Part1 - Best Practices in JavaScript & Typescript

Good parts and what to avoid, new features, performance tips etc.
Collective knowledge - easy to read, solid understanding how to use TS

Clean Architecture: Patterns, Practices, and Principles

Azat Satklycov

GitHub Repo: https://github.com/azatsatklichov/nodejs-app (See all in-deep examples – JS, TS and Node.JS related)

Agenda

- ❑ Pyramid of Clean Code
- ❑ JS History and Ecosystem
- ❑ Pain Points Caused by JS Sloppy Architecture
- ❑ Best Practices in JS/TS
- ❑ Types of Testing - Unit Testing
- ❑ Refactoring Practices
- ❑ Code Review – DoD
- ❑ Why to use VSCode?

Medium blog: [Good to Know Parts When Coding with JavaScript]
Examples: src-js-deep/z-examples-for-blog.js

Pyramid of Clean Code

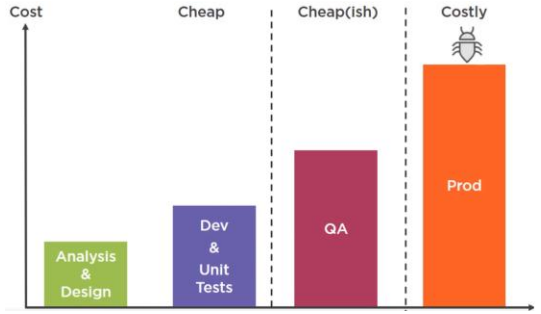
What is technical (design/code) debt?

Priority over Quality for long periods (speedy delivery – business pressure),

Poor skills, **Workarounds**, **Deferred refactoring**, **Lack of alignment to standards** etc.

To keep compromise between **Time to shift** and **Technical debt**, simply apply clean code principles.

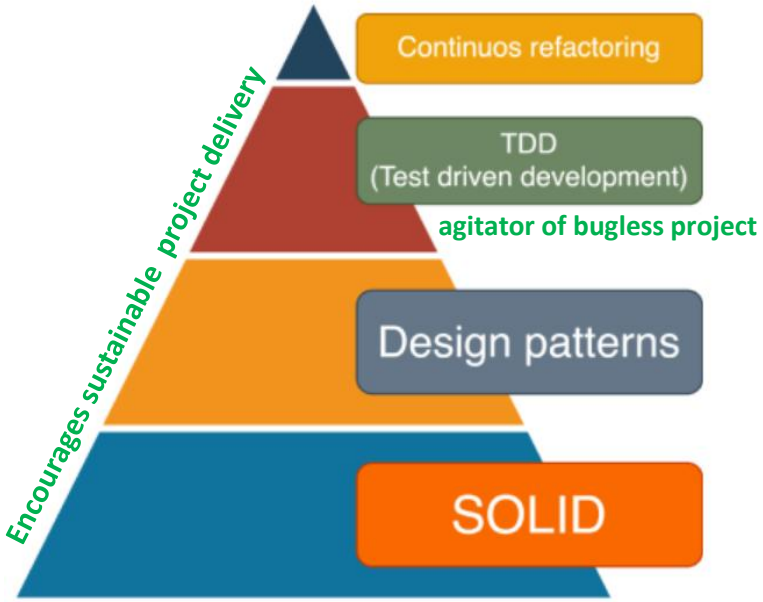
> “Errors should be detected as soon as possible […] Ideally at compile time ” Joshua Bloch, Effective Java



What is a Good Software?

Reliable, Efficient, Maintainable, Usable

Achievable by applying clean-code principles with right tooling support



JS History

JS Engines:
V8 from Google
SpiderMonkey by Mozilla
JavaScriptCore from Apple
Chakra by Microsoft
Tamarin from Adobe Flash.
Rhino & Nashorn by Oracle Java

2020	Chrome	Edge/IE	Firefox	Safari	Opera
October	80.4 %	5.2 %	7.1 %	3.7 %	2.1 %

Year	Chrome	IE	Firefox	Safari	Opera
2015	63.3 %	6.5 %	21.6 %	4.9 %	2.5 %
2009	6.5 %	39.4 %	47.9 %	3.3 %	2.1 %
2008		52.4 %	42.6 %	2.5 %	1.9 %
2007		58.5 %	35.9 %	1.5 %	1.9 %
				Netscape	
2006		62.4 %	27.8 %	0.4 %	1.4 %
2005		73.8 %	22.4 %	0.5 %	1.2 %
			Mozilla		
2004		80.4 %	12.6 %	2.2 %	1.6 %
2003		87.2 %	5.7 %	2.7 %	1.7 %
2002		84.5 %	3.5 %	7.3 %	

Languages for Web: HTML/CSS, **JavaScript**, Java applets, Flash/Flex

The Original JavaScript ES1 ES2 ES3 (1997-1999)
1993 Mosaic - first browser
1994 - Netscape Navigator
1995 – Mocha -> Live Script(JS) -> JavaScript (Brendan Eich, 10 days dev.)
1996 **Browser war** started: Microsoft JScript for IE browser
1997 – ECMAScript (ES specification): JavaScript, JScript, ActionScript
ECMA-262 is the **official name** of the standard. ECMAScript is official name
ECMAScript versions abbreviated to ES1, ES2, ES3, ES5, and ES2015 a.k.a **ES6**.
Since 2016, version names (ECMAScript 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024).

1999|2005 – **AJAX born** (E4X)
2008 – ES4 (Yahoo, Google, Microsoft, ..) -> ES6

The **First Main Revision** ES5 (2009)
2009 – **CommonJS**: **Browserless JS**, way to **Node.js – JS Runtime** (Ryan Dahl).
Second Revision 2015-2018 versions of JavaScript (ES6 - ECMAScript 2015)

2015–2020 Node.js Era

2022 - Internet Explorer retired on 15.6.2022 after 25+ service years

2020–2022 Deno-JS/TS&WebAssebly start next-gen Runtime Era (same author)

JS Ecosystem (What is JavaScript Used For?)

JS Alternatives: in writing code / transpilers: DART, **Typescript**, Purescript, AtScript, Elm, ClojureScript, Kaffeine, CoffeeScript, BottomLine, Opal, Roy, …

Libraries/Frameworks/Templates: jQuery, MooTools, D3.js, ExtJS (writing JS code in Java, eq:Rhyno, etc), Angular, Vue, React, Svelte, etc., Lodash, Async.js, Math.js, Nodemon, Sharp, Axios, Mustache, Handlebars, etc.

