Clean Code

Code smells, refactoring and good practices

Jose Fenic Peiteado Padilla

fenic.peiteado.20@ull.edu.es

Pablo Santana González

pablo.santana.gonzalez.15@ull.edu.es

Topics

- Code smells.
- Clean Code.
- Good practices.
- Refactoring.

What is a Code Smell

- Code smell is a hint that something has gone wrong somewhere in your code.
- There might be an error or not, but other people may feel something is wrong.
- Demands your attention.

Types of Code Smell

- General.
- Functions.
- Class-level code smells.

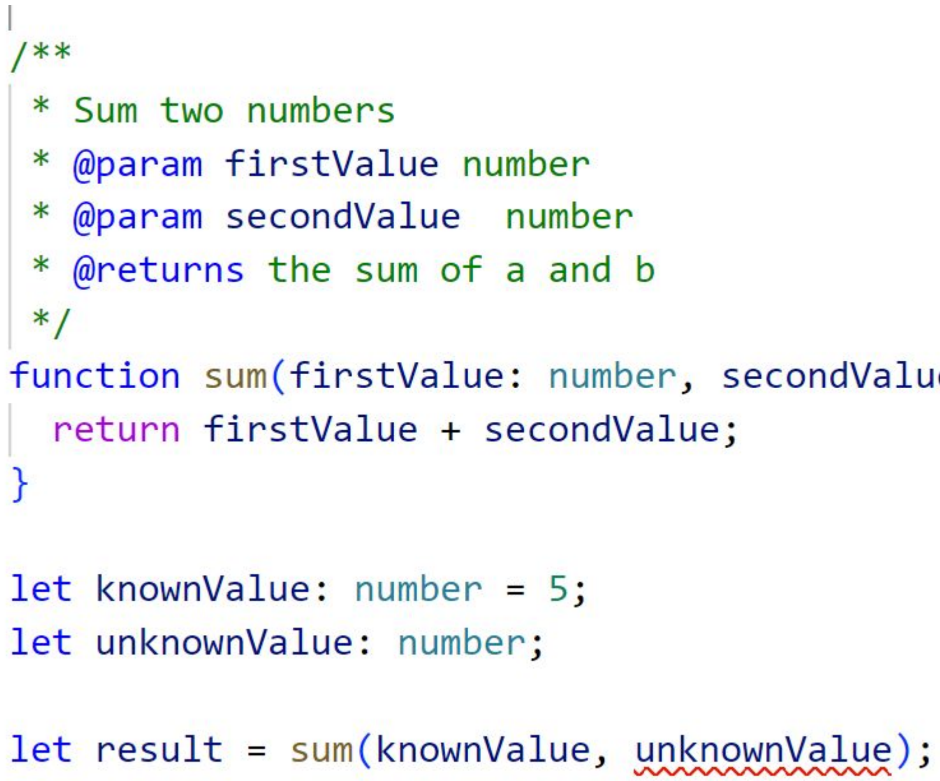
Code Smells: General

- Undefined variables.
- Hard-coded values.
- Magic numbers.
- Many indentation levels.

Undefined variables

- Some languages allow it, avoid it anyways.
- Unexpected errors caused by not knowing the value.

Undefined variables



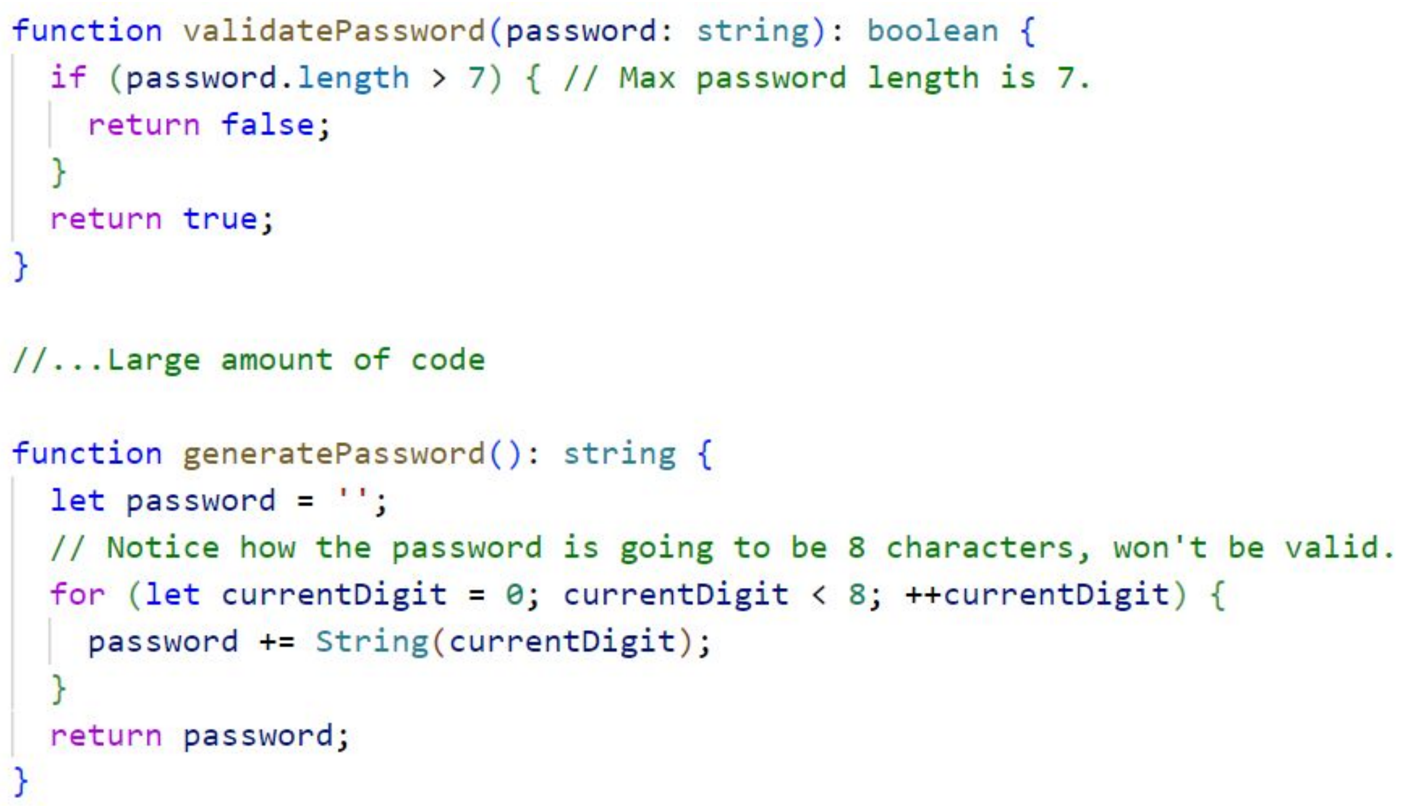
```
/**
 * Sum two numbers
 * @param firstValue number
 * @param secondValue  number
 * @returns the sum of a and b
 */
function sum(firstValue: number, secondValue: number): number {
  return firstValue + secondValue;
}

let knownValue: number = 5;
let unknownValue: number;

let result = sum(knownValue, unknownValue);
```

Hard-coded values

- Various literals that reference the same element across the code.

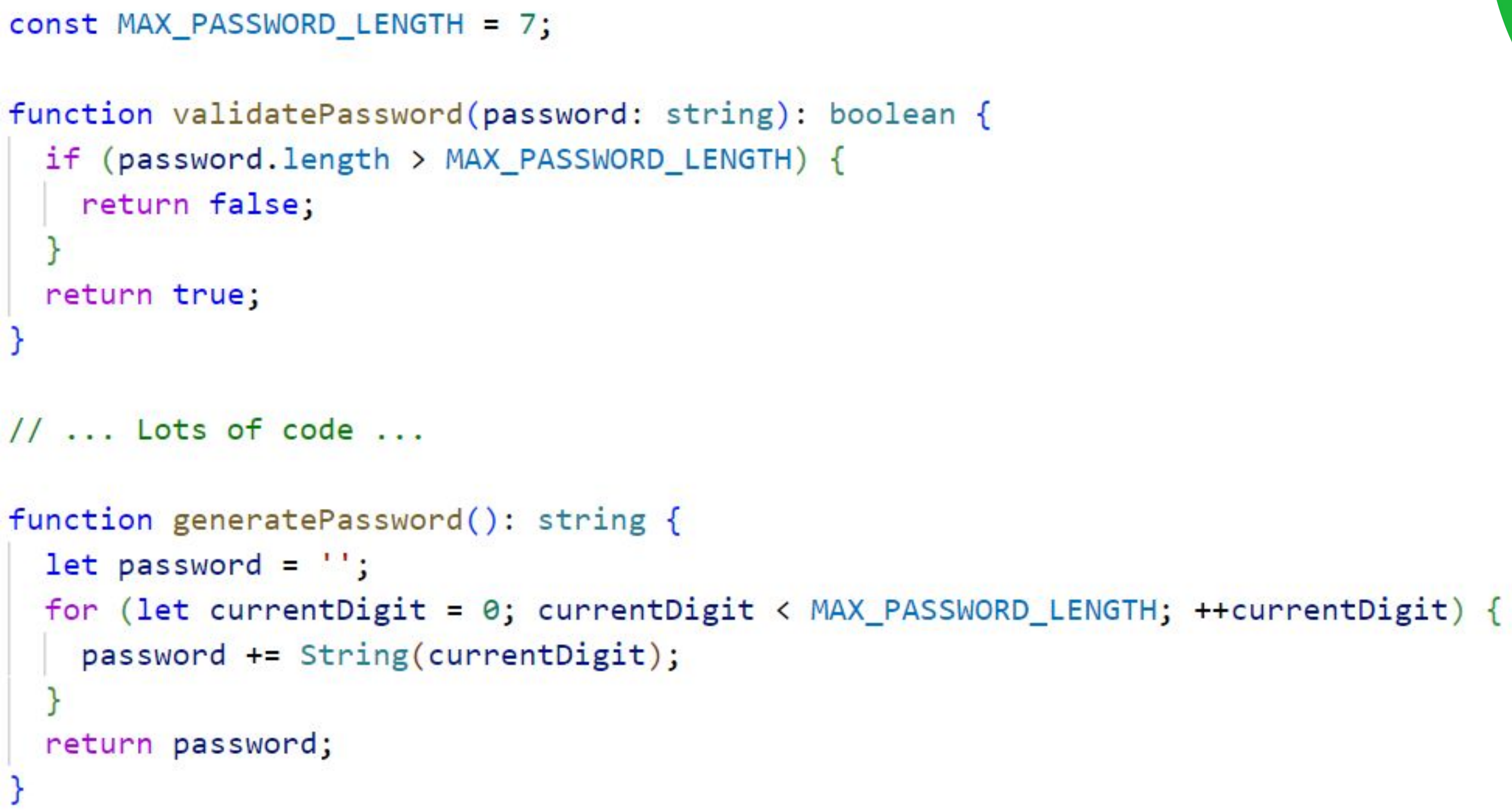
- Use a constant.

