression: it needs no name and can be called immediately.

```
// Ways to create IIFE
(function() {
        alert("Parentheses around the function");
})();

(function() {
        alert("Parentheses around the whole thing");
}());

!function() {
        alert("Bitwise NOT operator starts the expression");
}();

+function() {
        alert("Unary plus starts the expression");
}();
```

Var vs Let vs Const

- Var : Global and Function scope
    - var variables are either function-wide or global, they are visible through blocks.
    - Variables have no block scope, they are visible minimum at the function level.
    - Hoisting : Variable declarations are processed at function start.
- Let : Will provide true block scoping, unlike var
    - Ideal to use it for programming as it will help to avoid hosting.
- Const
    - will make variable Read only
    - Never change , will give error if you try to change it.
    - Will shadow outer declaration
    - Block Scoping
    - We generally use upper case for constants that are “hard-coded”. Or, in other words, when the value is known prior to execution and directly written into the code.

Global object

- The global object provides variables and functions that are available anywhere.
- “window” object
    - "let/const" doesn't create a window property
    - The value of this in the global scope is window.
    - All scripts share the same global scope, so variables declared in one script.js file become visible in another ones:

```
// Window 1
<script>
        var a = 1;
        let b = 2;
</script>

// Window 2
<script>
        alert(a); // 1
        alert(b); // 2
</script>
```

- Different scripts (possibly from different sources) see variables of each other.
- The idea to merge multiple aspects into a single window object was to “make things simple", but the multi-purpose window is considered a design mistake in the language.
- Solution:
  - Using "script type='module'" fixes the design flaw of the language by separating top-level scope from window.
  - Such script is considered a separate “module” with its own top-level scope (lexical environment).

  ```
  <script type="module">
  // ----- var x does not become a property of window -----
  var x = 5;
  alert(window.x); // undefined
  // ----- this in a module is undefined -----
  alert(this); // undefined
  let x = 5;
  </script>

  <script type="module">
          alert(window.x); // undefined
          alert(x); // Error: undeclared variable
  </script>
  ```

- Valid uses of the global object
  - create “polyfills”: add functions that are not supported by the environment (say, an old browser), but exist in the modern standard.

  ```
  if (!window.Promise) {
          window.Promise = ... // custom implementation of the
  modern language feature
  }
  ```

Functional Object

- Functions are objects.
- JavaScript, functions are first-class objects, because they can have properties and methods just like any other object.
- Named Function Expression
  - Named Function Expression, or NFE, is a term for Function Expressions that have a name.

  ```
  let sayHi = function func(who) {
          alert(`Hello, ${who}`);
  };
  ```

  - It allows the function to reference itself internally.
  - It is not visible outside of the function.

"new Function" Sysntax

- "new Function" allows to turn any string into a function.

```
// Sysntax
// let func = new Function ([arg1[, arg2[, ...argN]],] functionBody)

let sum = new Function('a', 'b', 'return a + b');
alert( sum(1, 2) ); // 3
```

- Functions created with new Function, have [[Environment]] referencing the global Lexical Environment, not the outer one.
- Hence, they cannot use outer variables. But that's actually good, because it saves us from errors. Passing parameters explicitly is a much better method architecturally and causes no problems with minifiers.

Scheduling: setTimeout and setInterval

- setTimeout : allows to run a function once after the interval of time.

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

function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}
setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

- setInterval : allows to run a function regularly with the interval between the runs.

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

```
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);
// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
Modal
```

- Recursive setTimeout
    - Recursive setTimeout guarantees a delay between the executions, while setInterval – does not.
    - The real delay between func calls for setInterval is less than in the code because the time taken by func's execution “consumes” a part of the interval.

```
// he real delay between func calls for setInterval is less than in
the code
let i = 1;
setInterval(function() {
        func(i);
}, 100);

// The recursive setTimeout guarantees the fixed delay (here 100ms).
let i = 1;
setTimeout(function run() {
        func(i);
        setTimeout(run, 100);
}, 100);
```

- Zero time scheduling
    - Zero-timeout scheduling setTimeout(...,0) is used to schedule the call “as soon as possible, but after the current code is complete”.

```
setTimeout(() => alert("World"));
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
    - Where do we use it ?
        - Trick to split CPU-hungry tasks using setTimeout.
        - To let the browser do something else while the process is going on (paint the progress bar).

Call / Apply / Bind

- Call
    - The call() method is used to call a function with a given this and arguments provided to it individually.

- myFunc.call(context, arg1, arg2…) – calls func with given context and arguments.

- Apply

  - The apply() method is an important method of the function prototype and is used to call other functions with a provided this keyword value and arguments provided in the form of array or an array like object.

  - myFunc.apply(context, [args]) – calls func passing context as this and array-like args into a list of arguments.

  - The generic call forwarding is usually done with apply:

```
let wrapper = function() {
        return original.apply(this, arguments);
}
```

- Bind

  - In JavaScript it's easy to lose this. Once a method is passed somewhere separately from the object – this is lost.
  - The bind() method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

```
let user = {
        firstName: "John",
        sayHi() {
                alert(`Hello, ${this.firstName}!`);
        }
};

setTimeout(user.sayHi, 1000); // Hello, undefined!
// When setTimeout invokes it later, this will refer to something
useless
```

  - Functions provide a built-in method bind that allows to fix this.
  - `let boundFunc = func.bind(context);`
  - The result of func.bind(context) is a special function-like "exotic object", that is callable as function and transparently passes the call to func setting this=context.
  - Method func.bind(context, ...args) returns a "bound variant" of function func that fixes the context this and first arguments if given.

```
let user = {
        firstName: "John",
        say(phrase) {
                alert(`${phrase}, ${this.firstName}!`);
        }
};

let say = user.say.bind(user);

say("Hello"); // Hello, John ("Hello" argument is passed to say)
say("Bye"); // Bye, John ("Bye" is passed to say)
```

- apply(), call(), and bind() all take a this argument as a context to execute a function in, but call() and apply() invoke the function immediately where bind() returns a function that we can pass around or store as needed.

Partial Function

- partial function application – we create a new function by fixing some parameters of the existing one.
- partial application is useful when we have a very generic function and want a less universal variant of it for convenience.