Testing (before Java used): Unit.js, Jasmine/Jest, Mocha, Tape, Chai, Selenium, Protractor, Cypress, Playwright, **ViTest**, ForSlash-Harness etc.

JS Engines: SpiderMonkey by Mozilla, JavaScriptCore from Apple, Chakra by Microsoft Tamarin from Adobe Flash, Rhino & Nashorn by Oracle Java, GraalJS from Oracle, V8 from Google,
Libuv focuses on asynchronous I/O in Node.js. In **Deno**, Tokio was introduced in place of libuv as the async tool

Build Tools and Package Managers: Grunt, Gulp. Npm, Yarn are package managers.
Desktop, Mobile, Game Development: VSCode Extensions, Electron, Titanium, Pixi.js, GameQuery, etc.

Machine Learning: TensorFlow, Brain.js, Langchain with Typescript
Security, Authentication and Authorization: Helmet, Hashes, JWT and Sessions, OAuth, Passport, etc.
Linting, Debugging, and Logging: ESLint, JSLint, TSLint, winston, log4js, Pino etc.
APIs and Documenting: AG Grid, Cloudinary, Twilio, JSHint, JSDoc, Swagger etc.
OS and Runtime Environments: Node.js, Node.OS, and Deno

**console,log(arguments) (browser vs Node.JS.
in Node.js (module, requires,..) but not in browser**

Alongside client and server software, machine learning, it is now even possible to write native mobile apps using JavaScript.
Also dedicated one language for FE&BE&DB – kind of full stack

Pain Points Caused by JS Sloppy Architecture

JS is an Error Tolerant Language

You can still make coding problems easily in your JS apps because of its **sloppy architecture** (broken dynamic types, object wrappers, coercion, browser-support, IEEE 754, etc.).
JavaScript has an **elegance and beauty inside** if you follow what you need to avoid when coding with it.

```
console.log("3" - "2");//1
console.log("3" + "2");//32
```

- Avoid Global Variables: Global variables and functions can be overwritten by other scripts
- Beware of Automatic Type Conversions: "3"+"2"=32, "3"-"1"=2 Confusing Addition & Concatenation
- Use Parameter Defaults: otherwise undefined if not provided
- Avoid using eval() : run text as code, security issue, like bat or shell files
- document.write loads the page, you lost all - so be CAREFUL
- Misunderstanding Floats: var y = 0.2; var z = x + y // result in z will not be 0.3;
- Avoid creating objects via **new** on Number, String, Boolean, Objects, Array, ..
- Accessing Arrays with Named Indexes: WoW [] converted to {} but no warning
  properties will produce undefined or incorrect results:
- Undefined is Not Null
- Be ready to expect weird results
- Reduce Activity in Loops
- Reduce DOM Size
- Avoid Using with
- Reduce DOM Access: Accessing the HTML DOM is slow
- Browser compatibility issues, e.g. css properties, tags, etc…
- …
- Use style guides, and best practices , common mistakes, ..
- Some Problematic parts resolvable by: ES5/6, TS, JSLint/EsLint, FlowType, new libraries like lodash ,.. But code is open to do basic errors

```
var x ="Hello";
var y = new String( value: "Hello");
(x === y)// false,  x is a string and y is an object.

//Even worse
var x = new String( value: "Hello");
var y = new String( value: "Hello");
(x == y)// is false because you cannot compare objects.
```

```
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
var x = person.length;      // person.length will return 0
var y = person[0];          // person[0] will return undefined
```

WARNING !!

Internet Explorer 8 will crash.

JSON does not allow trailing commas.

```
person = {firstName:"John", lastName:"Doe", age:46,}
points = [40, 100, 1, 5, 25, 10,];
```

Ending Definitions with a Comma
Trailing commas in object and array definition are legal in ES 5.

JavaScript Data, Object types:

Primitive Data Types (stored on stack, by default has no properties or methods unless calling from wrappers):
string, number, bigint, boolean, symbols(es6), null, undefined. Primitive values are **immutable**

Complex Data Types
A complex data type can store multiple values and/or different data types together. JavaScript has one complex data type:
object: All other complex types like arrays, functions, sets, and maps are just different types of objects.
Types of Objects: [type object] Object, Date, Array, String, Number, Bigint, Symbol, Boolean, Function[function].
The typeof operator returns only two types: object, function. Also, see Wrapper.prototype – which method available

The object data type can contain both **built-in objects**, and **user defined** objects:
Built-in object types can be: objects, arrays, dates, maps, sets, int arrays, float arrays, promises, and more.

Types contain no values: null, undefined [typeof null is **object**], [typeof undefined is **undefined**]

```
Datatype examples:
"";           // primitive string
0;            // primitive number
false;        // primitive boolean

{};           // object object
[];           // array object
/()/          // regexp object
function(){}; // function
```

Calling methods from a wrapper on a primitive type can be done directly on the primitive itself or on a variable pointing to the primitive.

```
let someVariable = 'string'
someVariable.at(1) // 't'
'string'.at(2) // 'r‘
//to apply Number methods, wrap with bracket or two dots
(5).toString() // '5'
5..toString() // '5‘
```

b-data-types.html,
b-data-types2.html

Variable Scopes

Scope determines the accessibility (visibility) of variables. JavaScript variables have 3 types of scope:

- Function scope
- Global scope
- Block scope - ES6 (2015) introduced the two new JS keywords let and const to provide **Block Scope**

Local Scope

Variables declared within a JavaScript function, are **LOCAL** to the function:
Local variables have **Function Scope**: They can only be accessed from within the function.

They all have **Function Scope**:

```
function myFunction() {
  var carName = "Volvo";   // Function Scope
}
function myFunction() {
  let carName = "Volvo";   // Function Scope
}
function myFunction() {
  const carName = "Volvo";   // Function Scope
}
```

Global Scope

Variables declared **Globally** (outside any function) have **Global Scope**. **Global** variables can be accessed from anywhere in a JavaScript program. Variables declared with `var`, `let` and `const` are quite similar when declared outside a block.

