

(Proper)

Object Oriented Programming and **SOLID**

by **Dario Cerviño Luridiana**
(dario.cervino.11@ull.edu.es)
and **Juan García Santos**
(juan.garcia.santos.20@ull.edu.es)





TEAM PRESENTATION



Dario Cerviño Luridiana

Email:

dario.cervino.11@ull.edu.es



Juan García Santos

Email:

juan.garcia.santos.20@ull.edu.es



Before we start

- This is **not** a guide on how to do OOP in JS...
- ...But, we'll touch on some basics.
- This is a guide on **good practices** of OOP (in JS).



CONTENTS

How to write OOP in JavaScript

1. What is OOP
2. How to do OOP in JS
3. Google Style Guide

How to design OOP in JavaScript

1. Code smells
2. What is SOLID
3. SOLID Principles
4. Other principles

Object Oriented Programming In JS





1

Basics of OOP

Basics of OOP

INHERITANCE

basing a class off of another class so that it maintains the same behavior as its parent class

POLYMORPHISM

having the same interface to instances of different types

CLASSES

hiding the unnecessary implementation details from the users

ABSTRACTION

bundling data and methods that work on that data within one unit

ENCAPSULATION



2

OOP in JavaScript



JS classes vs C++ classes

- JavaScript is not class based like C++.
- JS classes can be made with new/prototype (don't).
- JS classes can also be made with ES6 **class** keyword.
- In any case, JS classes are not like C++ classes.



Class syntax

C++ Class:

```
class MyClass{  
public:  
    int atributo;  
    MyClass() {  
        atributo = 1;  
    }  
  
    int method() {...}  
};
```

JavaScript Class:

```
class MyClass {  
    constructor() {  
        this.atributo = 1;  
    }  
  
    method() {...}  
}
```



Differences to look out for

- Classes can be dynamically generated and updated (**Avoid this**)
- JS doesn't implement protected access.
 - The convention is to finish protected properties with an underscore.



Similarities

- Methods explicitly stop being iterable.
(for-in loops work as intended)
- Static works the same way using the keyword **static**.
- Public and private access levels are quite similar.



Access Levels

C++ Access Levels:

```
class MyClass {  
    public int x;  
    protected int y;  
    private int z;  
};
```

JavaScript Access Levels:

```
class MyClass {  
    constructor() {  
        this.x = ...;  
        this.y_ = ...; // kinda  
        this.#z = ...;  
        // ! NOT INCOMPATIBLE  
        // WITH (public) this.z  
    }  
}
```



3

Google Style Guide on JavaScript Classes



Constructors and inheritance

- Constructors are optional.
- Subclass constructors must call **super()** before setting any fields or otherwise accessing **this**.
- Set all of a concrete object's fields in the constructor.



Fields (naming, privacy, initialization)

- Fields are all properties other than methods.
- End all **@private** fields' names with an underscore.
- Annotate non-public fields with the proper visibility annotation (**@private**, **@protected**, **@constructor**).

DO NOT touch the prototype

- ◆ Plain simple, don't define prototype properties.
- ◆ Why? Find out [here!](#)
- ◆ Also, don't use set/get
(They're weird)





Static methods

- ◆ Prefer module-local functions over private static methods.
- ◆ Static methods should only be called on the base class itself. Avoid using them dynamically.
- ◆ **NEVER** call static methods on subclasses that don't define them directly.



Overriding ToString

- The **toString** method may be overridden,
- But must always succeed and **never have visible side effects.**



Abstract classes and Interfaces

- Abstract classes and methods must be annotated with **@abstract**.
- Use them when appropriate.



Computed Properties

- **Computed properties** may only be used in classes when the property is a symbol.
- A **[Symbol.iterator]** method should be defined for any classes that are logically iterable.
- Use symbols sparingly.