```
function mul(a, b) {
  return a * b;
}

let double = mul.bind(null, 2);

alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10
```

Currying

- Currying is a transformation of functions that translates a function from callable as f(a, b, c) into callable as f(a)(b)(c).
- The currying requires the function to have a known fixed number of arguments.
- Currying is great when we want easy partials.

```
function curry(f) { // curry(f) does the currying transform
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}

// usage
function sum(a, b) {
  return a + b;
}
```

```
let carriedSum = curry(sum);
alert( carriedSum(1)(2) ); // 3
```

- Advanced currying allows the function to be both callable normally and partially.
- Most implementations of currying in JavaScript are advanced, as described: they also keep the function callable in the multi-argument variant.

```
function curry(func) {

  return function curried(...args) {
    if (args.length >= func.length) {
      return func.apply(this, args);
    } else {
      return function(...args2) {
        return curried.apply(this, args.concat(args2));
      }
    }
  };
}
function sum(a, b, c) {
  return a + b + c;
}

let curriedSum = curry(sum);

alert( curriedSum(1, 2, 3) ); // 6, still callable normally
alert( curriedSum(1)(2,3) ); // 6, currying of 1st arg
alert( curriedSum(1)(2)(3) ); // 6, full currying
```

Arrow functions

- Arrow functions have no “this”
- Arrow functions do not have this. If this is accessed, it is taken from the outside.
- Arrow functions can’t be used as constructors. They can’t be called with new.
- Arrows have no “arguments”

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
                // Traditional func will give error
    this.students.forEach(function(student) {
      // Error: Cannot read property 'title' of undefined
      alert(this.title + ': ' + student)
    });

                // Arrow function will work
```

```
            this.students.forEach(
      student => alert(this.title + ': ' + student)
    );
  }
};

group.showList();
```

Property flags and descriptors

- Objects can store properties and property was a simple “key-value” pair to us.
- However, Object properties, besides a value, have three special attributes (so-called “flags”)
    - writable – if true, can be changed, otherwise it’s read-only.
    - enumerable – if true, then listed in loops, otherwise not listed.
    - configurable – if true, the property can be deleted and these attributes can be modified, otherwise not.
    - Object.getOwnPropertyDescriptor allows to query the full information about a property.

```
// Syntax : Object.getOwnPropertyDescriptor(obj, propertyName);
let user = {
        name: "John"
};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/* property descriptor:
{
        "value": "John",
        "writable": true,
        "enumerable": true,
        "configurable": true
}
*/
```

- Object.defineProperty allows to change property.

```
// Syntax : Object.defineProperty(obj, propertyName, descriptor)
// For Multiple properties : Object.defineProperties(obj, descriptors)
let user = {};

Object.defineProperty(user, "name", {
        value: "John",
        writable: false,
        enumerable: true,
configurable: true
});

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
        "value": "John",
        "writable": false,
        "enumerable": true,
        "configurable": true
}
*/
```

Property Getter and Setters

- Two type of property in object.
    - data properties :
    - accessor properties : essentially functions that work on getting and setting a value
        - Accessor properties are only accessible with get/set.
        - get – a function without arguments, that works when a property is read,
        - set – a function with one argument, that is called when the property is set,
        - enumerable – same as for data properties,
        - configurable – same as for data properties.

```
let user = {
        name: "John",
        surname: "Smith",

        get fullName() {
                return `${this.name} ${this.surname}`;
        },

        set fullName(value) {
                [this.name, this.surname] = value.split(" ");
        }
};

// set fullName is executed with the given value.
user.fullName = "Alice Cooper";

alert(user.name); // Alice
alert(user.surname); // Cooper
```

    - Smarter getters/setters

- Getters/setters can be used as wrappers over “real” property values to gain more control over them.

```
let user = {
        get name() {
                return this._name;
        },

        set name(value) {
                if (value.length < 4) {
                        alert("Name is too short, need at least 4 characters");
                        return;
                }
                this._name = value;
        }
};

user.name = "Pete";
alert(user.name); // Pete

user.name = ""; // Name is too short...
```

Prototypal inheritance

- Prototypes are the mechanism by which JavaScript objects inherit features from one another.
- In JavaScript, all objects have a hidden [[Prototype]] property that’s either another object or null.

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

// walk is taken from the prototype chain
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (from rabbit)
```

- There are actually only two limitations:
    - The references can't go in circles. JavaScript will throw an error if we try to assign **proto** in a circle.
    - The value of **proto** can be either an object or null, other types (like primitives) are ignored.
- "this" :
    - No matter where the method is found: in an object or its prototype. In a method call, "this" is always the object before the dot "object.method()".
    - If we call obj.method(), and the method is taken from the prototype, this still references obj. So methods always work with the current object even if they are inherited.
    - In inheritance methods are shared, but the object state is not.

F.prtotype

- Setting a [[Prototype]] for objects created via a constructor function
- F.prototype only used at new F time

```
let animal = {
  eats: true
};
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype = animal;
let rabbit = new Rabbit("White Rabbit"); //  rabbit.__proto__ == animal
alert( rabbit.eats ); // true
```

Native prototypes

- The "prototype" property is widely used by the core of JavaScript itself. All built-in constructor functions use it.
- Object.prototype
    - obj = {} is the same as obj = new Object(), where Object is a built-in object constructor function, with its own prototype referencing a huge object with toString and other methods.

    ```
    let obj = {};
    alert(obj.__proto__ === Object.prototype); // true
    // obj.toString === obj.__proto__.toString ==
    Object.prototype.toString
    // obj -> Object.prototype -> null
    ```

- built-in objects : Array, Date, Function
    - Array/Date/Function -> Array/Date/Function.prototype (provides Array/Date/Function methods) -> Object.prototype (provides toString, object methods...). -> null

  - By specification, all of the built-in prototypes have Object.prototype on the top. Sometimes people say that “everything inherits from objects”.