Hard-coded values



```
function validatePassword(password: string): boolean {
  if (password.length > 7) { // Max password length is 7.
    return false;
  }
  return true;
}

//...Large amount of code

function generatePassword(): string {
  let password = '';
  // Notice how the password is going to be 8 characters, won't be valid.
  for (let currentDigit = 0; currentDigit < 8; ++currentDigit) {
    password += String(currentDigit);
  }
  return password;
}
```

Hard-coded values

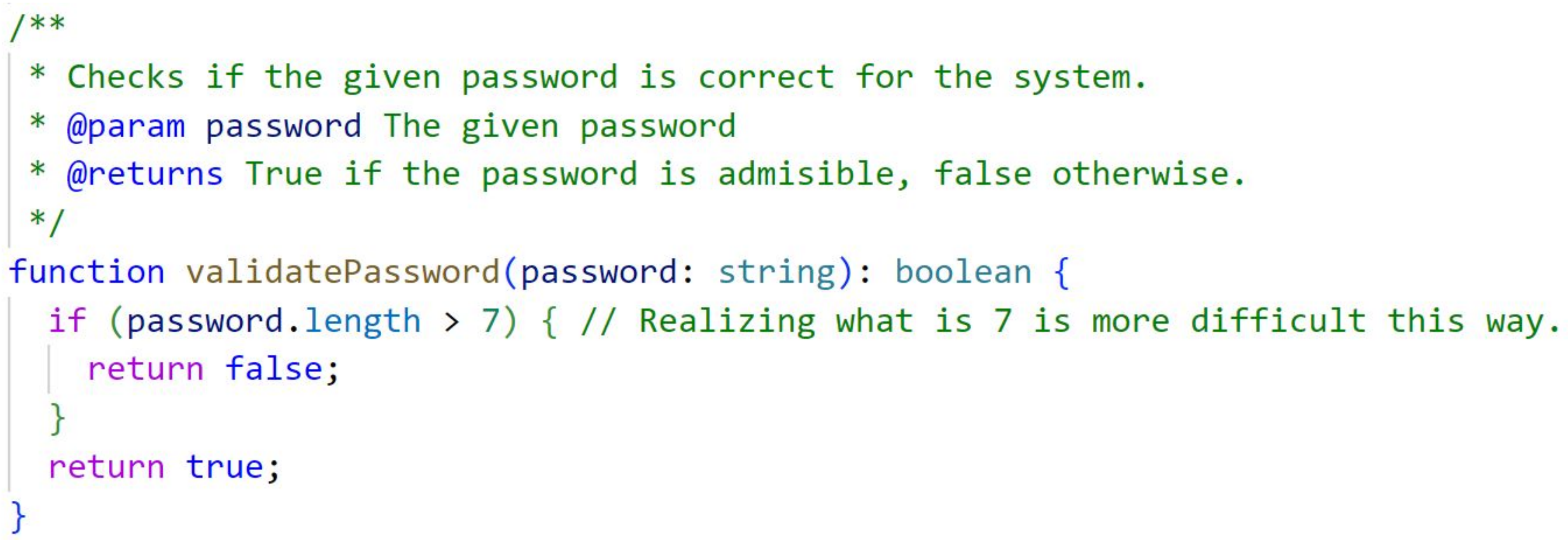


```
const MAX_PASSWORD_LENGTH = 7;

function validatePassword(password: string): boolean {
  if (password.length > MAX_PASSWORD_LENGTH) {
    return false;
  }
  return true;
}

// ... Lots of code ...

function generatePassword(): string {
  let password = '';
  for (let currentDigit = 0; currentDigit < MAX_PASSWORD_LENGTH; ++currentDigit) {
    password += String(currentDigit);
  }
  return password;
}
```

Magic Numbers

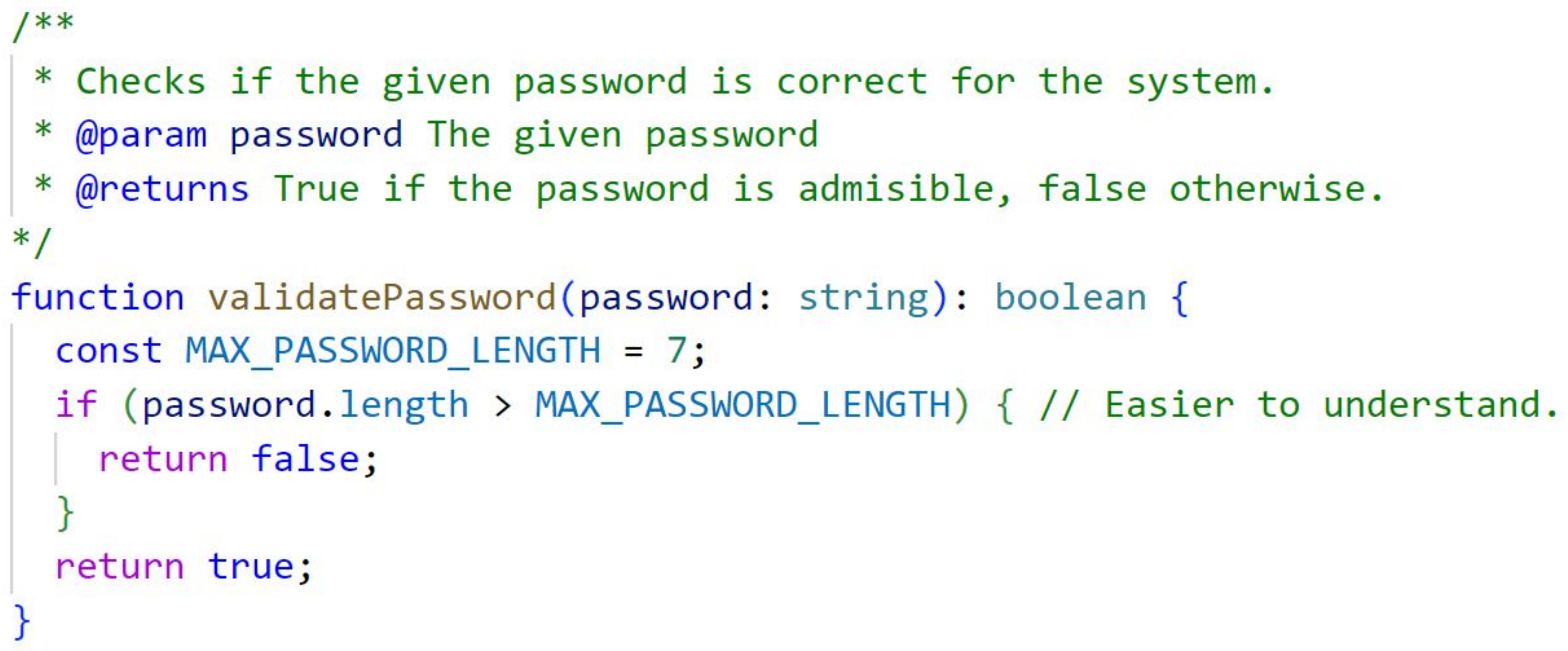
- A specific kind of instance of hard-coded values.
- Number literals with no explicit meaning.
- Tip: if you see a random number and think “what is that?”, is probably a magic number.
- Again, use a constant.

Magic Numbers



```
/**
 * Checks if the given password is correct for the system.
 * @param password The given password
 * @returns True if the password is admisible, false otherwise.
 */
function validatePassword(password: string): boolean {
  if (password.length > 7) { // Realizing what is 7 is more difficult this way.
    return false;
  }
  return true;
}
```

Magic Numbers ✓

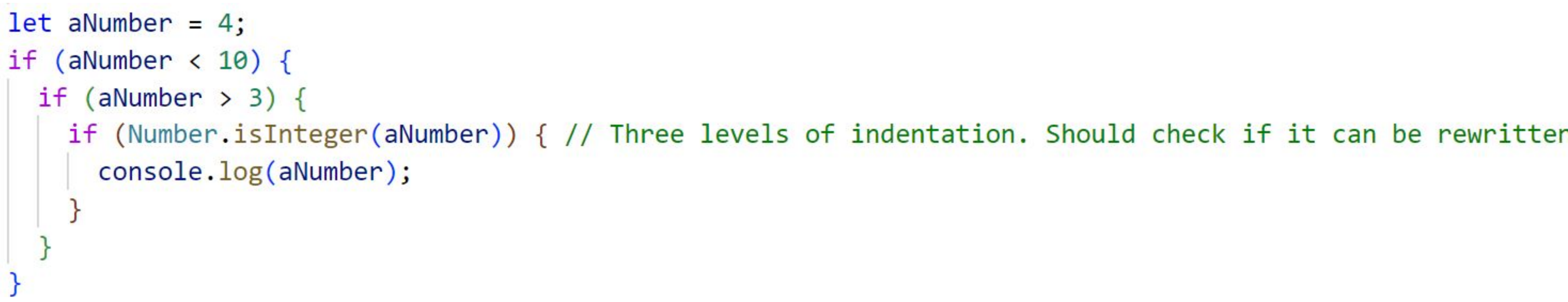


```
/**
 * Checks if the given password is correct for the system.
 * @param password The given password
 * @returns True if the password is admisible, false otherwise.
*/
function validatePassword(password: string): boolean {
  const MAX_PASSWORD_LENGTH = 7;
  if (password.length > MAX_PASSWORD_LENGTH) { // Easier to understand.
    return false;
  }
  return true;
}
```

Many indentation levels

- In control statements.
- Makes the code harder to read.
- Three levels deep is already suspicious (although it doesn’t mean it’s bad code).