Variables are Containers for Storing Data

JavaScript Variables can be declared in 4 ways:

- Automatically e.g. carName = "Volvo"; //automatic via VAR declaration
- Using var
- Using let (scoped but can be re-assigned)
- Using const (scoped, means less cleanup, more efficient memory utilization)

var keyword was used in all JS code from 1995 to 2015.

d-variable-scopes.html

Global (Scoped) Variables should be avoided

- Visible in every scope, can significantly complicate the behavior of the program, harder to run subprograms
- Can be **re-declared**, re-assigned, … can lead to conflicts and unintended side effects, causes security issue

Three ways to define global variables

- **var foo = value;** (any place outside the function)
- **window.foo = value;** (add directly to global object)
- **foo = value;** (implied global or implicit global via variable hoisting) more open to buggy programs

JS has function SCOPE: Each function creates new scope. Variables declared in function, become **LOCAL** to the function.
Undefined Variables and Functions: If a variable is not explicitly declared, then JS assumes that the variable was global

E.g.1

```
function foo() {
    var variable1, variable2;
    variable1 = 5; varaible2 = 6; //????
    return variable1 + varaible2;
}
console.log(variable1);//ReferenceError: variable1 is not defined
console.log(varaible2);
```

E.g.2

```
var carName = "Ford";
console.log(carName);
var carName; //still has value  Ford
console.log(carName);
var carName = 76; //type changed (terrible)
console.log(carName);
```

E.g.3

```
for (var i = 1; i <= 10; i++) {
  // Block Scope
}
console.log('i = '+i) //11

for (let j = 1; j <= 10;j++) {
  // Block Scope
}
console.log('j = '+j) //j is not defined
```

Alternative:

- To minimize globalism use modules as global-var: **var myApp = {};** which holds all fields for your app. e.g. **Angular style…**
- Use **let|const** (scoped variable) starting from ES5 or Typescript .
- Use **jslint/eslint**: helps to catch variable-hoisting bugs, to find if you use variable out of scope
- Or closure for data hiding

```
{
  var x = 2;//automatic
}
// x CAN be used here
```

```
var x = 10;
// x is 10
{
var x = 2;
// x is 2
}
// x is 2
```

```
{
  let x = 2;
}
// x can NOT be used here, cannot be accessed from outside the block
```

```
let x = 10;
// x is 10
{
let x = 2;
// x is 2
}
// x is 10
```

```
var x = "JS-value";
// 'If you re-declare a JavaScript variable, it
will not loose its value
//in JAVA it gives duplicate field error
var x; //but not for x=6
console.log('x = '+x);// x = JS-value
```

Use: /*eslint no-redeclare: "error"*/

```
let x = "John Doe";
let x = 0; // Cannot re-declare block-scoped variable 'x'
//let can not be redeclared, But can be re-assigned  BE CAREFUL or use Typescript
let xx = 32; //"John Doe";
xx = "sd34.40"; //xx = 776
console.log(xx) //sd34.40
```

```
const cars = ["Saab", "Volvo", "BMW"];
cars = ["Toyota Corolla", "Volvo", "Audi"];    // ERROR
// But you can change/modify/mutate an element (or inside object)
cars[0] = "Toyota RAV4"; //JS Objects are MUTABLE by default
```

d-vars.js,
d-global-vars.js,
b-undefined_vs_null.js

Block Scoping with Logical Statements

If-else statements by default have block scopes once {} used. Same can be done for switch statements (not by default)

```
let x = 5
switch(x) {
 case 5:
    let x = 6
    console.log("hello")
    break
 case 6:
    let x = 5 //Cannot redeclare block-scoped variable 'x'.ts(2451)
    console.log("goodbye")
    break
}
```

```
let x = 5
switch(x) {
 case 5: {
    let x = 6
    console.log("hello")
    break
 }
 case 6:{
    let x = 5
    console.log("goodbye")
    break
 }
}
```

Difference Between var, let and const

	Scope	Redeclare	Reassign	Hoisted	Binds this
var	No	Yes	Yes	Yes	Yes
let	Yes	No	Yes	No	No
const	Yes	No	No	No	No

When to Use var, let, or const?

- Always use `const` if the value should not be changed
- Only use `let` if you can't use `const`
- Only use `var` if you MUST support old browsers

What is Not Good?

`var` does not have to be declared. AUTO
`var` is hoisted.
`var` binds to this.

Browser Support

The `let` and `const` keywords **are not supported in Internet Explorer 11 or earlier**.

d-global-vars-hoisting.js, z-modern-js.js

HOISTING

JS declarations are **Hoisted**. Variables can be used before declared. By default JS moves all **declarations** to the top of the scope

```
x = 5; // Assign 5 to x
elem = document.getElementById("demo");
elem.innerHTML = x;    // 5
var x; // Declare x
```

```
var x; // Declare x
x = 5; // Assign 5 to x
elem = document.getElementById("demo");
elem.innerHTML = x;                    // 5
```

Variables defined with `let` and `const` are hoisted to the top of the block, but not *initialized*. Using a `let` variable before it is declared will result in a `ReferenceError`. Using a const variable before it is declared, is a syntax error, so the code will simply not run.

```
carName = "Volvo";
let carName;
```

JavaScript Hoisting
With **let**, you cannot use a variable before it is declared.
ReferenceError: Cannot access 'carName' before initialization

JavaScript Initializations are Not Hoisted

```
var x = 5; // Initialize x
var y = 7; // Initialize y

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // 5 7
```

JavaScript only hoists declarations, not initializations.

```
var x = 5; // Initialize x
elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // x is 5 and y is undefined
var y = 7; // Initialize y,
```

only the declaration (var y), not the initialization (=7) is hoisted to the top.

```
var x = 5; // Initialize x
var y;     // Declare y
elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // 5 undefined
y = 7;     // Assign 7 to y
```

Declare Your Variables At the Top !

- ✓ To avoid bugs, always declare all variables at the beginning of every scope.
- ✓ JavaScript in **strict mode** does not allow variables to be used if they are not declared.

The function Statement, function Expression - function statements (not expressions, not arrow f.) are subject to hoisting.

function foo( ) {} `//function statement`

var foo = function foo( ) {};`//function expression`

h-hoisting.html, l-function-types-IIFE-hoisting.html

How to Define a JavaScript Object

Objects are containers (resizable) for **Properties** and **Methods**. An object literal is a list of **name(key):value** pairs inside curly braces **{}**.
The **named values**, in JS objects, are called **properties**. Keys can be number, string or symbols. Values can be any type – strings, functions, etc.