```
// ----- Array ------
let arr = [1, 2, 3];
// it inherits from Array.prototype?
alert( arr.__proto__ === Array.prototype ); // true
// then from Object.prototype?
alert( arr.__proto__.__proto__ === Object.prototype ); // true
// and null on the top.
alert( arr.__proto__.__proto__.__proto__ ); // null
// the result of Array.prototype.toString
alert(arr); // 1,2,3

// ----- Function -------
function f() {}
// it inherits from Function.prototype?
alert(f.__proto__ == Function.prototype); // true
// then from Object.prototype?
alert(f.__proto__.__proto__ == Object.prototype); // true, inherit
from objects
```

- Primitives

  - strings, numbers and booleans are not object. But....
    - They are not objects. But if we try to access their properties, then temporary wrapper objects are created using built-in constructors String, Number, Boolean, they provide the methods and disappear.
  - null and undefined have no corresponding prototypes too

- Changing native prototypes

  - Native prototypes can be modified. For instance, if we add a method to String.prototype, it becomes available to all strings:
  - During the process of development, we may have ideas for new built-in methods we’d like to have, and we may be tempted to add them to native prototypes. But that is generally a bad idea.
  - In modern programming, there is only one case where modifying native prototypes is approved. That’s polyfilling.

```
String.prototype.show = function() {
        alert(this);
};

"BOOM!".show(); // BOOM!
```

Prototype methods, objects without proto

- Modern Prototype methods

    - Object.create(proto[, descriptors]) – creates an empty object with given proto as [[Prototype]] (can be null) and optional property descriptors.
        - Object.create provides an easy way to shallow-copy an object with all descriptors:
        - `let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));`
    - Object.getPrototypeOf(obj) – returns the [[Prototype]] of obj (same as **proto** getter).
    - Object.setPrototypeOf(obj, proto) – sets the [[Prototype]] of obj to proto (same as **proto** setter).
- The built-in **proto** getter/setter is unsafe if we'd want to put user-generated keys in to an object.
- `Object.create(null)` to create a "very plain" object without **proto**

```
let animal = {
  eats: true
};

// create a new object with animal as a prototype
let rabbit = Object.create(animal);

alert(rabbit.eats); // true
alert(Object.getPrototypeOf(rabbit) === animal); // get the prototype of
rabbit

Object.setPrototypeOf(rabbit, {}); // change the prototype of rabbit to {}
```

Class

- A class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).
- MyClass is technically a function, while methods are written to MyClass.prototype.

```
lass MyClass {
  prop = value; // field

  constructor(...) { // constructor
    // ...
  }

  method(...) {} // method

  get something(...) {} // getter method
  set something(...) {} // setter method

  [Symbol.iterator]() {} // method with computed name/symbol name
  // ...
}
```

- What is class ? : a class is a kind of a function.

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// proof: User is a function
alert(typeof User); // function

// ...or, more precisely, the constructor method
alert(User === User.prototype.constructor); // true

// The methods are in User.prototype, e.g:
alert(User.prototype.sayHi); // alert(this.name);

// there are exactly two methods in the prototype
alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi
```

- Sometimes people say that class is a “syntax sugar” in JavaScript,because we could actually declare class without class keyword.
- there are important differences
    1. a function created by class is labelled by a special internal property [[FunctionKind]]:"classConstructor". So it’s not entirely the same as creating it manually.
    2. Class methods are non-enumerable. A class definition sets enumerable flag to false for all methods in the "prototype".
    3. Classes always use strict. All code inside the class construct is automatically in strict mode.
- Class Expression

```
let User = class MyClass {
  sayHi() {
    alert(MyClass); // MyClass is visible only inside the class
  }
};

new User().sayHi(); // works, shows MyClass definition
alert(MyClass); // error, MyClass not visible outside of the class
```

Class inheritance

- To inherit from another class, we should specify "extends" and the parent class before the braces {..}.

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} stopped.`);
  }
}

// Inherit from Animal by specifying "extends Animal"
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
}

let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.hide(); // White Rabbit hides!
```

- Any expression is allowed after extends
- `class User extends f("Hello") {}`
- Overriding a method and constructor
    - Classes provide "super" keyword for that.
    - super.method(...) to call a parent method.
    - super(...) to call a parent constructor (inside our constructor only).
    - Arrow functions have no super
    - Constructors in inheriting classes must call super(...), and (!) do it before using this.

Static properties and methods

- Static method calls are made directly on the class and are not callable on instances of the class. Static methods are often used to create utility functions.
- Static methods are used for the functionality that belongs to the class as a whole.
- Static properties are used when we'd like to store class-level data, also not bound to an instance.

```
class MyClass {
  static property = ...;

  static method() {
    ...
  }
}
```

Private and protected properties and methods

- In terms of OOP, delimiting of the internal interface from the external one is called encapsulation.
- Coffee machine outside and inside, Programming objects are like coffee machines.
- JavaScript, there are three types of properties and members:
    - Public: accessible from anywhere. They comprise the external interface. Till now we were only using public properties and methods.
    - Protected: accessible only from inside the class and those extending it.
        - Protected properties are usually prefixed with an underscore _. "_waterAmount"
        - That's a well-known convention, not enforced at the language level. Programmers should only access a field starting with _ from its class and classes inheriting from it.
        - Use getter/setter syntax to setup.

    ```
    class CoffeeMachine {
            _waterAmount = 0;

            set waterAmount(value) {
                    if (value < 0) throw new Error("Negative water");
                    this._waterAmount = value;
            }

            get waterAmount() {
                    return this._waterAmount;
            }

            constructor(power) {
                    this._power = power;
            }

    }

    // create the coffee machine
    let coffeeMachine = new CoffeeMachine(100);

    // add water
    coffeeMachine.waterAmount = -10; // Error: Negative water
    ```

    - Private: accessible only from inside the class. These are for the internal interface.
        - Privates should start with #. They are only accessible from inside the class.
        - On the language level, # is a special sign that the field is private. We can't access it from outside or from inheriting classes.

    ```
    class CoffeeMachine {
            #waterLimit = 200;
    ```

```
        #checkWater(value) {
                if (value < 0) throw new Error("Negative water");
                if (value > this.#waterLimit) throw new
Error("Too much water");
        }

        _waterAmount = 0;

        set waterAmount(value) {
                this.#checkWater(value);

                this._waterAmount = value;
        }

        get waterAmount() {
                return this.waterAmount;
        }

}

let coffeeMachine = new CoffeeMachine();

coffeeMachine.#checkWater(); // Error
coffeeMachine.#waterLimit = 1000; // Error

coffeeMachine.waterAmount = 100; // Works
```

Extending built-in classes

- Built-in classes like Array, Map and others are extendable also.
- Built-in methods like filter, map and others – return new objects of exactly the inherited type. They rely on the constructor property to do so.
- `arr.constructor === PowerArray`

```
// here PowerArray inherits from the native Array:
// add one more method to it (can do more)
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // false
```

Class checking: "instanceof"

- The instanceof operator allows to check whether an object belongs to a certain class. It also takes inheritance into account.