Many indentation levels



```
let aNumber = 4;
if (aNumber < 10) {
  if (aNumber > 3) {
    if (Number.isInteger(aNumber)) { // Three levels of indentation. Should check if it can be rewritter
      console.log(aNumber);
    }
  }
}
```

Many indentation levels ✓

```
let aNumber = 4;
if (aNumber < 10 && aNumber > 3 && Number.isInteger(aNumber)) { // One line, much better
  console.log(aNumber);
}
```

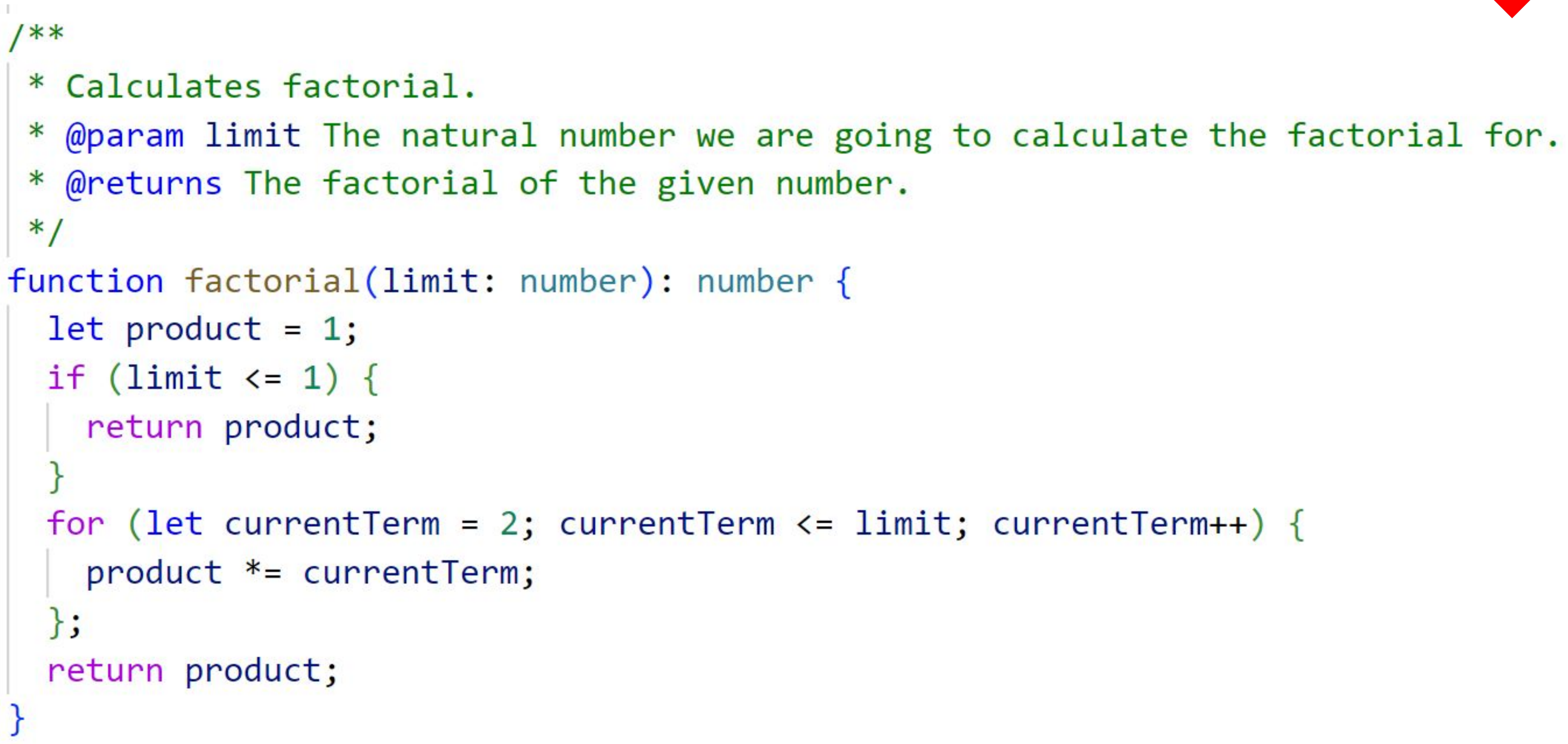
Code Smells: Function

- Poor use of exceptions.
- Long function.
- Many Parameters (Limit 3).

Poor use of exceptions

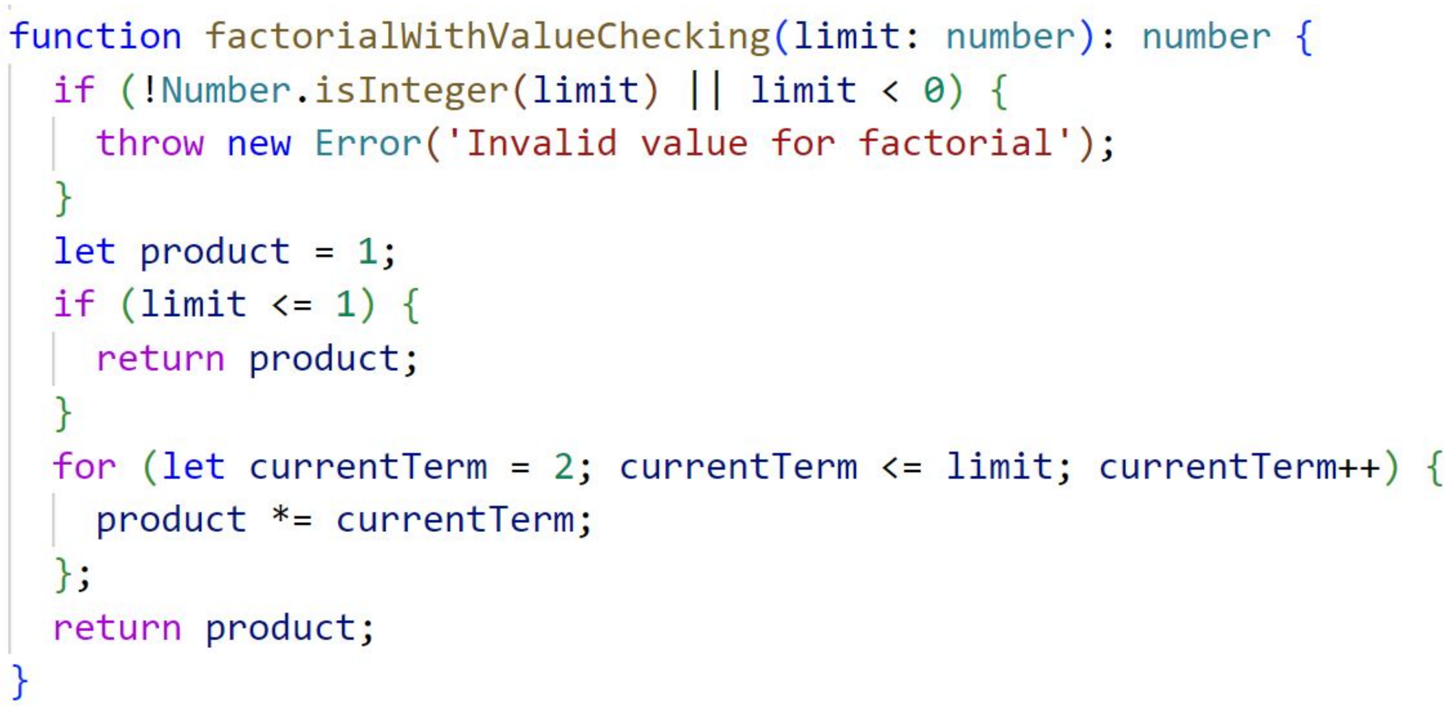
- Functions parameters usually have a range of valid values.
- Not controlling the values may cause undefined behaviour.
- Use exceptions to warn that the function isn’t working as planned.
- Also use try and catch blocks.

Poor use of exceptions



```
/**
 * Calculates factorial.
 * @param limit The natural number we are going to calculate the factorial for.
 * @returns The factorial of the given number.
 */
function factorial(limit: number): number {
  let product = 1;
  if (limit <= 1) {
    return product;
  }
  for (let currentTerm = 2; currentTerm <= limit; currentTerm++) {
    product *= currentTerm;
  };
  return product;
}
```

Poor use of exceptions



```
function factorialWithValueChecking(limit: number): number {
  if (!Number.isInteger(limit) || limit < 0) {
    throw new Error('Invalid value for factorial');
  }
  let product = 1;
  if (limit <= 1) {
    return product;
  }
  for (let currentTerm = 2; currentTerm <= limit; currentTerm++) {
    product *= currentTerm;
  };
  return product;
}
```