- Using an Object Literal - also called **object initializers**.

```
const person = {firstName:"John", lastName:"Doe", age:50,yeColor:"blue"}
```

- Using the new Keyword

```
// Create an Object
const person = new Object();
// Add Properties
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

- Using Object methods-see next page

- Using an Object Constructor

You can delete object properties,
Not possible in JAVA

```
// Constructor Function for Person objects
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
} // Create a Person object
const person = new Person("John", "Doe", 50, "blue");
delete person.age; // Delete a object property
console.log(person3.age) // undefined
```

How to access object properties -
Dot Notation vs. Square Brackets

Use quotation marks when accessing with square Brackets, but be careful same is not true when accessing Using .Dot notation.

```
let myObject = {
      "key": "value",
      "someKey": 5,
      "anotherKey" : true
}
console.log(myObject.key) //  "value“
console.log(myObject["key"]) //  "value“

let keyName = "key"
console.log(myObject.keyName) //   undefined
console.log(myObject[keyName]) //  "value"
```

Note: JavaScript objects are mutable. Any changes to a copy of an object will also change the original. See '**Object Mutability**' section

j-objects.js, j-object-cteation.html

Using Wrappers (without new keyword) to Create Types

Wrappers can be used to create new data of a certain type.
While calling Number() and String() using "without new keyword" it creates a new primitive, calling it as a classic constructor will lead to some unexpected behavior.

```
let newString = String("hello") // "hello"
let objString = String({ "some" : "object" }) // "[object Object]"
let newString = String(5) // "5"
let newNumber = Number("5") // 5

let newString = new String("hello") // String { "hello" }
let newNumber = new Number(5) // Number { 5 }
```

new - Creates a new object that inherits from the operand's prototype member, and then calls the operand, binding the new object to this.
Avoid creating Strings, Numbers, Booleans, Objects, & Arrays as Objects – simply avoid using new!
Unexpected **results** and **slow down exec-speed. Two objects can't be even compared**. There is no compile-time or runtime warning.

```
var a = new String('b');       //String object
var b = new Number(10);        //Number object
var c = new Boolean(true);     //Boolean object
var d = new Object();          //Object object
var e = new Array();           //Array object
```

```
let check  = new Boolean (false);
if(check) { //reason, not falsy
    console.log("WOW FOUND, even FALSE");//FOUND, even FALSE
}

let check2  = false;
if(check2) {
    console.log("WOW FOUND, even FALSE");//skipped
}
```

j-constructors.html
j-wrappers.js
k-display-getters-setters.html

```
//Another WEIRD thing with OBJECTs, printed differently in console
var x = new String("JavaScript");  // x is an object
console.log(x); // [String: 'JavaScript']  //ARRAR [] ;)
var  xx = {c: "s"};//{ c: 's' }            //OBJECT {} ;)
//below printed with { }
const person = new Person("John", "Doe", 50, "blue");
```

Better Use

Use primitive types when working with variables
Use object literals `{}` instead of `new Object()`.
Use array literals `[]` instead of `new Array()`.
Use pattern literals `/()/` instead of `new RegExp()`.
Use function expressions `() {}` instead of `new Function()`

- Using `Object.assign()`
- Using `Object.create()`
- Using `Object.fromEntries()`

```
Object.assign(target, source) // Copies properties from a source object to a target object
Object.create(object) // Creates an object from an existing object
Object.fromEntries() // Creates an object from a list of keys/values
Object.keys(object) // Returns an array of the keys of an object
Object.values(object) // Returns an array of the property values of an object
Object.groupBy(object, callback) // Groups object elements according to a function
```

See more on: general object methods

```
//Use `Object.assign()` to clone or merge objects.
const target = { a: 1 };
const source = { b: 2 };
const merged = Object.assign(target, source);
console.log(merged); // { a: 1, b: 2 }
```

Deep Clone Objects

Deep cloning ensures that nested objects are copied by value, not by reference, preventing unintended modifications to the original object.
Use structured cloning or libraries like Lodash to deep clone objects.
Or use '**deepmerge**' NodeJS library to deeply merge .

```
const obj = { a: 1, b: { c: 2 } };
const deepClone = JSON.parse(JSON.stringify(obj));
console.log(deepClone); // { a: 1, b: { c: 2 } }
```

Object Property Shorthand

Assigning variables to object properties can be repetitive.
Use property shorthand to simplify object creation

```
const name = 'Jane';
const age = 25;
const user = { name, age };  //{ name: 'Jane', age: 25 }
console.log(user.name);
console.log(user["name"]);
```

Dynamic Property name `//later detailed`

Creating objects with dynamic property names can be verbose. Use computed property names to dynamically create object properties.

```
const propName = 'age';
const user = { name: 'Yul', [propName]: 25 };
console.log(user); // { name: 'Yul', age: 25 }
```

JS objects inherit members from the prototype chain so they are *never truly empty*.
To test for membership without prototype chain involvement, use the hasOwnProperty() method or limit your results.
When using a *for in* loop, usually a good idea to use **hasOwnProperty(variable)** to **make sure the property belongs to the object you want** and is not instead an inherited property from the prototype chain:

```
for (myvariable in object) {
 if
(object.hasOwnProperty(myvariable)) {
    ... //statements to be executed
    }
}
```

hasOwnProperty - can be replaced with other function

```
var name;
another_stooge.hasOwnProperty = null;       // trouble
for (name in another_stooge) {
    if (another_stooge.hasOwnProperty(name)) { // boom
        document.writeln(name + ': ' + another_stooge[name]);
    }
}
```

Object.hasOwn()[new in ES2022] is replacement for Object.prototype.hasOwnProperty()

this - keyword refers to the object it belongs to. It has different values depending on where it is used (also: sloppy mode or strict mode):

- In a method, `this` refers to the **owner object**. (when used in an object, this refers to the object)
- Alone, `this` refers to the **global object,** [object Window].
- Also, in "**use strict**" `this` refers to the **global object** [object Window].
- In a function, `this` refers to the **global object**. [object Window].
- In a function, in strict mode, `this` is `undefined`.
- In an event, `this` refers to the **element** that received the event.
- Methods like `call()`, and `apply()` can refer `this` to **any object**.
- In short, with arrow functions there are no binding of `this`.
- Arrow functionality with `this` - they do not have their own context, inherit from parent

l--objects-this.html
l-objects-this.js
l-function-this.js
l-this-using-strict.html

In general, strict mode is a more reliable way to write code.

Nullish Values

JavaScript null vs undefined

You can test if an **object exists** by testing if the type is undefined:

```
if (typeof myObj === "undefined")
```

Undefined refers to something which has not yet been assigned a value (yet).
Null refers to something which definitively has no value. In that case, just return a null.
Nullish values are always falsy, but they are not interchangeable, e.g. see below

```
let  myObj = {}
value = myObj["age"]; //undefined
if (value == null) {
  console.log("I am null"); //I am null
}
if (value == undefined) {
  console.log("I am undefined"); //I am undefined
}
if (value === null) {
  console.log("I am null"); //skipped
}
if (value === undefined) {
  console.log("I am undefined"); //I am undefined
}
```

null and undefined differences

```
typeof undefined //undefined
typeof null      // object
null == undefined // true
null === undefined // false
```

Note:

In JS, null is a primitive value.
However, typeof returns "object".
This is a well-known bug in JavaScript and has historical reasons.

2) Also different browsers seem to be returning these differently.
So, ensure this check if object is not null

```
if (typeof myObj === "object" && myObj !== null)
```

Undefined is Not Null

- ✓ JavaScript objects, variables, properties, and methods can be undefined.
- ✓ In addition, empty JavaScript objects can have the value null.
  This can make it a little bit difficult to test if an object is empty.
- ✓ You can test if an object exists by testing if the type is undefined:

- **Null represents a value which is unambiguously absent.**
- **null and undefined are distinct types.**
- **Handling null/undefined safely prevents run-time errors**

b-undefined_vs_null.js, b-data-types2.html

Phony Arrays - Arrays are special type of object used to store one-dimensional data (1D). JavaScript does not have real arrays. No need to give them a dimension [can't do **[][]** definition], and they never generate out-of-bounds errors. But their performance can be considerably worse (slower) than real arrays.

The **typeof operator does not distinguish between arrays and objects**.
var fruits = ["Banana", "Orange", "Apple", "Mango"];
typeof fruits; //object, how can I recognize ARRAY?

```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
var x = person.length;       // person.length will return 3
var y = person[0];           // person[0] will return "John"
```

1) isArray function or instanceof operator returns true when used on an array
2)
```
if (my_value && typeof my_value === 'object'
            && typeof my_value.length === 'number'
          && !(my_value.propertyIsEnumerable('length'))) { //my_value is
definitely an array!
}
```

```
function isArray(myArray) {
        return
myArray.constructor.toString().
indexOf("Array") > -1;
    }
If(fruits instanceof Array)
```

Array example:
```
var elms = new Array(40,100);//Creates an array with two elements (40 and 100)
var elms = new Array(40);// Creates an array with 40 undefined elements!!!!!
```

re-defining arrays with named-indexes

k-arrays.html

Be careful when you use [] 'array literal' to define arrays, you can easily mislead once you use '**named-indexes**', it will be converted to real **JavaScript object {}**

```
var person = [] //array
person[0] = "Meret"
person[1] = "63"
console.log(person.length+";
"+person[0]);//2; Meret
```

```
var person = [] //re-define array(becomes {}) with named indexes
person["name"] = "Maral"
person["age"] = "63"
console.log(person.length+"; "+person[0]);//0; undefined
console.log(Object.keys(person).length+"; "+person["age"]);//2; 63
```

Semicolon insertion When inserting semicolons in places where they are not welcome, returns **undefined ;)**

```
return
{
status: true
}; //returns undefined
//JS understand this {} as a NEW scope
```

```
return {
status: true
};  //returns { status: true }
```

See: c-semicolon.js

void - In JavaScript, void is an operator (but in many languages, **void is a return type**) that takes an operand and returns undefined. This is not useful, and it is very confusing. Avoid void.

```
console.log(void 0); //undefined
console.log(undefined); //undefined
```


```
// a naughty global variable from another script
var undefined = "oops";
```

ES5 improved this by making undefined immutable

void - introduced to handle the shadowing of undefined in **pre-ES5,** where undefined on the global scope can be **overridden** or **shadowed**.

```
1) let i = void 2; // i === undefined

2) <a href="javascript:void(0)">Click Here</a>

3) void function what() {
    console.log('What')
   }();
   //using void for IIFE will always evaluate to undefined.

4) let tt = (function() {})() === void 0; //undefined
   console.log(tt); //true
```

c-void-demo.js
c-void-url.html

VOID method returns UNDEFINED. //In Java VOID method does not return anything, …

Typeof

Returns a string that identifies the type of a value, BUT there is **no type such as**: **S**trings, **N**umbers, **B**ooleans, **O**bject, **A**rray, **D**ate they all have type “object”, but “function” has function type.

```
console.log( typeof 0 ); // "number“
console.log( typeof Number() ); // “number”, no new operator
console.log( typeof new Number() ); // "object"
console.log( typeof Infinity ); //"number"
console.log( typeof NaN ); // "number"
console.log( typeof 'Hi' ); // "string"
console.log( typeof new String("Hi") ); // "Object"
console.log( typeof true ); // "boolean"
console.log( typeof new Boolean(true) ); //"object"
console.log( typeof null ); // "object“,  not null ???
console.log( typeof undefined ); //"undefined"
//console.log( typeof new Symbol() ); // "symbol" - only in ES6
console.log( typeof [] ); //"object“ - arrays are objects.
console.log( typeof [1,2,3]); // "object", not "array"
console.log( typeof new Array(1,2,3)); // "object"
console.log( typeof new Date() ); //"object"
console.log( typeof /^$/ ); // "object“, //???
console.log( typeof new RegExp('^$')); //"object"
console.log( typeof {a: "abc"} ); //"object"
console.log( typeof typeof {} ); //"string"
console.log(typeof function () {}); //function
```

```
console.log(typeof Number('1') === 'number');//true
console.log(typeof Number('foo') === 'number');//true
console.log(typeof NaN === 'number'); //true
```

It is quite natural that we want to use **the ‘typeof’ operator to check if the value type is object or primitive**. BUT, unfortunately ‘typeof’ operator can not even distinguish between ‘objects’ and ‘null’, because ‘null’ belongs to ‘Falsy’ value.

```
var myObj = ['63']

if (myObj && typeof myObj === 'object') {

    console.log('I am an object');

}

myObj = null

if (myObj &&  typeof myObj === 'object') {

    console.log('I am an object');

}
```

d-typeof-constructor-instanceof.js,
b-data-types2.html

You **cannot use typeof to determine** if a JS object is an array (or a date). Use below ..