```
class Rabbit {}
let rabbit = new Rabbit();

// is it an object of Rabbit class?
alert( rabbit instanceof Rabbit ); // true

// works with constructor functions instead of class
function Rabbit() {}
alert( new Rabbit() instanceof Rabbit ); // true

// built-in classes like Array:
let arr = [1, 2, 3];
alert( arr instanceof Array ); // true
alert( arr instanceof Object ); // true
```

Mixins

- We can only inherit from a single object. There can be only one [[Prototype]] for an object.
- A class may extend only one other class.
- Sometimes that feels limiting
- A mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes.
- A mixin provides methods that implement a certain behavior, but we do not use it alone, we use it to add the behavior to other classes.
- Mixins may become a point of conflict if they occasionally overwrite native class methods. So generally one should think well about the naming for a mixin, to minimize such possibility.

```
let sayMixin = {
  say(phrase) {
    alert(phrase);
  }
};

let sayHiMixin = {
  __proto__: sayMixin, // (or we could use Object.create to set the
prototype here)

  sayHi() {
    // call parent method
    super.say(`Hello ${this.name}`);
  },
  sayBye() {
    super.say(`Bye ${this.name}`);
  }
};

class User {
  constructor(name) {
    this.name = name;
  }
}

// copy the methods
Object.assign(User.prototype, sayHiMixin);

// now User can say hi
new User("Dude").sayHi(); // Hello Dude!
```

Error handling, "try..catch"

- The try..catch construct has two main blocks: try, and then catch:
- try..catch to work, the code must be runnable means javascript code required to be syntactically correct.
-

```

try {
 // code...
} catch (err) {
 // error handling
}

// ---- Example -----
try {
 alert('Start of try runs');  // (1) <--
 lalala; // error, variable is not defined!
 alert('End of try (never reached)');  // (2)
} catch(err) {
 alert(`Error has occurred!`); // (3) <--
   alert(err); // (4) <-- ReferenceError: lalala is not defined

   // Error object
   // message – the human-readable error message.
   // name – the string with error name (error constructor name).
   // stack (non-standard) – the stack at the moment of error creation.
   alert(err.name); // ReferenceError
 alert(err.message); // lalala is not defined
 alert(err.stack); // ReferenceError: lalala is not defined at ...
}
alert("...Then the execution continues");
```

- Real life example, If json is malformed, JSON.parse generates an error, so the script “dies”.

```
let json = "{ bad json }";

try {

  let user = JSON.parse(json); // <-- when an error occurs...
  alert( user.name ); // doesn't work

} catch (e) {
  // ...the execution jumps here
  alert( "Our apologies, the data has errors, we'll try to request it one
more time." );
  alert( e.name );
  alert( e.message );
}
```

- Throw and Rethrowing
  - JavaScript has many built-in constructors for standard errors: Error, SyntaxError, ReferenceError, TypeError and others. We can use them to create error objects as well.
  - We can also generate our own errors using the `throw` operator. Technically, the argument of throw can be anything, but usually it's an error object inheriting from the built-in `Error` class. More on extending errors in the next chapter.
  - Rethrowing is a basic pattern of error handling: a `catch` block usually expects and knows how to handle the particular error type, so it should rethrow errors it doesn't know.

```
// Creating your own error using Error constructor
let error = new Error("Things happen o_O");
alert(error.name); // Error
alert(error.message); // Things happen o_O

// Example
let json = '{ "age": 30 }'; // incomplete data
try {
  let user = JSON.parse(json);
  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name"); // throw
  }
  blabla(); // unexpected error
  alert( user.name );

} catch(e) {
  if (e.name == "SyntaxError") {
    alert( "JSON Error: " + e.message );
  } else {
    throw e; // rethrow (*)
  }
}
```

- try…catch…finally

  - The finally clause works in case of any exit from try..catch, even via the return statement: right after try..catch is done, but before the calling code gets the control.
  - The try..catch construct may have one more code clause: finally.
  - The code has two ways of execution:
    - If you answer “Yes” to “Make an error?”, then try -> catch -> finally.
    - If you say “No”, then try -> finally.
  - The finally clause is often used when code start doing something before try..catch and want to finalize it in any case of outcome.

    ```
    try {
            ... try to execute the code ...
    } catch(e) {
            ... handle errors ...
    } finally {
            ... execute always ...
    }
    ```

- Even if we don’t have try..catch, most environments allow to setup a “global” error handler to catch errors that “fall out”. In-browser that’s window.onerror.

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

Custom errors, extending Error

- Error, SyntaxError, ReferenceError, TypeError, HttpError, DbError, NotFoundError
- Our ValidationError class should inherit from the built-in Error class.
- The `instanceof` version is much better, because in the future we are going to extend `ValidationError`, make subtypes of it, like `PropertyRequiredError`.

```

// The "pseudocode" for the built-in Error class defined by JavaScript
itself
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (different names for different built-in error
classes)
    this.stack = "<nested Calls>"; // non-standard, but most environments
support it
  }
}
```

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

// The ValidationError class is very generic. Many things may go wrong.
// The property may be absent or it may be in a wrong format (like a
string value for age)
class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property);
    this.name = "PropertyRequiredError";
    this.property = property;
  }
}

// Usage of ValidationError
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new ValidationError("No field: age");
  }
  if (!user.name) {
    throw new ValidationError("No field: name");
  }
  return user;
}

// Working example with try..catch
try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // Invalid data: No property:
name
    alert(err.name); // PropertyRequiredError
    alert(err.property); // name
  } else if (err instanceof SyntaxError) {
    alert("JSON Syntax Error: " + err.message);
  } else {
    throw err; // unknown error, rethrow it
  }
}
```

Callbacks

- Many actions in JavaScript are asynchronous.

```
loadScript('/my/script.js');
// the code below loadScript doesn't wait for the script loading to finish
// ...

// Naturally, the browser probably didn’t have time to load the script.
// So the immediate call to the new function fails.
loadScript('/my/script.js'); // the script has "function newFunction()
{…}"
newFunction(); // no such function!

//CallBack to Rescure
loadScript('/my/script.js', function() {
  // the callback runs after the script is loaded
  newFunction(); // so now it works
  ...
});
```

CallBack Hell

- CallBack Hell
- From the first look, it’s a viable way of asynchronous coding. And indeed it is. For one or maybe two nested calls it looks fine.
- But for multiple asynchronous actions that become “callback hell” or “pyramid of doom.”

```
// Pyramid of Doom , callback hell
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all scripts are loaded (*)
          }
        });

      }
    })
  }
});
```

- Don't nest functions. Give them names and place them at the top level of your program
- Use **function hoisting** to your advantage to move functions 'below the fold'
- Handle **every single error** in every one of your callbacks. Use a linter like standard to help you with this.
- Create reusable functions and place them in a module to reduce the cognitive load required to understand your code.
- Splitting your code into small pieces like this also helps you handle errors, write tests, forces you to create a stable and documented public API for your code, and helps with refactoring.