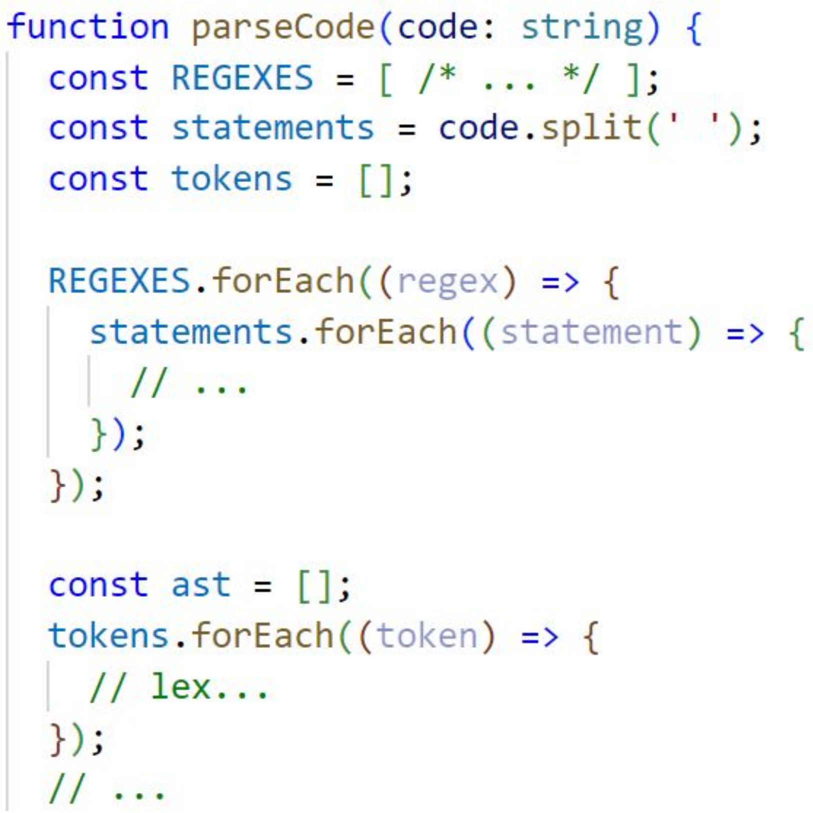
Long functions

- A long function sometimes is needed but it can also mean that is doing multiple things.

- JS and TS Google Style Guides have nothing on function length, but C++'s recommends checking the function if it is over 40 lines long.

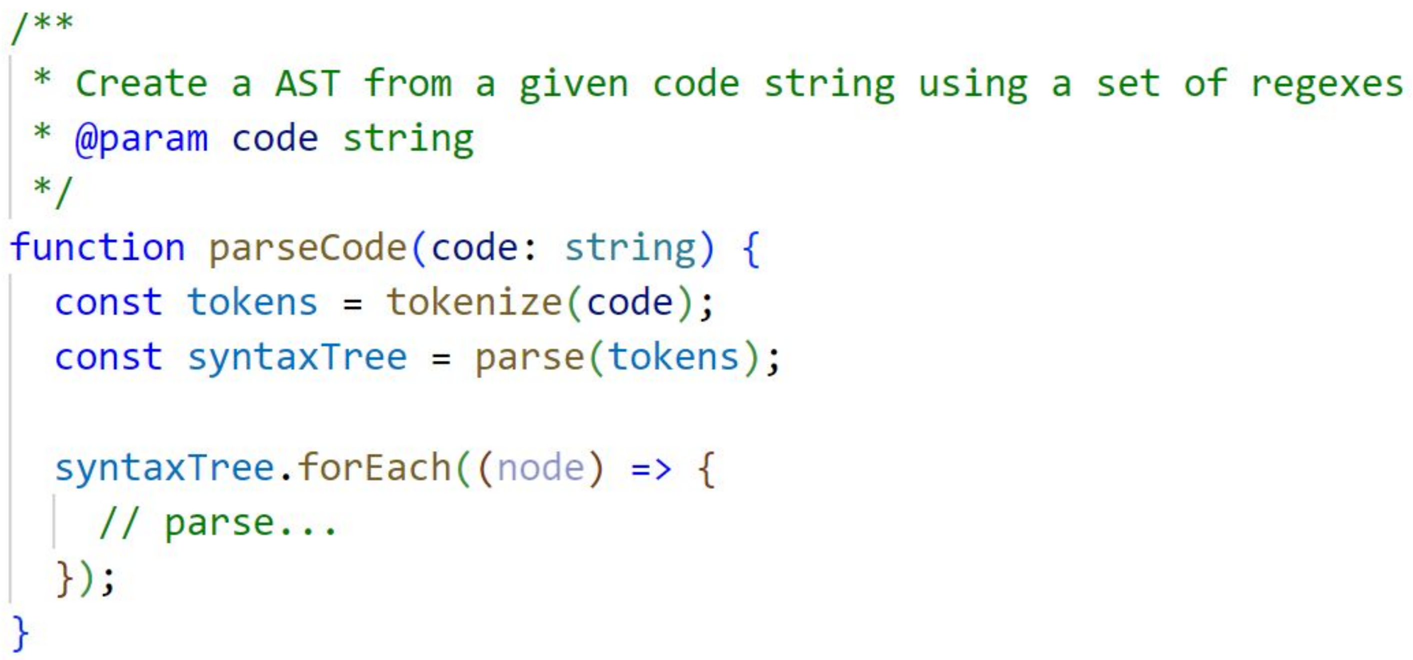
- Another sign of this is how hard it's to describe the function.

Long functions



```
function parseCode(code: string) {
  const REGEXES = [ /* ... */ ];
  const statements = code.split(' ');
  const tokens = [];

  REGEXES.forEach((regex) => {
    statements.forEach((statement) => {
      // ...
    });
  });

  const ast = [];
  tokens.forEach((token) => {
    // lex...
  });
  // ...
```

Long functions ✓

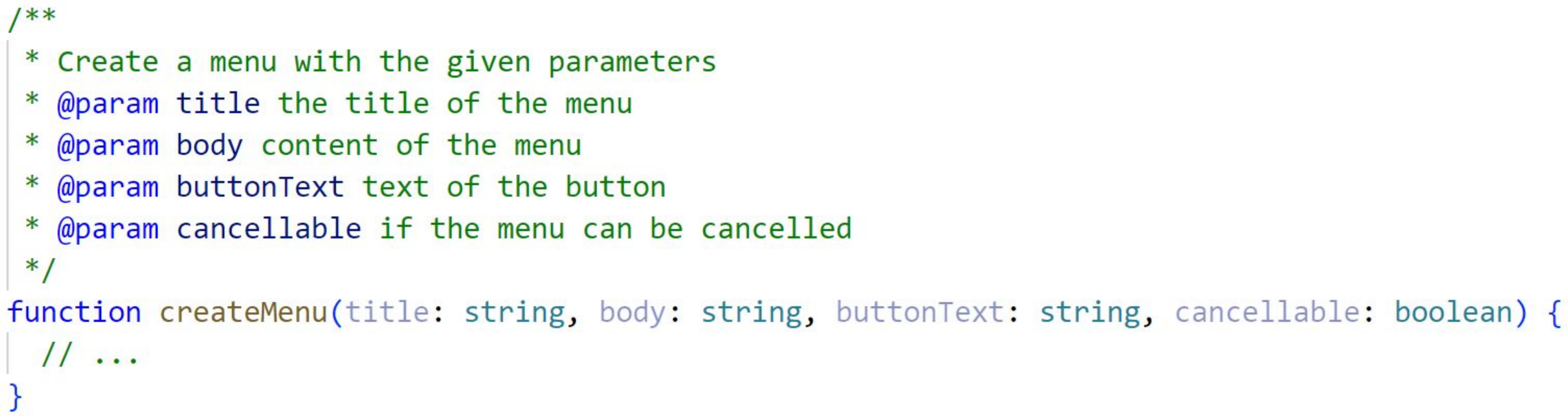


```
/**
 * Create a AST from a given code string using a set of regexes
 * @param code string
 */
function parseCode(code: string) {
  const tokens = tokenize(code);
  const syntaxTree = parse(tokens);

  syntaxTree.forEach((node) => {
    // parse...
  });
}
```

Many parameters (limit 3)

- A function needs a good reason to have three parameters, let alone more.
- Testing the function gets harder.
- A parameter with an object with various options is a easier to test alternative and more readable.

Many parameters (limit 3)



```
/**
 * Create a menu with the given parameters
 * @param title the title of the menu
 * @param body content of the menu
 * @param buttonText text of the button
 * @param cancellable if the menu can be cancelled
 */
function createMenu(title: string, body: string, buttonText: string, cancellable: boolean) {
  // ...
}
```

Many parameters (limit 3) ✓

```
// MenuOptions is an object that contains the parameters of the menu
type MenuOptions = {
  title: string,
  body: string,
  buttonText: string,
  cancellable: boolean
};

/**
 * Create a menu with the given parameters
 * @param options the parameters of the menu
 */
function createMenu(options: MenuOptions) {
  // ...
}
```