```
var myArr = [1,2,3,4];
console.log(myArr.constructor.toString().indexOf("Array") > -1 );//1-way
console.log(myArr.constructor === Array );//2-way
3-way, … or use instanceof operator which returns true when used on an array
```

The constructor property returns the constructor function for all JavaScript variables.

Instanceof checks if an object is an instance of a **class or constructor**.
instance of only works for **complex data type**

```
//array is complex type, even though [] has typeof Object
console.log(typeof []); //object
console.log([] instanceof Array); // true
```

```
const a = "I'm a string primitive";
const b = new String("I'm a String Object");
console.log(a instanceof String); //returns false
console.log(b instanceof String); //returns true
```

Wrapper objects are not fully functional like in JAVA

E.g. shows clarity on the differences between **typeof** and **instanceof**

```
const isString = (str) => {
  return typeof str === 'string' || str instanceof String
}
```


```
console.log(isString(a))//true
console.log(isString(b))//true
```

f-typeof-constructor-instanceof-falsy.js

Falsy Values - These values (in table) are all **falsy | falsey**, but they are not interchangeable.

All other values are ***truthy*** including all objects & the string 'false'.

Value	Type
0	Number
NaN (not a number)	Number
'' (empty string)	String
false	Boolean
null	Object
undefined	Undefined

```
//no need to check if value is empty etc.
if (my_value) {
    //then my value is definitely not empty (not falsy)
}

//or, to identify if it is an object
if (my_value && typeof my_value === 'object') {
    //then my value is definitely an object or an array
    //and truthy (first statement)
}
```

All objects are *truthy* and null is *falsy* (but null is an object)

```
let check  = new Boolean (false); //has FALSE value
if(check) {
    console.log("WOW FOUND, even FALSE");//FOUND, even FALSE
}

let check2  = false;
if(check2) {
    console.log("WOW FOUND, even FALSE");//skipped
}
```

f-typeof-constructor-instanceof-falsy.js

Comparing Object Equality

e-equals-values.js
e-equals2.js

Comparing two JavaScript objects will **always** return `false`.

1. looseEqual **==** (equal values) 2. **strictEqual ===** (both type and value, but still not complete) 3. **Object.is()**

```
//result is same with number/Number, boolean/Boolean, array/Array
var x = "JavaScript";
var y = new String("JavaScript");
x == y  //true, equal values
x === y  //false, have different types (string and object)


//Even worse: Objects cannot be compared:
var x = new String("JavaScript");
var y = new String("JavaScript");
// (x == y) is false because x and y are different objects
// (x === y) is false because x and y are different objects

// just if you want to compare Object with itself
Object.is(x, x)); //true
Object.is(x, y)); //false
```

== operator always converts (to matching types) before comparison.
=== operator forces comparison of values and type:

```
5=='5' //true
5==="5" //false

0 == "";      // true
1 == "1";     // true
1 == true;    // true

0 === "";     // false
1 === "1";    // false
1 === true;   // false
```

So how do you compare object equality for objects with different references in JavaScript? Well, it's not easy, and there is no built-in way to do it.
Instead, you have the following options:

e-equals-values.js, e-equals2.js, e-objects-equality.html

Alternatives

a) You can write your own function to compare each key and value in an object individually or use libraries

b) Read properties and compare them manually. Eg. `object1.name === object2.name`
c) You also can use Object.keys() [shallow(for primitives) or deep versions] just to fasten the process.
d) ES6 way Object.entries(k1).toString() === Object.entries(k2).toString(); No for nested obj+keys, key order matters
e) Use **LODASH** utility methods
f) Or you can use Node.js built-in assert.deepStrictEqual(obj1, obj2) function to compare two objects.

g) Finally, although it's not recommended, you can also convert both objects to strings using JSON.stringify() to compare their values.

In JAVA .. Just equals() checks value equality, == if both references to same object in heap memory

Equality with Switch Statements

(it requires **strictEqual** **===** comparison)

Types are inferred in Javascript, so this can be confusing, e.g. both 5 and '5' has same value but different types

```
let x = 5
switch(x) {
  case "5": {
          console.log("hello")
          break
  }
  case 5: {
          console.log("goodbye") //prints
          break
  }
}
```

JavaScript Numbers

Floating point

In JS all numbers are represented as 64-bit double-precision floating-point (decimal number) format (IEEE 754 standard.
That is why JS can only **safely represent integers**: Up to **9007199254740991** + (2^{53}-1) and Down to **-9007199254740991** -(2^{53}-1).
Integer values outside this range lose precision. E.g. **Safe integer**: 9007199254740991. This is **not safe**: 9007199254740992.
Simply: Numbers without a period or exp. are accurate up to 15 digits

```
let x1 = 34.00;//decimal number
let x2 = 34;//without decimals
```

```
console.log(0.1 + 0.2); //0.30000000000000004
console.log(0.1 + 0.2 === 0.3); //false // ???
x = 1.0000000000000001
console.log(x === 1); //true // ???
```

Also, **Behaves differently in different browsers** once converts string into an integer.

```
let x = Number.MAX_SAFE_INTEGER;
let x = Number.MIN_SAFE_INTEGER;
```

```
Number.isInteger()
Number.isSafeInteger()
```

```
var num1 = 999999999999999, num2 = 9999999999999999;
console.log(num1);//999999999999999
console.log(num2);//10000000000000000

console.log(0.1 + 0.2 == 0.3); //false
x = 1.0000000000000001
console.log(x == 1); //true
```

Note

Most programming languages have many number types: Whole Numbers: byte (8-bit), short (16-bit), int (32-bit), long (64-bit), Real numbers (floating-point): float (32-bit), double (64-bit).
(not JavaScript - **no integer, no long,** etc.)

f-concat-nan-inf.js, f-float-parseint-demo.js

JavaScript BigInt

All JavaScript numbers are stored in a 64-bit floating-point format. JavaScript **BigInt** is a new datatype (ES2020) that can be used to store integer values that are too big to be represented by a normal JavaScript Number.