Promise

- Promise for dummies
- Video with Code Example
- A promise is a special JavaScript object that links the “producing code” and the “consuming code” together.
- In terms of our analogy: this is the “subscription list”. The “producing code” takes whatever time it needs to produce the promised result, and the “promise” makes that result available to all of the subscribed code when it’s ready.

```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "singer")
});
```

- The function passed to new Promise is called the executor.
- When the promise is created, this executor function runs automatically. It contains the producing code, that should eventually produce a result.
- The resulting promise object has internal properties:
  - `state` — initially “pending”, then changes to either “fulfilled” or “rejected”,
  - `result` — an arbitrary value, initially undefined.
- When the executor finishes the job, it should call one of the functions that it gets as arguments:
  - `resolve(value)` — to indicate that the job finished successfully:
    - sets state to "fulfilled",
    - sets result to value.
  - `reject(error)` — to indicate that an error occurred:
    - sets state to "rejected",
    - sets result to error.

```
// reject
let promise = new Promise(function(resolve, reject) {
  // after 1 second signal that the job is finished with an error
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

/ resolve
let promise = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is
constructed
```

```
  // after 1 second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
});
```

- The executor should call only one resolve or one reject. The promise's state change is final.

Consumers: then, catch, finally

- The properties `state` and `result` of the Promise object are internal.
- We can't directly access them from our "consuming code".
- We can use the methods .then/.catch/.finally for that.
- Promise object serves as a link between the executor (the "producing code" or "singer") and the consuming functions (the "fans"), which will receive the result or error.
- Consuming functions can be registered (subscribed) using methods .then, .catch and .finally.
- **then**
- If `promise` is resolved receives result
- If `promise` is rejected receives error

```
promise.then(
  function(result) { /* handle a successful result */ },
  function(error) { /* handle an error */ }
);
```

- **catch**
- Use `.catch(errorHandlingFunction)` if you're interested only in errors.
- The call .catch(f) is a complete analog of .then(null, f), it's just a shorthand.
- **finally**
- finally is a good handler for performing cleanup, e.g. stopping our loading indicators, as they are not needed any more, no matter what the outcome is.

```
new Promise((resolve, reject) => {
  /* do something that takes time, and then call resolve/reject */
})
  // runs when the promise is settled, doesn't matter successfully or not
  .finally(() => stop loading indicator)
  .then(result => show result, err => show error)
```

- It's not exactly an alias of then(f,f) though. There are several important differences:

- A finally handler has no arguments. In finally we don’t know whether the promise is successful or not. That’s all right, as our task is usually to perform “general” finalizing procedures.
- A finally handler passes through results and errors to the next handler.
- `.finally(f)` is a more convenient syntax than `.then(f, f)`: no need to duplicate the function f.

Promises vs Callbacks

```
// CallBack
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for
${src}`));

  document.head.append(script);
}

// Promise object that resolves when the loading is complete.
// The outer code can add handlers using .then
// Declare Promise
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Script load error for
${src}`));

    document.head.append(script);
  });
}

// Usage
let promise =
loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodas
h.js");

promise.then(
  script => alert(`${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('One more handler to do something else!'));
```

- **Promises**

- Promises allow us to do things in the natural order. First, we run loadScript(script), and .then we write what to do with the result.
- We can call .then on a Promise as many times as we want. Each time, we're adding a new "fan", a new subscribing function, to the "subscription list".
- **Callbacks**
- We must have a callback function at our disposal when calling loadScript(script, callback). In other words, we must know what to do with the result before loadScript is called.
- There can be only one callback.

Promises chaining

- When you want sequence of asynchronous tasks to be done one after another
- Promise -> .then -> .then -> .then
- We can have multiple handlers for one promise.
  - Promise -> .then1, Promise -> .then2, Promise -> .then3 (Same Promise)

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;
```

- **example : Bigger example: fetch**
- In frontend programming promises are often used for network requests.
- We'll use the fetch method to load the information about the user from the remote server.

```
// Make a request for user.json
fetch('/article/promise-chaining/user.json')
  // Load it as json
  .then(response => response.json())
  // Make a request to GitHub
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  // Load the response as json
  .then(response => response.json())
  // Show the avatar image (githubUser.avatar_url) for 3 seconds (maybe
animate it)
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

   setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  // triggers after 3 seconds
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));

// split the code into reusable functions
function loadJson(url) {
  return fetch(url)
    .then(response => response.json());
}

function loadGithubUser(name) {
  return fetch(`https://api.github.com/users/${name}`)
    .then(response => response.json());
}

function showAvatar(githubUser) {
  return new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  });
}

// Use them:
loadJson('/article/promise-chaining/user.json')
  .then(user => loadGithubUser(user.name))
  .then(showAvatar)
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
  // ...
```

Error handling with promises

- Asynchronous actions may sometimes fail: in case of an error the corresponding promise becomes rejected.
- The final `.catch` not only catches explicit rejections, but also occasional errors in the handlers above.
- .catch handles promise rejections of all kinds: be it a reject() call, or an error thrown in a handler.
- We should place .catch exactly in places where we want to handle errors and know how to handle them. The handler should analyze errors (custom error classes help) and rethrow unknown ones.
- It's ok not to use .catch at all, if there's no way to recover from an error.
- In any case we should have the unhandledrejection event handler (for browsers, and analogs for other environments), to track unhandled errors and inform the user (and probably our server) about the them, so that our app never "just dies".