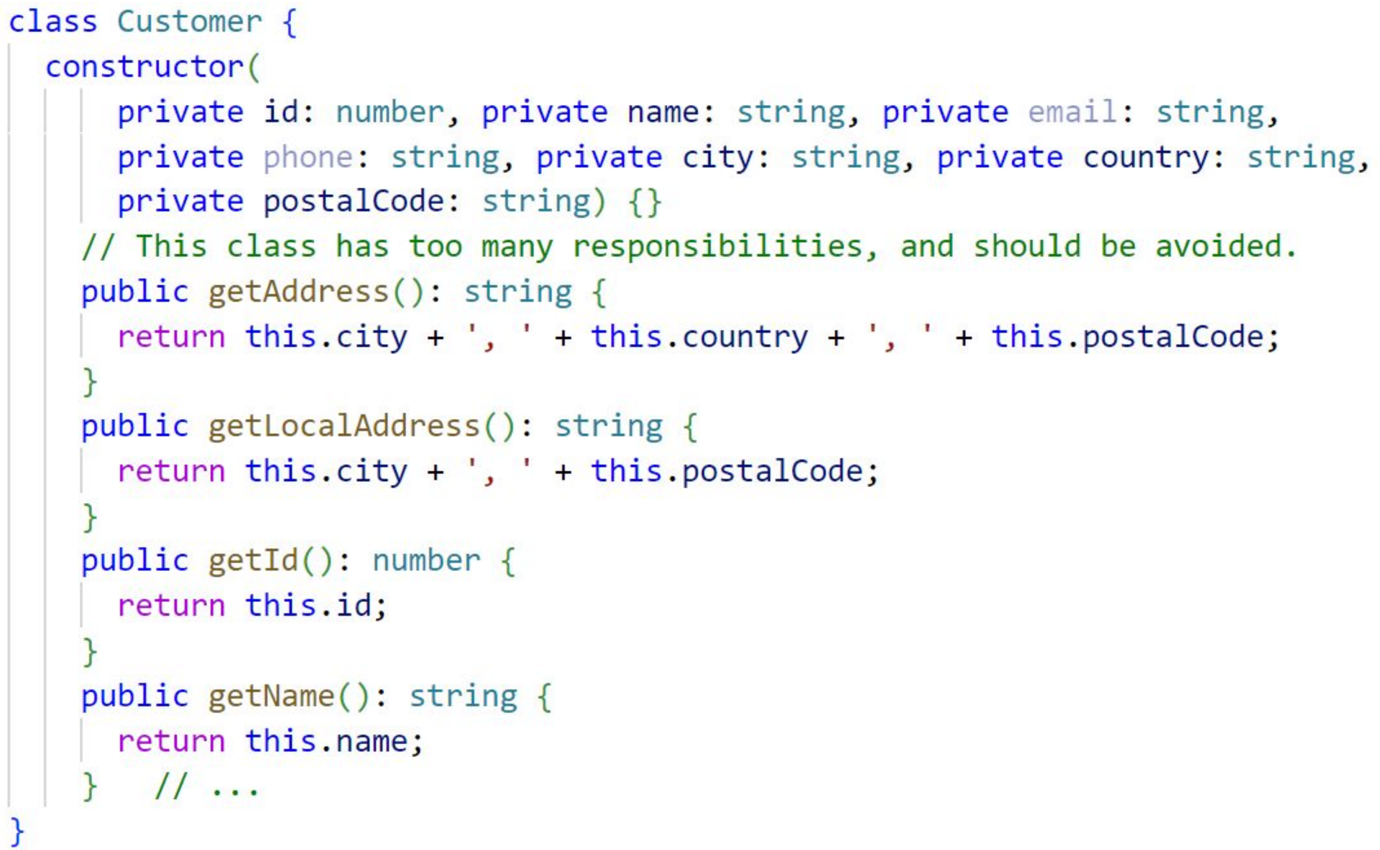
Code Smells: Class

- God classes.
- Inheritance instead of composition.
- Class with only getters and setters.

God classes

- Classes that do a lot of things (or everything) are a sign that the class should be broken into other classes.
- Remember that classes should do only one thing.
- SOLID - SRP (**S**ingle **R**esponsibility **P**rinciple).

God classes



```
class Customer {
  constructor(
      private id: number, private name: string, private email: string,
      private phone: string, private city: string, private country: string,
      private postalCode: string) {}
    // This class has too many responsibilities, and should be avoided.
    public getAddress(): string {
      return this.city + ', ' + this.country + ', ' + this.postalCode;
    }
    public getLocalAddress(): string {
      return this.city + ', ' + this.postalCode;
    }
    public getId(): number {
      return this.id;
    }
    public getName(): string {
      return this.name;
    }   // ...
}
```

God classes ✓

```
class Address {
  constructor(
    private city: string, private country: string,
    private postalCode: string) {}

  // Method that return only postal code and city.
  public getLocalAddress(): string {
    return this.city + ', ' + this.postalCode;
  } // ... other methods
}

class Customer {
  constructor(private address: Address/*...*/) {}

  public getAddress(): Address {
    return this.address;
  }
  // Other methods...
}
```

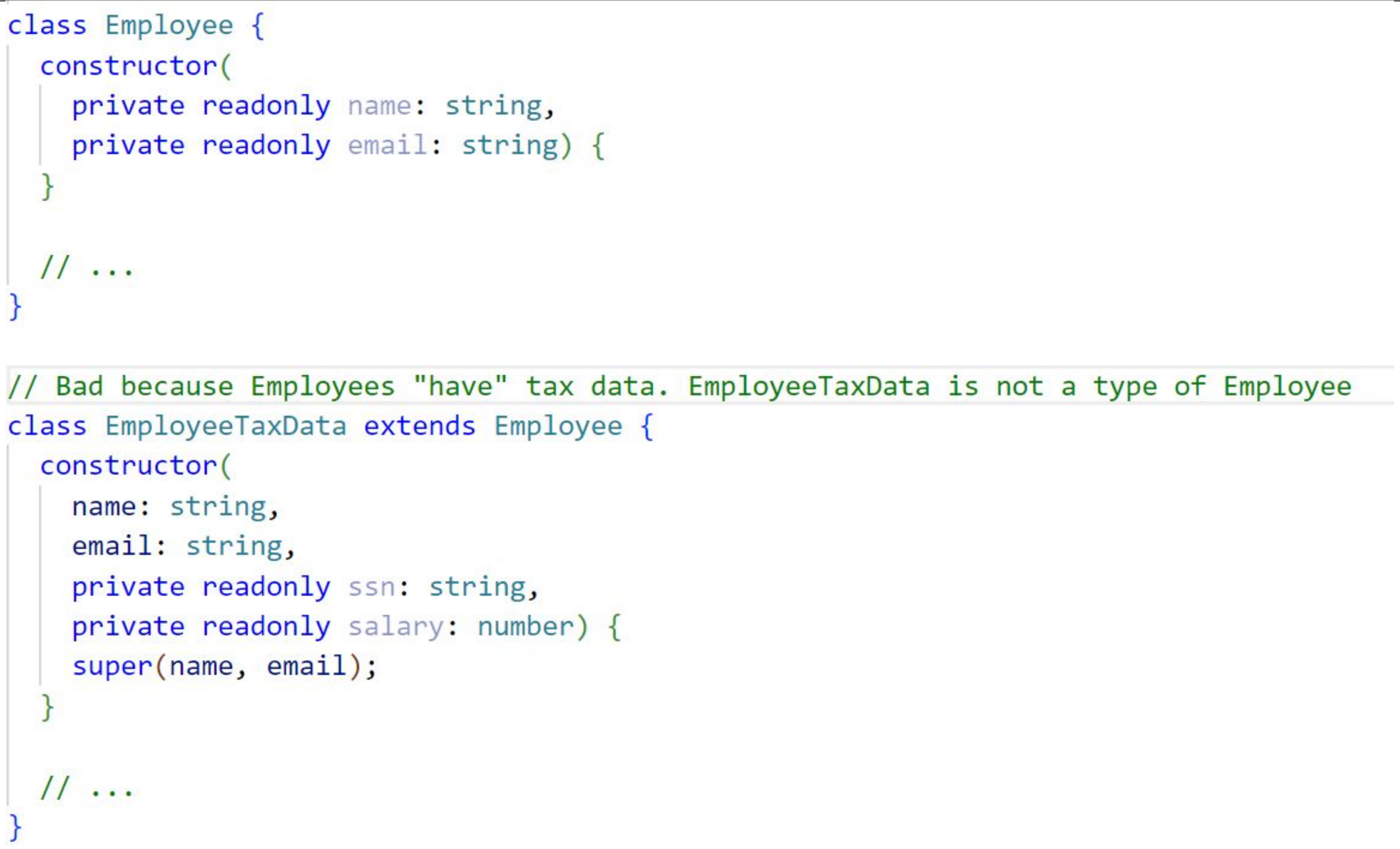
Inheritance instead of composition

- Inheritance is less common than composition.
- Mostly, you are going to be using composition.

Ask yourself the following questions:

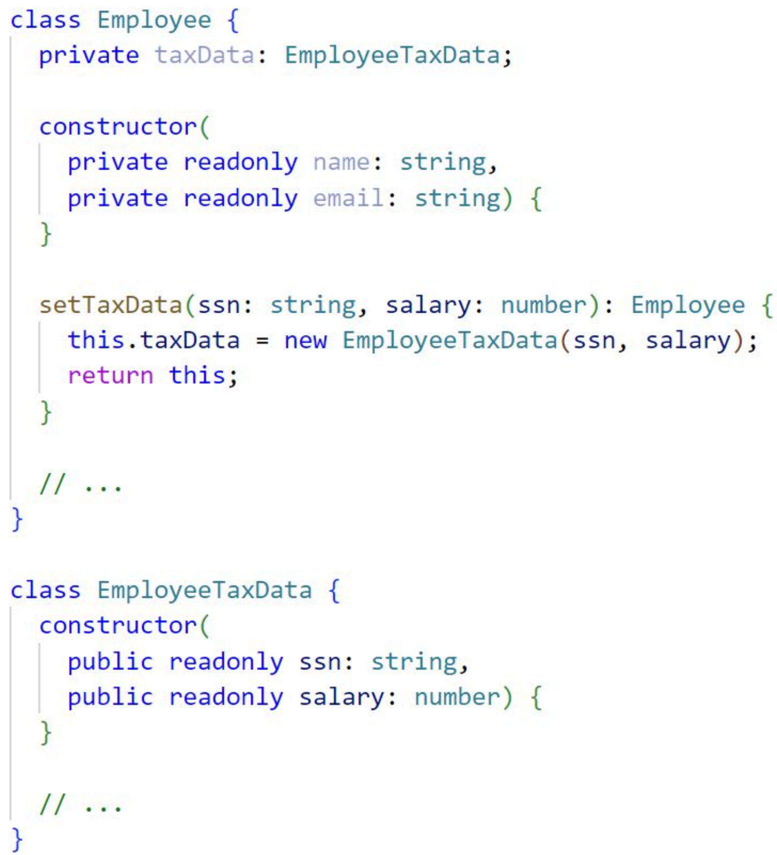
- Class B is Class A? Inheritance is appropriate.
- Class A has Class B? Use composition.