Object Oriented Design in JS

(AKA How to detect, avoid and correct code smells)





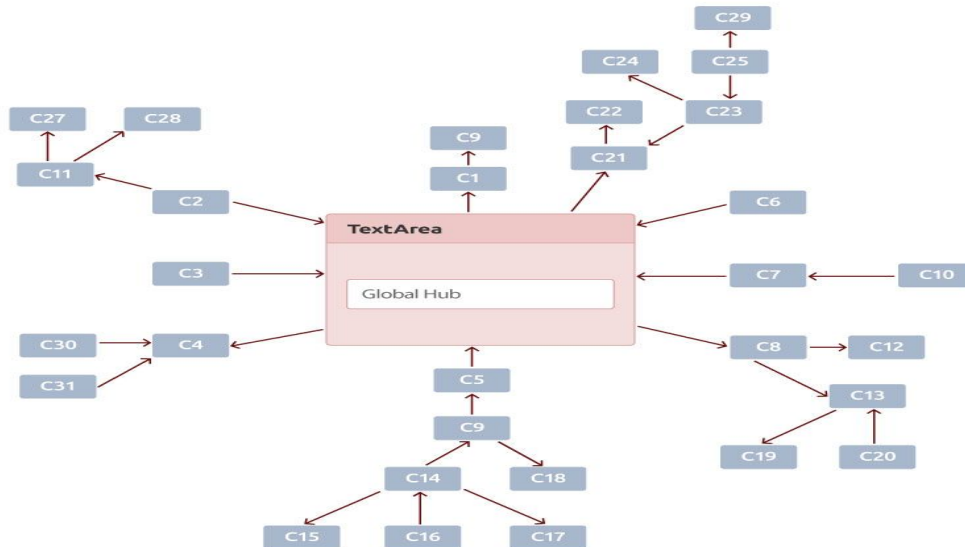
Code Smells

- “A code smell is any characteristic in the code that hints at a deeper problem”



Anti-patterns

- Counterpart of design patterns (AKA bad coding patterns).
- For example: The God Object





Examples

- Is it **rigid**? (hard to update)
- Is it **fragile**? (does not handle errors too well)
- Is it too **compact**? (unable to modularize it)
- Is it **over-complex**? (has unused defined methods)
- Is it **opaque**? (hard to understand at first sight)
- Is it **mirror-like**? (tons of similar code)



Code smell

- Too many levels of indentation

```
function confusingFunction(arg) {  
  for(const element of arg){  
    if(arg !== 0){  
      if(isOdd(arg){  
        switch(arg) {  
          ...  
        }  
      }  
    }  
  }  
}
```