```
fetch('/') // fetch works fine now, the server responds with the HTML page
  .then(response => response.json()) // rejects: the page is HTML, not a valid json
  .catch(err => alert(err)) // SyntaxError: Unexpected token < in JSON at position 0

//Example
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  blabla(); // no such function
}).catch(alert); // ReferenceError: blabla is not defined
```

Promise API

- Promise.resolve(value) – makes a resolved promise with the given value.

```
let promise = Promise.resolve(value);
// Same as:
let promise = new Promise(resolve => resolve(value));
```

- Promise.reject(error) – makes a rejected promise with the given error.

```
let promise = Promise.reject(error);
// Same as:
let promise = new Promise((resolve, reject) => reject(error));
```

- Promise.all(promises) – waits for all promises to resolve and returns an array of their results. If any of the given promises rejects, then it becomes the error of Promise.all, and all other results are ignored.

```
let promise = Promise.all([...promises...]);
// Example:
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // 1,2,3 when promises are ready: each promise contributes
an array member
```

- Promise.race(promises) – waits for the first promise to settle, and its result/error becomes the outcome.

```
let promise = Promise.race(iterable);
// Example:
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1 as first settled promise “wins the race”, all further
results/errors are ignored.
```

- Of these four, Promise.all is the most common in practice.

Promification

- Promisification is a great approach, especially when you use async/await (see the next chapter), but not a total replacement for callbacks.
- Remember, a promise may have only one result, but a callback may technically be called many times.
- So promisification is only meant for functions that call the callback once. Further calls will be ignored.

Microtasks

- Promise handlers .then/.catch/.finally are always asynchronous.
- Why ? because of Microtask queue
  - the queue is first-in-first-out: tasks enqueued first are run first.
  - Execution of a task is initiated only when nothing else is running.
- Promise handling is always asynchronous, as all promise actions pass through the internal “promise jobs” queue, also called “microtask queue” (v8 term).
- So, .then/catch/finally handlers are called after the current code is finished.
- If we need to guarantee that a piece of code is executed after .then/catch/finally, it’s best to add it into a chained .then call.

```
let promise = Promise.resolve();
promise.then(() => alert("promise done"));
alert("code finished"); // this alert shows first
```

Event loop

- “Event loop” is a process when the engine sleeps and waits for events. When they occur – handles them and sleeps again.
- Events of Macrotask queue
  - mousemove, a user moved their mouse.
  - setTimeout handler is to be called.
  - an external `<script src="...">` is loaded, ready to be executed.
  - a network operation, e.g. fetch is complete.
- Microtask queue has a higher priority than the macrotask queue.
- Macrotasks run after the code is finished and after the microtask queue is empty.

```
setTimeout(() => alert("timeout")); // third as MicroTask
Promise.resolve()
  .then(() => alert("promise")); // second as MicroTask
alert("code"); // first
```

- "Unhandled rejection" is when a promise error is not handled at the end of the microtask queue.

```
let promise = Promise.reject(new Error("Promise Failed!"));

window.addEventListener('unhandledrejection', event => {
  alert(event.reason); // Promise Failed!
});
```

- regular code -> then promise handling(MicroTask) -> then everything else, like events (MacroTask)

Async/await

- A special syntax to work with promises in a more comfortable fashion, called “async/await”.
- `async` ensures that the function returns a promise, and wraps non-promises in it.

```
async function f() {
  return 1;
}
f().then(alert); // 1

// Equals to following
async function f() {
  return Promise.resolve(1);
}
f().then(alert); // 1
```

- **await**
- The keyword `await` makes JavaScript wait until that promise settles and returns its result.
- `await` literally makes JavaScript wait until the promise settles, and then go on with the result.
- `await` only works inside an `async` function.

  - If it’s an error, the exception is generated, same as if throw error were called at that very place.
  - Otherwise, it returns the result, so we can assign it to a value.

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });
  let result = await promise; // wait till the promise resolves (*)
  alert(result); // "done!"
}

f();
```

- With async/await we rarely need to write promise.then/catch, but we still shouldn’t forget that they are based on promises, because sometimes (e.g. in the outermost scope) we have to use these methods.
- Also Promise.all is a nice thing to wait for many tasks simultaneously.

```
// With Promise
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
    })
}

loadJson('no-such-user.json') // (3)
        .catch(alert); // Error: 404

// With async
async function loadJson(url) { // (1)
  let response = await fetch(url); // (2)
  if (response.status == 200) {
    let json = await response.json(); // (3)
    return json;
  }
  throw new Error(response.status);
}

loadJson('no-such-user.json')
  .catch(alert); // Error: 404 (4)
```

Generators

- Regular functions return only one, single value (or nothing).

- Generators can return (“yield”) multiple values, possibly an infinite number of values, one after another, on-demand.
- The main method of a generator is `next()`. When called, it resumes execution till the nearest yield `<value>` statement

```
// function* f(…) or function *f(…)
// both correct
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
let generator = generateSequence();

let one = generator.next();
alert(JSON.stringify(one)); // {value: 1, done: false}

let two = generator.next();
alert(JSON.stringify(two)); // {value: 2, done: false}

let three = generator.next();
alert(JSON.stringify(three)); // {value: 3, done: true}

//  iterable
for(let value of generator) {
  alert(value); // 1, then 2, then 3
}
// With spread operator '...'
let sequence = [0, ...generateSequence()];
alert(sequence); // 0, 1, 2, 3

//In Function
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    yield i;
  }
}
let sequence = [...generateSequence(1,5)];
alert(sequence); // 1, 2, 3, 4, 5
```

- **“yield” is a two-way road**
- generators were like “iterators on steroids”
- yield is a two-way road: it not only returns the result outside, but also can pass the value inside the generator.

```
function* gen() {
  // Pass a question to the outer code and wait for an answer
  let result = yield "2 + 2?"; // (*)
  alert(result);
}

let generator = gen();
let question = generator.next().value; // <-- yield returns the value
generator.next(4); // --> pass the result into the generator
```

- **Async iterators and generators**
- Regular iterators and generators work fine with the data that doesn't take time to generate.
- When we expect the data to come asynchronously, with delays, their async counterparts can be used, and for await..of instead of for..of.

```
// Iterators
Symbol.iterator // Object method to provide iteraterable
any value // next() return value is

// Generators
function*  // Declaration
{value:…, done: true/false}     // generator.next() returns

//-----

//Async iterators
Symbol.asyncIterator // Object method to provide iteraterable
Promise // next() return value is

// Async generators
async function* // Declaration
Promise that resolves to {value:…, done: true/false} // //
generator.next() returns
```

- In web-development we often meet streams of data, when it flows chunk-by-chunk. For instance, downloading or uploading a big file.
- We can use async generators to process such data, but it's worth to mention that there's also another API called Streams, that provides special interfaces to transform the data and to pass it from one stream to another (e.g. download from one place and immediately send elsewhere).
- Streams API not a part of JavaScript language standard. Streams and async generators complement each other, both are great ways to handle async data flows.