Inheritance instead of composition ❌



```
class Employee {
  constructor(
    private readonly name: string,
    private readonly email: string) {
  }

  // ...
}

// Bad because Employees "have" tax data. EmployeeTaxData is not a type of Employee
class EmployeeTaxData extends Employee {
  constructor(
    name: string,
    email: string,
    private readonly ssn: string,
    private readonly salary: number) {
    super(name, email);
  }

  // ...
}
```

Inheritance instead of composition ✓



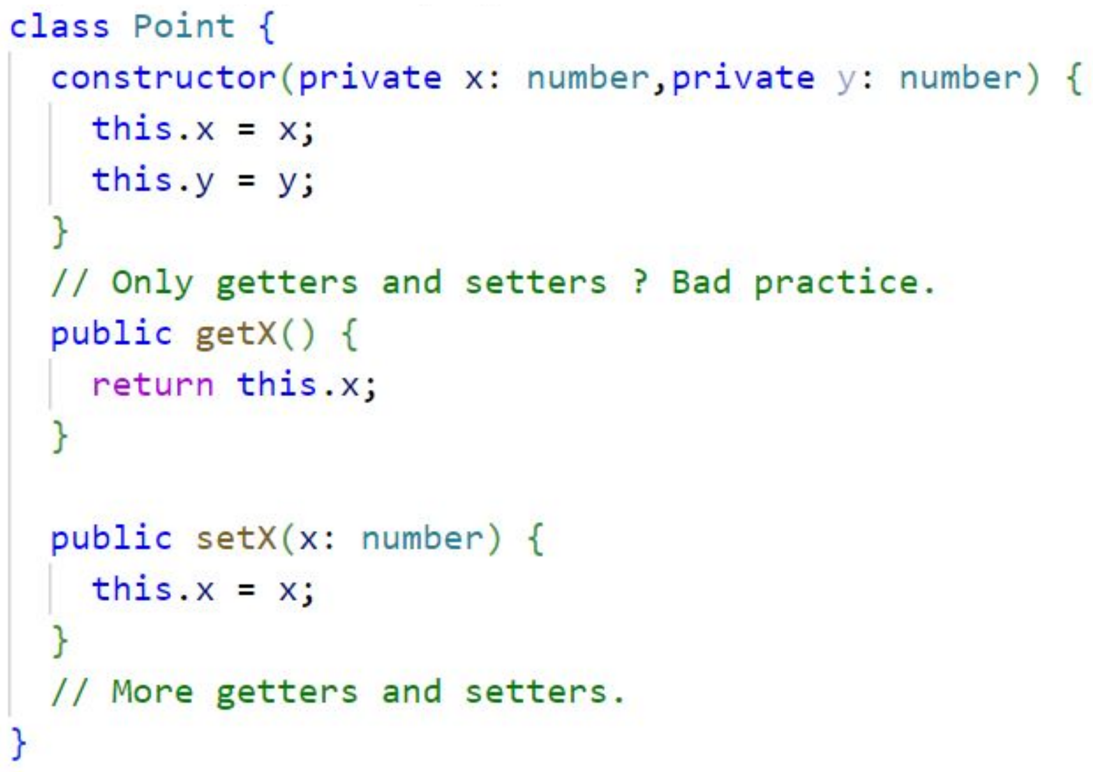
```
class Employee {
  private taxData: EmployeeTaxData;

  constructor(
    private readonly name: string,
    private readonly email: string) {
  }

  setTaxData(ssn: string, salary: number): Employee {
    this.taxData = new EmployeeTaxData(ssn, salary);
    return this;
  }

  // ...
}

class EmployeeTaxData {
  constructor(
    public readonly ssn: string,
    public readonly salary: number) {
  }

  // ...
}
```

Class with only getters and setters

- If the getters and setters are generic, using classes with no other functionality can be confusing.

- Use other kinds of data structure like structs or maps.

Class with only getters and setters



```
class Point {
  constructor(private x: number,private y: number) {
    this.x = x;
    this.y = y;
  }
  // Only getters and setters ? Bad practice.
  public getX() {
    return this.x;
  }

  public setX(x: number) {
    this.x = x;
  }
  // More getters and setters.
}
```

Class with only getters and setters ✓

```
// Good example with type literals
type Point = {
  x: number;
  y: number;
};
```

What is Clean Code?

- ”I like my code to be **elegant** and **efficient**. The logic should be straightforward…” - Bjarne Stroustrup, inventor of C++.

- “Clean code is **simple** and **direct**. Clean code reads like well-written prose. Clean code never obscures the designer’s intent…” - Grady Booch, author of Object Oriented Analysis and Design with Applications and co-author of UML.

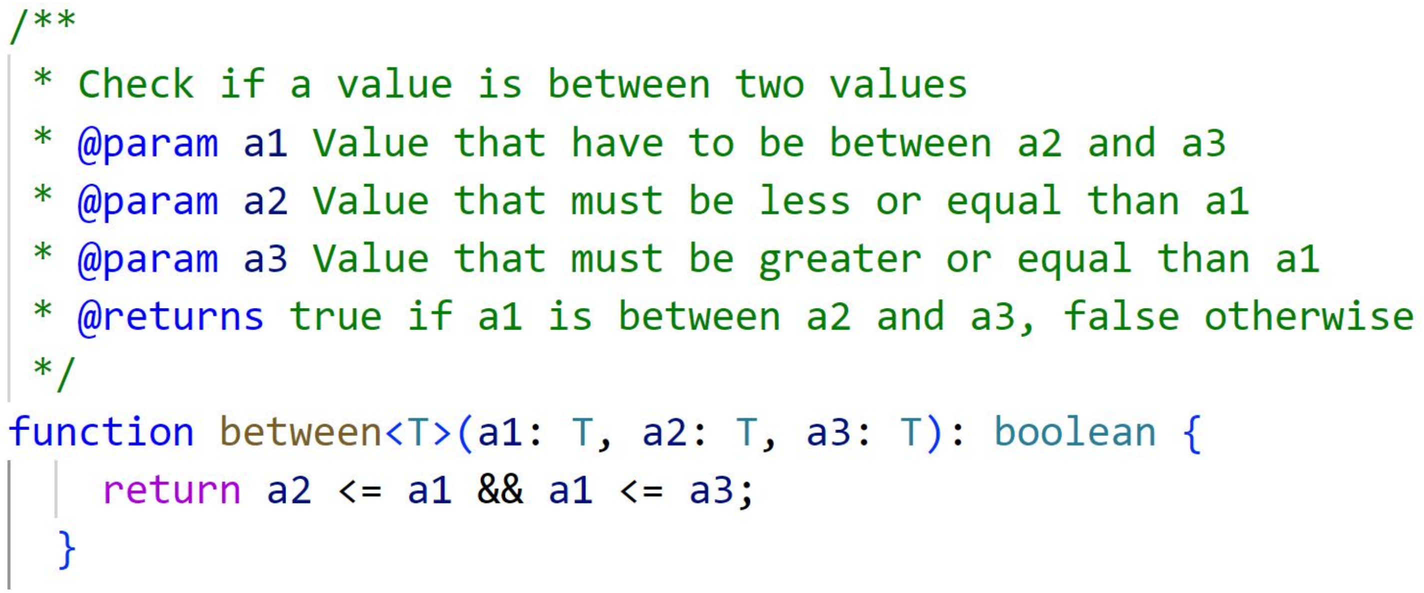
Clean code, some tips

- Names.
- Function.
- General.

Names

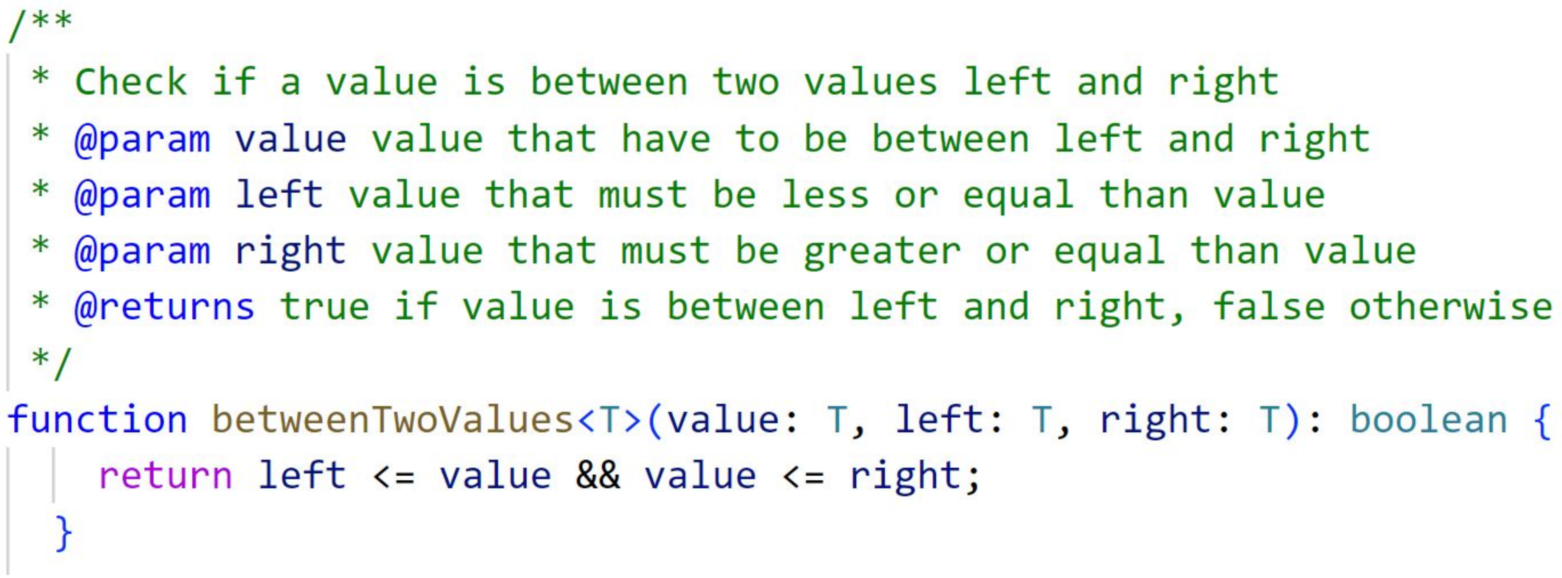
- Use meaningful variable names.
- Use explanatory variables.
- Data type in name.