```
let x = BigInt("12345678901234567890123456789O");
```

Extra large or extra small numbers can be written with scientific (exponential) notation:

```
let y = 123e5;    // 12300000
let z = 123e-5;   // 0.00123
```

JavaScript EPSILON

`Number.EPSILON` is the difference between the smallest floating point number greater than 1 and 1. e.g. `let x = Number.EPSILON;`

Notes

Arithmetic between a `BigInt` and a `Number` is not allowed (type conversion lose information).
Unsigned right shift (>>>) can not be done on a `BigInt` (it does not have a fixed width).

BigInt Division Example

```
let x = 5n;
let y = x / 2;
// Error: Cannot mix BigInt and other types, use explicit conversion.
let x = 5n;
let y = Number(x) / 2;
```

NaN – NaN (Not-a-Number) is not equal to any value (including itself, **NaN !=== NaN**) and is essentially an **illegal number value**, but *typeOf(NaN)===number //true*. Use isNaN(number) to check for NaNs.

```
var isNumber = function isNumber(value) {
    return typeof value === 'number' && isFinite(value); //not NaN & Infinity
}
console.log(isNumber(NaN))//false
console.log(isNumber(123))//true
console.log(isNumber("Oraz"))//false
```

```
Using a Number property on a variable, expression, or
value, will return undefined
var x = 6;
console.log(x.MAX_VALUE);//undefined
//no warning, no error, in JAVA you have a compiler error
```

parseInt – no warning or informative message when converting also issue is with dates/times/

Eg1: parseInt("**32**") and parseInt("**32 meters**") produce the same result 32,

Eg2: parseInt("08") and parseInt("09") based on Base 8, but es5 (integer) **;)** Provide Radix.

```
console.log(parseInt("23")  );
//=> 23
console.log(parseInt("023")   );
//=> should be 19 (octal)  //WHY 23?
console.log(parseInt("023", 8) );
```

Never write a number with a leading zero (like 07). Some JavaScript versions interpret numbers as octal if they are written with a leading zero.

Note: `Number.parseInt()` and `Number.parseFloat()` are the same as the global methods `parseInt()` and `parseFloat()`. The purpose is modularization of globals (to make it easier to use the same JS code outside the browser).

Take it into consideration that once you let the language handle coercion for you, it acts differently, coercion is not applied for string concatenation "+" but for **other arithmetic operations coercion is applied**

```
console.log(10 + "10");//1010
console.log("2" + "3");//23, why not behave like "2" - "3" as below?
console.log(2 + 3);//5
console.log("2" - "3");//-1
console.log(2 - 3);//-1
console.log("100" / "10");//10
console.log("63" * "10");//630
```

f-concat-nan-inf.js
f-float-parseint-demo.js

Bitwise

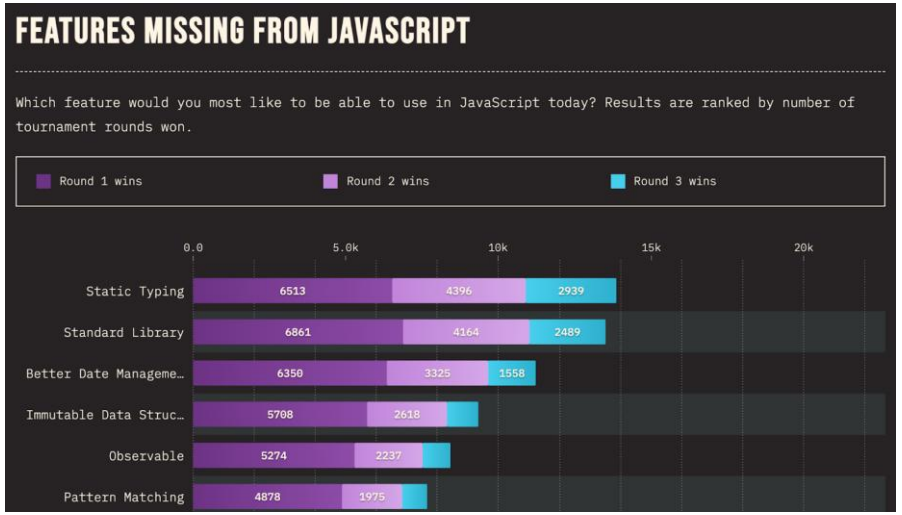
In Java bit operator work with integers. BUT JavaScript has no integers.
It works on 32bit, result converted back to JS. In JavaScript, bitwise operators (&, |, ^, ~, >>, ..) are very far from the hardware and very slow. JavaScript is rarely used for doing bit manipulation. As a result, in JavaScript programs, it is more likely that **& is a mistyped && operator**.

f-bitwise-opt.js

Modern Javascript

Node.js != JS

- ES TC39, https://github.com/tc39, TC-39 Process, Ecma-262
- V8 Engine will follow implementing TC39
- Yearly Releases since ES2015 [ES6], ES2016, ….
- 5 StagedProcess [0-Strawman,1-Proposal,2-Draft,3-Candidate,4-Finished]
- Babel - faster
- Variables and Block Scopes, Object Literals
- Arrow Functions
- Destructing and Rest/Spread
- Inheritance – state and action based (in Java action based)
- Promises and Async/Await
- Classes,…
- **Templates-string (interpolation, ` ${dyno-exp}`)** (also like **multi-lines** Java 13)
- **Dynamic properties**
- **Not yet – TS (type, type aliases, etc…)**
- **BUT, Good bye TS time is approaching –Native JS Proposals (Type Annotation) coming**



Prefer ECMAScript Features to TypeScript Features

JavaScript ES5

ECMAScript 2009, also known as ES5, was the **first major** revision to JavaScript. See ES features ..

`"use strict"` defines that the JavaScript code should be executed in "strict mode".

The `charAt()` method returns the character at a specified index (position) in a string:

ES5 lets you define object methods with a syntax that looks like getting or setting a property.

With the `bind()` method, an object can borrow a method from another object.

ES5 Features

- "use strict"
- String[*number*] access
- Multiline strings
- String.trim()
- Array.isArray()
- Array forEach()
- Array map()
- Array filter()
- Array reduce()
- Array reduceRight()
- Array every()
- Array some()
- Array indexOf()
- Array lastIndexOf()
- JSON.parse()
- JSON.stringify()
- Date.now()
- Date toISOString()
- Date toJSON()
- Property getters and setters
- Reserved words as property names
- Object.create()
- Object.keys()
- Object management
- Object protection
- Object defineProperty()
- Function bind()
- Trailing commas

JavaScript ES6

ECMAScript 2015 a.k.a ES6 **second major** revision of JS

The `let` keyword allows you to declare a variable with block scope.

The JavaScript `for/of` statement loops through the values of an iterable objects.

New Features in ES6

- ✓ The let keyword
- ✓ The const keyword
- ✓ Arrow Functions
- ✓ The {a,b} = Operator
- ✓ The [a,b] = Operator
- ✓ The ... Operator
- ✓ For/of
- ✓ Map Objects
- ✓ Set Objects
- ✓ Classes
- ✓ Promises
- ✓ Symbol
- ✓ Default Parameters
- ✓ Function Rest Parameter
- ✓ String.includes()
- ✓ String.startsWith()
- ✓ String.endsWith()
- ✓ Array entries()
- ✓ Array.from()
- ✓ Array keys()
- ✓ Array find()
- ✓ Array findIndex()
- ✓ Math.trunc
- ✓ Math.sign
- ✓ Math.cbrt
- ✓ Math.log2
- ✓ Math.log10
- ✓ Number.EPSILON
- ✓ Number.MIN_SAFE_INTEGER
- ✓ Number.MAX_SAFE_INTEGER
- ✓ Number.isInteger()
- ✓ Number.isSafeInteger()
- ✓ New Global Methods
- ✓ JavaScript Modules

ECMAScript 2016

Old ECMAScript versions was named by numbers: ES5 and ES6.
From 2016, versions are named by year: ES2016, 2018, 2020 ...

New features in ECMAScript 2016:

- JavaScript Exponentiation (**): `let x = 5; let z = x ** 2; //25`
  `x ** y` produces the same result as `Math.pow(x, y)`
- JavaScript Exponentiation assignment (**=) `let x = 5; let z = x ** 2; //25`
- JavaScript Array includes() - if an element is present in an array

Prefer ECMAScript Features to TypeScript Features

ECMAScript 2017

New features in ECMAScript 2017:

- JavaScript String padding - `padStart()` and `padEnd()` to support padding at the beginning and at the end of a string
- JavaScript Object entries() - `Object.entries()` returns an array of the key/value pairs in an object
- JavaScript Object values() - `Object.values()` is similar to `Object.entries()`, but returns a single dimension array of the object values

- JavaScript async and await
- Trailing Commas in Functions

JavaScript allows trailing commas wherever a comma-separated list of values is accepted. In Array and Object Literals, Function Calls, Parameters, Imports and Exports. Not good for JSON, …