Modules

- As our application grows bigger, we want to split it into multiple files, so called 'modules'.
- A module usually contains a class or a library of useful functions.
- A module is just a file, a single script, as simple as that.
  - export keyword labels variables and functions that should be accessible from outside the current module.
  - import allows to import functionality from other modules.
- Modules always use strict, by default. E.g. assigning to an undeclared variable will give an error.

```
<script type="module">
  a = 5; // error
</script>
```

- Each module has its own top-level scope. In other words, top-level variables and functions from a module are not seen in other scripts.
- A module code is evaluated only the first time when imported

```
// 📁 admin.js
export let admin = { };
export function sayHi() {
  alert(`Ready to serve, ${admin.name}!`);
}

// 📁 init.js
import {admin} from './admin.js';
admin.name = "Pete";

// 📁 other.js
import {admin, sayHi} from './admin.js';
alert(admin.name); // Pete
sayHi(); // Ready to serve, Pete!
```

- **Top-level “this” is undefined**
- In a module, top-level this is undefined, as opposed to a global object in non-module scripts:

```
<script>
  alert(this); // window
</script>

<script type="module">
  alert(this); // undefined
</script>
```

- **Module scripts are deferred**
  - external module scripts `<script type="module" src="...">` don’t block HTML processing, they load in parallel with other resources.
  - module scripts wait until the HTML document is fully ready (even if they are tiny and load faster than HTML), and then run.
  - relative order of scripts is maintained: scripts that go first in the document, execute first.

```
// 3rd
// object: the script can 'see' the button below
// as modules are deferred, the script runs after the whole page is loaded
```

```
<script type="module">
  alert(typeof button);
</script>

// 2nd
// Compare to regular script below:
// Error: button is undefined, the script can't see elements below
// regular scripts run immediately, before the rest of the page is
processed
<script>
  alert(typeof button);
</script>

// 1st
<button id="button">Button</button>
```

- **async works on inline scripts**
- Async attribute `<script async type="module">` is allowed on both inline and external scripts. Async scripts run immediately when imported modules are processed, independently of other scripts or the HTML document.

```
// the script below has async, so it doesn’t wait for anyone.
// all dependencies are fetched (analytics.js), and the script runs
// doesn't wait for the document or other <script> tags
<script async type="module">
  import {counter} from './analytics.js';
  counter.count();
</script>
```

- No “bare” modules allowed and Compatibility, “nomodule”

```
// the module must have a path, e.g. './sayHi.js' or wherever the module
is
import {sayHi} from 'sayHi'; // Error, "bare" module

// nomodule
<script type="module">
  alert("Runs in modern browsers");
</script>

<script nomodule>
  alert("Modern browsers know both type=module and nomodule, so skip
this")
  alert("Old browsers ignore script with unknown type=module, but execute
this.");
</script>
```

- **Build tools**

- In real-life, browser modules are rarely used in their “raw” form. Usually, we bundle them together with a special tool such as `Webpack` and deploy to the production server.
- Why ?
  - Unreachable code removed.
  - Unused exports removed (“tree-shaking”).
  - Development-specific statements like console and debugger removed.
  - Modern, bleeding-edge JavaScript syntax may be transformed to older one with similar functionality using Babel.
  - The resulting file is minified (spaces removed, variables replaced with shorter named etc).

Export/Import

```
// 📁 say.js
function sayHi(user) {
  alert(`Hello, ${user}!`);
}
function sayBye(user) {
  alert(`Bye, ${user}!`);
}
export {sayHi, sayBye}; // a list of exported variables

// 📁 main.js
import {sayHi, sayBye} from './say.js';
sayHi('John'); // Hello, John!
sayBye('John'); // Bye, John!

// 📁 main.js with * and as
import * as say from './say.js';
say.sayHi('John');
say.sayBye('John');
```

- Named export
  - `export class User {...}`
  - `import {User} from ...`
- Default export
  - `export default class User {...}`
  - `import User from ...`
- named exports must (naturally) have a name, while export default may be anonymous.
- Please note that import/export statements don’t work if inside {...}.
- Re-export:
  - `export {x [as y], ...} from "mod"`
  - `export * from "mod"` (doesn’t re-export default).
  - `export {default [as y]} from "mod"` (re-export default).

```
import User from './user.js';
export {User};
// Re-Export
export {default as User} from './user.js';
```

- **Dynamic imports**
- First, we can’t dynamically generate any parameters of import.
- The module path must be a primitive string, can’t be a function call.
- we can’t import conditionally or at run-time:

```
import ... from getModuleName(); // Error, only from "string" is allowed
if(...) { import ...; // Error, not allowed! }
{import ...; // Error, we can't put import in any block}
```

- **The import() function**
- The `import(module)` function can be called from anywhere. It returns a promise that resolves into a module object.

```
let modulePath = prompt("Module path?");
import(modulePath)
  .then(obj => <module object>)
  .catch(err => <loading error, no such module?>)
```

Javascript Animation

- JavaScript animations should be implemented via requestAnimationFrame. That built-in method allows to setup a callback function to run when the browser will be preparing a repaint. Usually that’s very soon, but the exact time depends on the browser.