Use meaningful variable names



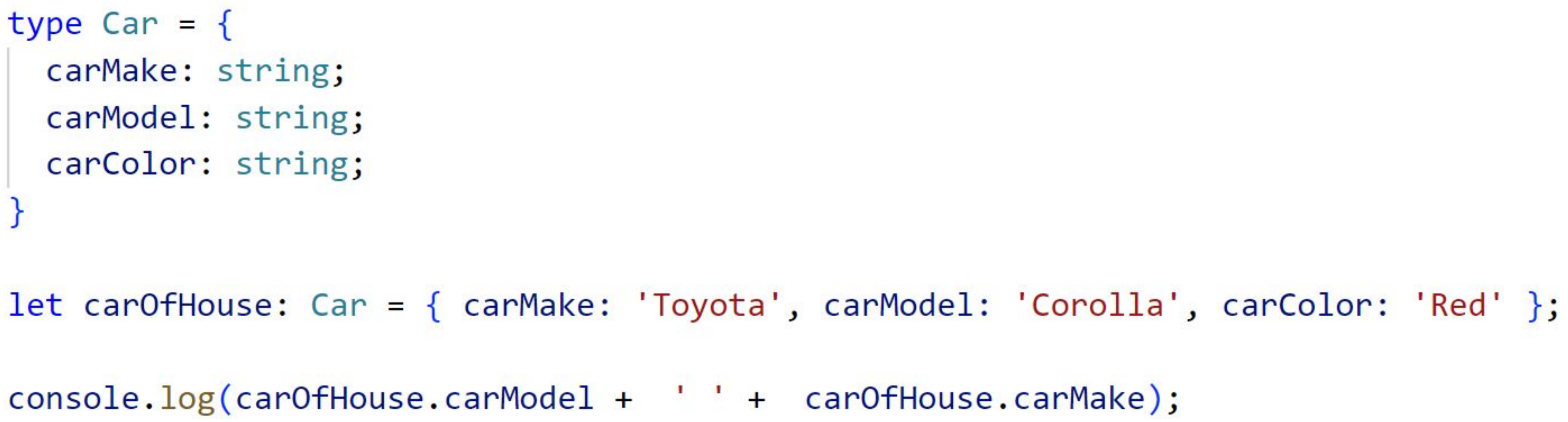
```
/**
 * Check if a value is between two values
 * @param a1 Value that have to be between a2 and a3
 * @param a2 Value that must be less or equal than a1
 * @param a3 Value that must be greater or equal than a1
 * @returns true if a1 is between a2 and a3, false otherwise
 */
function between<T>(a1: T, a2: T, a3: T): boolean {
    return a2 <= a1 && a1 <= a3;
}
```

Use meaningful variable names ✓



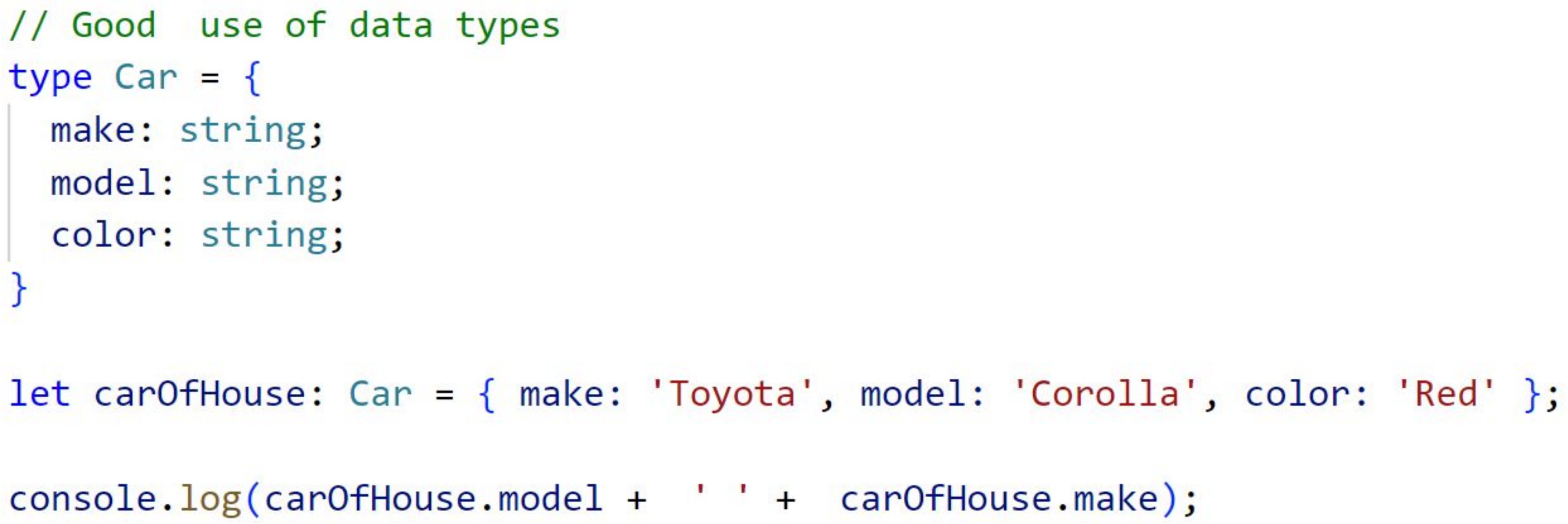
```
/**
 * Check if a value is between two values left and right
 * @param value value that have to be between left and right
 * @param left value that must be less or equal than value
 * @param right value that must be greater or equal than value
 * @returns true if value is between left and right, false otherwise
 */
function betweenTwoValues<T>(value: T, left: T, right: T): boolean {
    return left <= value && value <= right;
  }
```

Data type in the name



```
type Car = {
  carMake: string;
  carModel: string;
  carColor: string;
}

let carOfHouse: Car = { carMake: 'Toyota', carModel: 'Corolla', carColor: 'Red' };

console.log(carOfHouse.carModel +  ' ' +  carOfHouse.carMake);
```

Data type in the name



```
// Good  use of data types
type Car = {
  make: string;
  model: string;
  color: string;
}

let carOfHouse: Car = { make: 'Toyota', model: 'Corolla', color: 'Red' };

console.log(carOfHouse.model +  ' ' +  carOfHouse.make);
```

Use explanatory variables

```
declare const users: Map<string, User>;

for (const keyValue of users) {
  // iterate through users map
}
```

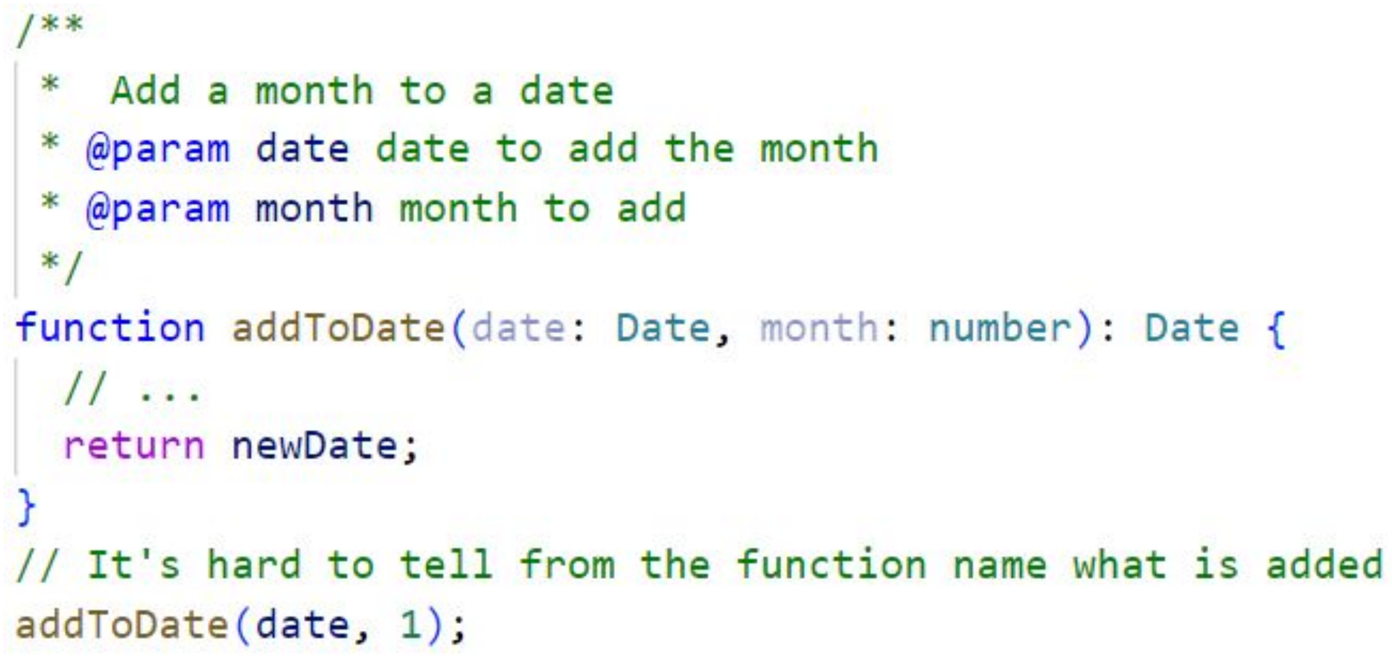
Use explanatory variables ✓

```
declare const user: Map<string, User>;

for (const [userName, user] of users) {
  // iterate through users map
}
```