Code smell

- Long functions (scrolling needed in order to read)

```
const main = () => {
  const MOST_POPULATED = maximize(countryData, 'population');
  const LEAST_POPULATED = minimize(countryData, 'population');
  const MOST_DENSE_NORTH = maximize(filter(countryData, 'continent',
    'North America'), 'density');
  const MOST_DENSE_SOUTH = maximize(filter(countryData, 'continent',
    'South America'), 'density');
  const MOST_DENSE = (MOST_DENSE_NORTH > MOST_DENSE_SOUTH ?
    MOST_DENSE_NORTH : MOST_DENSE_SOUTH);
  const continents = getAllVariations(countryData, 'continent');
  const longestLifeExp = [];
  for (let i = 0; i < continents.length; ++i) {
    longestLifeExp.push(maximize(filter(countryData, 'continent',
      continents[i]), 'expectancy', 3));
  }
  const maleHeightMean = [];
  for (let i = 0; i < continents.length; ++i) {
    maleHeightMean.push(meanOffTrait(filter(countryData, 'continent',
      continents[i]), 'height'));
  }
  const religionDataSet = invertDataset(countryData, 'religion');
  const popularReligions = maximize(religionDataSet, 'count', 5);
  const totalReligious = totalOffTrait(religionDataSet, 'count');
  const coldestCountries = [];
  for (let i = 0; i < continents.length; ++i) {
    coldestCountries.push(minimize(filter(countryData, 'continent',
      continents[i]), 'temperature', 3));
  }
  console.log('El país más poblado del mundo es ' + MOST_POPULATED +
    ` (${countryData[MOST_POPULATED].population} habitantes) y el menos `
+
    `poblado es ${LEAST_POPULATED} (` +
    `${countryData[LEAST_POPULATED].population} habitantes).\n`);
  console.log('El país con mayor densidad de población en América es ' +
    MOST_DENSE + '\n');
  console.log('Los 3 países con mayor esperanza de vida en cada uno de los ' +
    'continentes son:\n');
  for (let i = 0; i < continents.length; ++i) {
    console.log(` - ${continents[i]}: ${longestLifeExp[i].join(',
  ')}.\n`);
  }
  console.log('La media de la altura promedio de los varones en cada uno de ' +
    'los continentes es:\n');
  for (let i = 0; i < continents.length; ++i) {
    console.log(` - ${continents[i]}: ${maleHeightMean[i]} m.\n`);
  }
  console.log('Los porcentajes correspondientes (en función del número de ' +
    'países con cada religión) a las 5 religiones más extendidas en el '
+
    'mundo son:\n');
  for (let i = 0; i < popularReligions.length; ++i) {
    console.log(` - ${popularReligions[i]}: ` +
    `${(religionDataSet[popularReligions[i]].count /
    totalReligious * 100).toFixed(2)} ` +
    '%\n`);
  }
  console.log('Los países con más bajas temperaturas en cada uno de los ' +
    'continentes son los siguientes:\n');
  for (let i = 0; i < continents.length; ++i) {
    console.log(` - ${continents[i]}: ${coldestCountries[i].join(',
  ')}.\n`);
  }
}
```



Code smell

- ◆ Too many parameters

```
function confusingFunction(bookName, lineNumber,  
page, author, character, cover, publisher, age,  
length, word, paper, editor, publishingYear,  
awards, ...config) {  
    // Does stuff  
}
```



Code smell

- Wrong Use of Equality (== vs ===)

```
const checkEmptyString(string) => {  
  return string == '';  
}
```

```
checkEmptyString(0);      // true  
checkEmptyString('0');    // false
```



Code smell

- Outdated Comments

```
/**  
 * Planet Pluto  
 */  
const Pluto = new DwarfPlanet();
```



Other code smells

- Duplicated Code
- Large class
- Almost empty class
- Changes must be made in multiple classes each time
- Unmeaningful names
- Excessively long identifier



SOLID Principles



Single Responsibility Principle



“A class should have one, and only one, reason to change.”



Open Closed Principle



“Classes should be open for extension, closed for modification.”



Iskov Substitution Principle



“Derived classes must be substitutable for their base classes.”



Interface Segregation Principle



“Many client specific interfaces are better than one generic-purpose interface.”



Dependency Inversion Principle



“Everything should depend upon
abstractions, not details.”



...ther Principles



DRY

- “Don’t Repeat Yourself”
- Divide the logic of your system into smaller reusable pieces as much as possible.



KISS

- ◆ “Keep It Simple and Stupid”
- ◆ Keep your methods small (should not be larger than 40-50 lines).
- ◆ Each method should only solve one small problem.



YAGNI

- “You Ain’t Gonna Need It”
- Do not add any functionality until it’s deemed necessary.
- Carpe Diem, do not think in future needs



Preference Principles

- Composition > Inheritance
- Interface > Implementation



What should be common sense at this point

- Avoid Global stuff like the plague.
- If it changes, encapsulate.
- Delegate (prevent God Objects).



Wrapping up

- ◆ JavaScript is not at it's best in OOP.
- ◆ But if you use OOP, use ES6 classes.
- ◆ And more importantly: If you do, please use SOLID & Co.



References

- [Software design principles](#)
- [OOP best practices](#)
- [10 OOP principles](#)
- [Code Smell](#)
- [SOLID Code Examples](#)



THANKS!

Any questions?

You can contact us at:

- ◆ dario.cervino.11@ull.edu.es
- ◆ juan.garcia.santos.20@ull.edu.es