- When a page is in the background, there are no repaints at all, so the callback won’t run: the animation will be suspended and won’t consume resources. That’s great.

```
function animate({timing, draw, duration}) {

  let start = performance.now();

  requestAnimationFrame(function animate(time) {
    // timeFraction goes from 0 to 1
    let timeFraction = (time - start) / duration;
    if (timeFraction > 1) timeFraction = 1;

    // calculate the current animation state
    let progress = timing(timeFraction);
    draw(progress); // draw it

    if (timeFraction < 1) {
      requestAnimationFrame(animate);
    }

  });
}
```

- Options:
  - duration – the total animation time in ms.
  - timing – the function to calculate animation progress. Gets a time fraction from 0 to 1, returns the animation progress, usually from 0 to 1.
  - draw – the function to draw the animation.
- JavaScript animations can use any timing function. We covered a lot of examples and transformations to make them even more versatile. Unlike CSS, we are not limited to Bezier curves here.
- CSS animations use GPU while JS animations use CPU

Topics

Hoisting

- **Hoisting**
- Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.
- variable and function declarations are put into memory during the compile phase, but stays exactly where you typed it in your coding.
- The variables can be initialized and used before declared. But they cannot be used without initialization.
- all undeclared variables are global variables.
- JavaScript only hoists declarations, not initializations.
- Hoisted : Use -> Initialize (Hoisted variable is initialised with a value of undefined)

```
var x = 1; // Initialize x
console.log(x + " " + y); // '1 undefined'
var y = 2; // Initialize y
```

- Hoisted : Declare -> Use -> Initialize (Hoisted variable is initialised with a value of undefined)

```
// The following code will behave the same as the previous code:
var x = 1; // Initialize x
var y; // Declare y
console.log(x + " " + y); // '1 undefined'
y = 2; // Initialize y
```

- Avoid Hoisting pitfall : Initialize -> Use -> Declare

```
num = 6;
num + 7;
var num;
/* gives no errors as long as num is declared*/
```

- Avoid Hoisting pitfall : Declare -> Initialize -> Use

```
var x = 1; // Declare and Initialize x
console.log(x); // '1'
```

- "use strict" : strict mode can help expose undeclared variables.
- ES6 let : The interpreter throws an error if we use a constant before declaring and initialising it.

```
console.log(hoist); // Output: ReferenceError: hoist is not defined
let hoist = 'The variable has been hoisted.';
```

- ES6 const : The interpreter throws an error if we use a constant before declaring and initialising it.

```
const PI;
console.log(PI); // Ouput: SyntaxError: Missing initializer in const
declaration
PI=3.142;

function getCircumference(radius) {
        console.log(circumference)
        circumference = PI*radius*2;
        const PI = 22/7;
}

getCircumference(2) // ReferenceError: circumference is not defined
// PI was used before it was declared, which is illegal for const
variables.
```

- Function declarations are hoisted over variable declarations but not over variable assignments.

```
var double = 22;

function double(num) {
        return (num*2);
}

console.log(typeof double); // Output: number
```

```
var double;

function double(num) {
        return (num*2);
}

console.log(typeof double); // Output: function
```

Aync and Await

- Understanding Async and Await
- Async/Await Blows Promises
- Async/await is a new way to write asynchronous code. Previous options for asynchronous code are callbacks and promises.
- Async/await is, like promises, non blocking.
- Async/await makes asynchronous code look and behave a little more like synchronous code. This is where all its power lies.
- Async - declares an asynchronous function (async function someName(){...}). - Automatically transforms a regular function into a Promise. - When called async functions resolve with whatever is returned in their body. - Async functions enable the use of await.
- Await - pauses the execution of async functions. (var result = await someAsyncCall()😉. - When placed in front of a Promise call, await forces the rest of the code to wait until that Promise finishes and returns a result. - Await works only with Promises, it does not work with callbacks. - Await can only be used inside async functions.

Event delegation

- How JavaScript Event Delegation Works
- Event delegation
- Bubbling and capturing
    - event.target – the deepest element that originated the event.
    - event.currentTarget (=this) – the current element that handles the event (the one that has the handler on it)
    - event.eventPhase – the current phase (capturing=1, bubbling=3).
    - addEventListener without the 3rd argument or with the 3rd argument false.

```
el.addEventListener('click', listener, false) // listener doesn't capture
el.addEventListener('click', listener) // listener doesn't capture
```

Functional Programming

- MPJ FP Videos
- Less Code, Less Bugs, Less time.
- Functions are values, exploit them by divide in small simple function.
- Higher order function: Function taker other function as argument.
- Filter take another function as argument and process array.
- Composition : Composing them together using higher order function.
- **Map** : Higher Order Function, take callback function.

```
var names = animals.map(function(animal){
        return animal.name;
})
```

Functional vs OOP

- Functional: You pass an object to the function and do stuff
- `_.map([1, 2, 3], function(n){ return n * 2; });`
- OOP: You call function on the object and do stuff
- `_([1, 2, 3]).map(function(n){ return n * 2; });`
- In both examples `[1,2,3]` (array) is an object.

Mutability & Immutability In Javascript

- Mutable

```
let arr = [1];
let new_arr = arr.push(2);
// ------
let a = {
    foo: 'bar'
};
let b = a;
b.foo =  'test2'

console.log(b.foo); // test2
console.log(a === b) // true
```

- Immitable

```
let name = ‘Bill’;
let full_name = name.concat(‘ Gates’);
// ------
let a = {foo: "bar"};
let b = Object.assign({},a);
```

```
b.foo = "bar2"

console.log(a); // {foo: "bar"}
console.log(b);// {foo: "bar2"}
console.log(b === a) // false
```

JS Good Parts [Book]

Best Practice

- use UPPERCASE for global variable
- x +=1 instead of x++
- Use JSLint
- Use " for external strings.
- Use ' for internal strings and characters.
- Convert a number to a string - Use number's method (toString) - Use String function

```
str = num.toString();
str = String(num);
```

- Convert a string to a number - Use the Number function. - Use the + prefix operator. - Use the parseInt function.

```
                num = Number(str);
                num = +str;
```

- Declare all variables at the top of the function.
- Declare all functions before you call them.
- Return statement
  - If there is no expression, then the return value is undefined.
  - Except for constructors, whose default return value is this.

```
return expression; or return;
```

- Tennent's Principle of Correspondence

```
expression
(function () {
        return expression;
}())
```

Good Part

- Prototype
- Objects
- Only one number type
- Array : No need to provide a length or type when creating an array.
- Use array.splice(number,1) than delete arry[number] for deleting arry element.
- Use objects when the names are arbitrary strings.
- Use arrays when the names are sequential integers.
- Falsy values : false, null, undefined, "" (empty string), 0, NaN
- JS is loosely typed language
- It is always better to use === and !==, which do not do type coercion.

Bad Part

- NaN : Not a Number , NaN is not equal to anything, including NaN - NaN === NaN is false - NaN !== NaN is true -Equal and not equal : These operators can do type coercion