```
function myFunc(x,,,) {};
const myArr = [1,2,3,4,,,];
const myObj = {fname: John, age:50,,,};
```

- JavaScript Object.getOwnPropertyDescriptors

ECMAScript 2018

New Features in ECMAScript 2018

- Asynchronous Iteration - With asynchronous iterables, we can use the `await` keyword in `for/of` loops. E.g. `for await () {}`
- Promise Finally - finalizes the full implementation of the Promise object with Promise.finally
- Object Rest Properties - allows us to destruct an object and collect the leftovers onto a new object
- New RegExp Features - added 4 new RegExp features, (\p{...}), (?<=) and (?<!), s (dotAll) Flag
- JavaScript Shared Memory

```
let myPromise = new Promise();

myPromise.then();
myPromise.catch();
myPromise.finally();
```

JavaScript Threads - use the Web Workers API (for Browser based app) or Worker Threads (for NodeJS) to create threads

JavaScript Shared Memory: Shared memory is a feature that allows threads to access and update the same data in the same memory.
Instead of passing data between threads, you can pass a SharedArrayBuffer object that points to the memory where data is saved.
A **SharedArrayBuffer** object represents a fixed-length raw binary data buffer similar to the ArrayBuffer object.

ECMAScript 2019

- String.trimStart() - method works like trim(), but removes whitespace only from the start of a string.
- String.trimEnd() - works like trim(), but removes whitespace only from the end of a string.
- Object.fromEntries - method creates an object from iterable key / value pairs
- Optional catch binding - you can omit the catch parameter if you don't need it
- Array.flat() - creates a new array by flattening a nested array
- Array.flatMap() - first maps all elements of an array and then creates a new array by flattening the array
- Revised Array.Sort()
- Revised JSON.stringify()
- Separator symbols allowed in string litterals
- Revised Function.toString()

```
const myArr = [[1,2],[3,4],[5,6]];

const newArr = myArr.flat();//1,2,3,4,5,6
```

```
const myArr = [1, 2, 3, 4, 5,6];
const newArr = myArr.flatMap(x => [x, x * 10]);//1,10,2,20,3,30,4,40,5,50,6,60
```

New Features in ES2020

- BigInt - used to store big integer values that are too big to be represented by a a normal JavaScript Number
- String matchAll()
- The Nullish Coalescing Operator (??) - returns the first argument if it is not **nullish** (null or undefined).
- The Optional Chaining Operator (?.) – e.g. let name = car?.name;
- Logical AND Assignment Operator (&&=)
- Logical OR Assignment (||=)
- Nullish Coalescing Assignment (??=)
- Promise.allSettled() - method returns a single Promise from a list of promises

```
let name = null;
let text = "missing";
let result = name ?? text;
//like Elvis operator in Spring (?:)
```

New Features in ES2021

- Promise.any()
- String replaceAll()
- Numeric Separators (_) – e.g. const num = 1_000_000_000;

New Features in ES2022

- ✓ Array at()
- ✓ String at()
- ✓ RegExp /d
- ✓ Object.hasOwn()
- ✓ error.cause
- ✓ await import
- ✓ Class field declarations
- ✓ Private methods and fields

New Features in ES2023

- Array findLast()
- Array findLastIndex()
- Array toReversed()
- Array toSorted()
- Array toSpliced()
- Array with()
- #! (Shebang) – like Java 11 feature

New Features in ES2024

- Object.groupBy() //like Java stream API
- Map.groupBy()
- Temporal.PlainDate() – only date
- Temporal.PlainTime() – only time
- Temporal.PlainMonthDay() – no year
- Temporal.PlainYearMonth()

JS Reference,

Demo available: Testing Frameworks and Tools, Code Coverage

THANK YOU

References

https://www.w3schools.com/js/js_intro.asp
https://medium.com/codex/good-to-know-parts-when-coding-with-javascript-996cd68563d3
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects
https://medium.com/javascript-non-grata/the-top-10-things-wrong-with-javascript-58f440d6b3d8
https://blog.devgenius.io/45-javascript-super-hacks-every-developer-should-know-92aecfb33ee8
https://medium.com/@julienetienne/is-javascript-trash-part-1-5310ac4e20d0