Functions

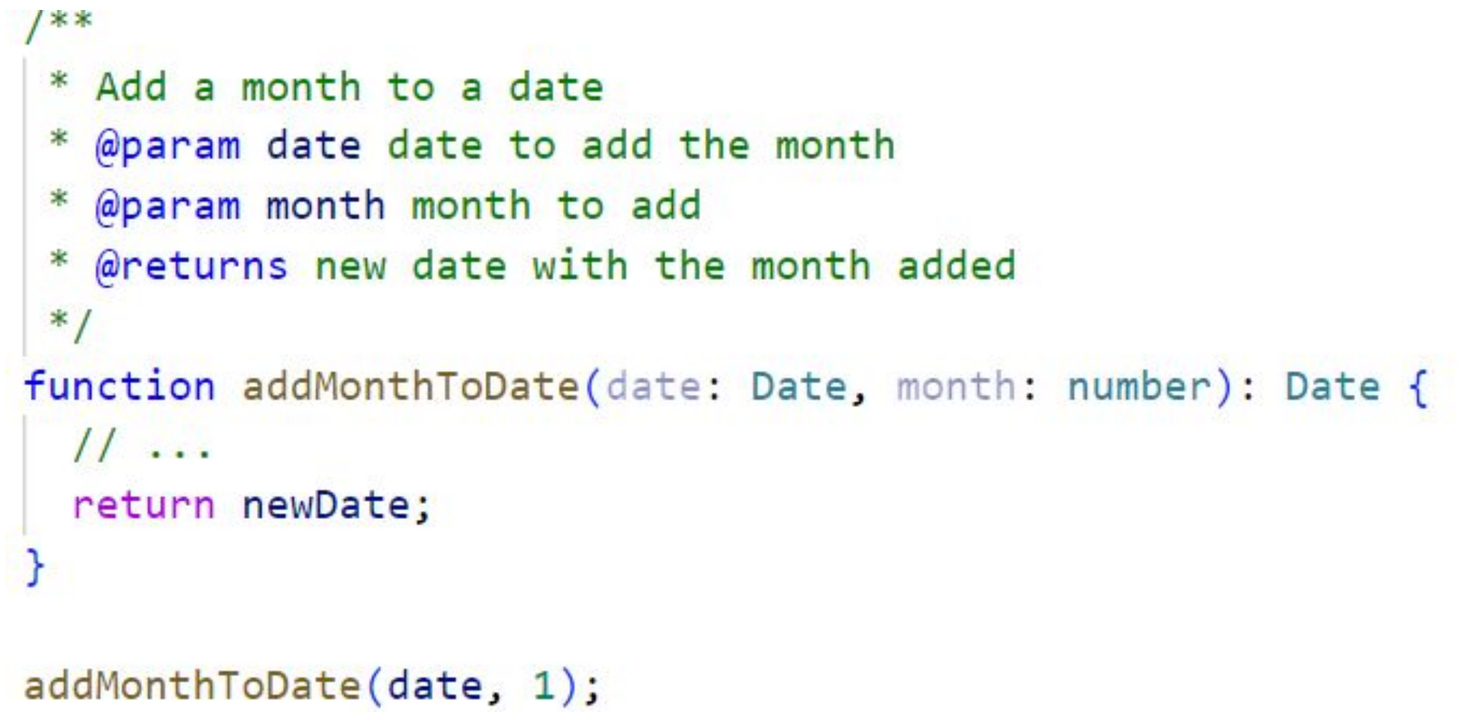
- Names should say what they do.
- Avoid negative names in boolean functions.
- Don’t use flags as function parameters.

Names should say what they do



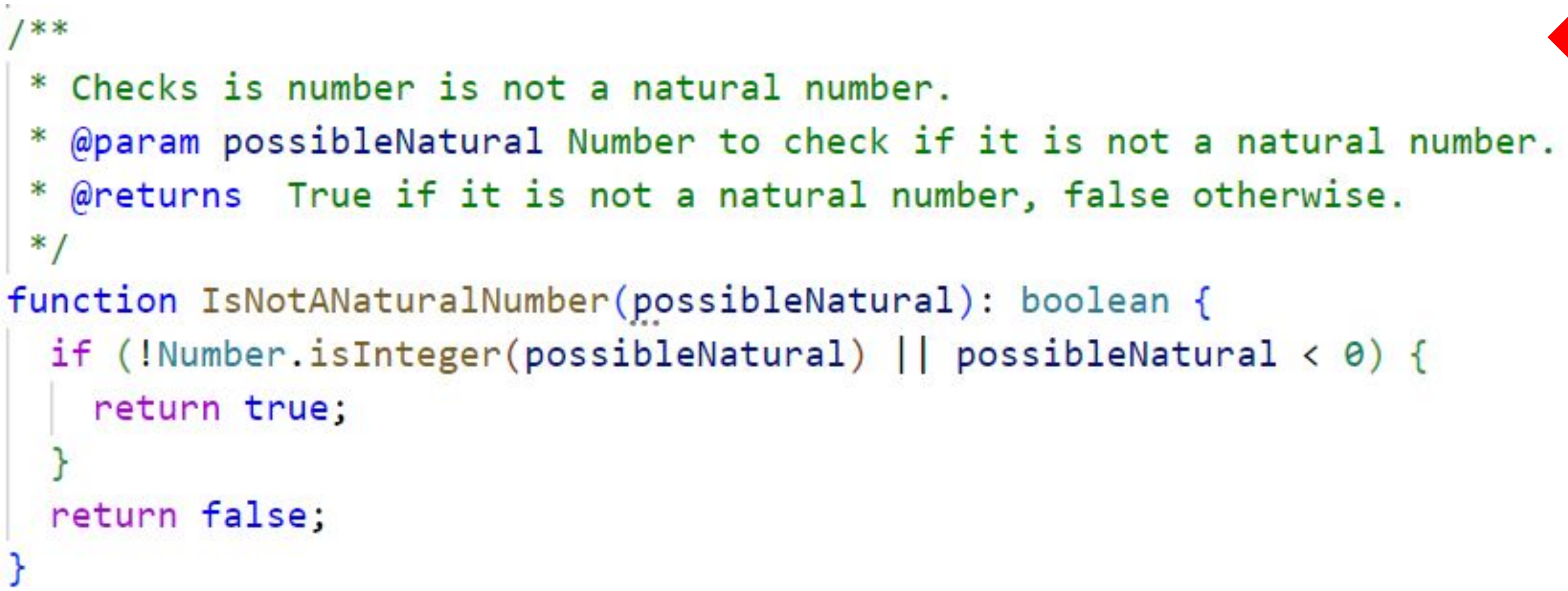
```
/**
 *  Add a month to a date
 * @param date date to add the month
 * @param month month to add
 */
function addToDate(date: Date, month: number): Date {
  // ...
  return newDate;
}
// It's hard to tell from the function name what is added
addToDate(date, 1);
```

Names should say what they do ✓



```
/**
 * Add a month to a date
 * @param date date to add the month
 * @param month month to add
 * @returns new date with the month added
 */
function addMonthToDate(date: Date, month: number): Date {
  // ...
  return newDate;
}

addMonthToDate(date, 1);
```

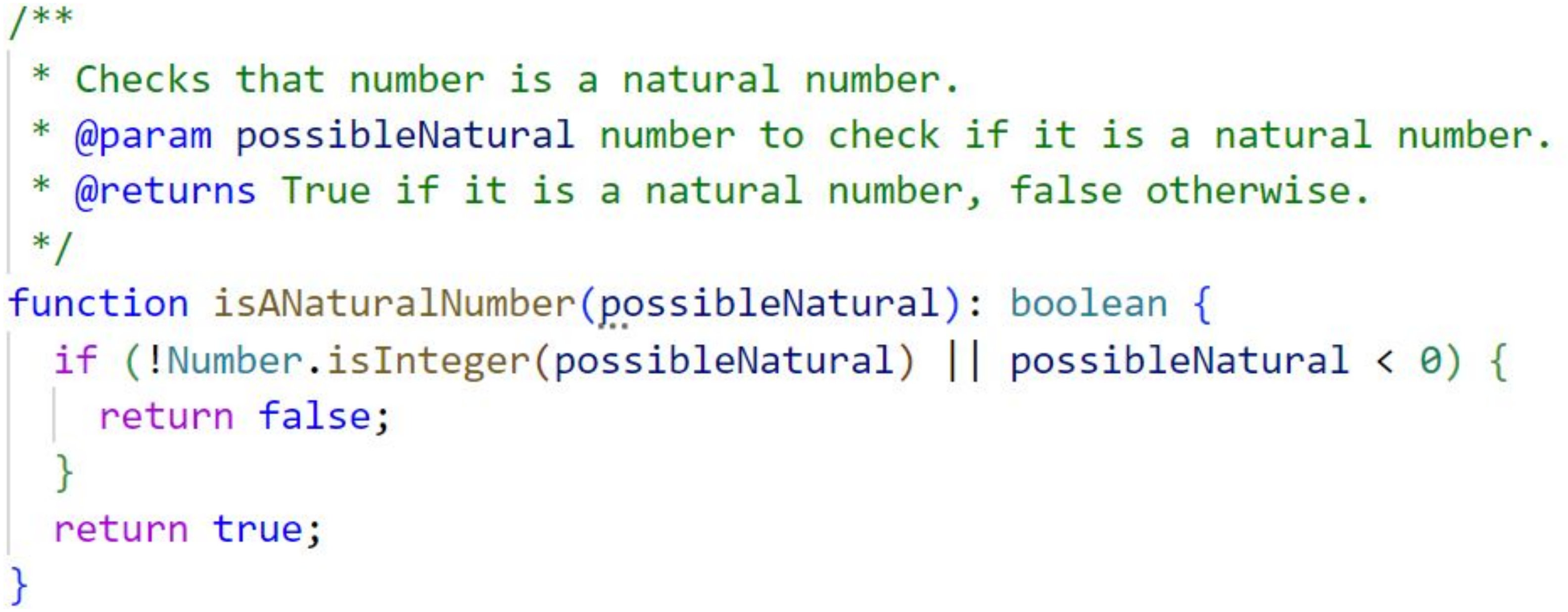
Negative names in boolean functions



```
/**
 * Checks is number is not a natural number.
 * @param possibleNatural Number to check if it is not a natural number.
 * @returns  True if it is not a natural number, false otherwise.
 */
function IsNotANaturalNumber(possibleNatural): boolean {
  if (!Number.isInteger(possibleNatural) || possibleNatural < 0) {
    return true;
  }
  return false;
}
```

Negative names in boolean functions

✓

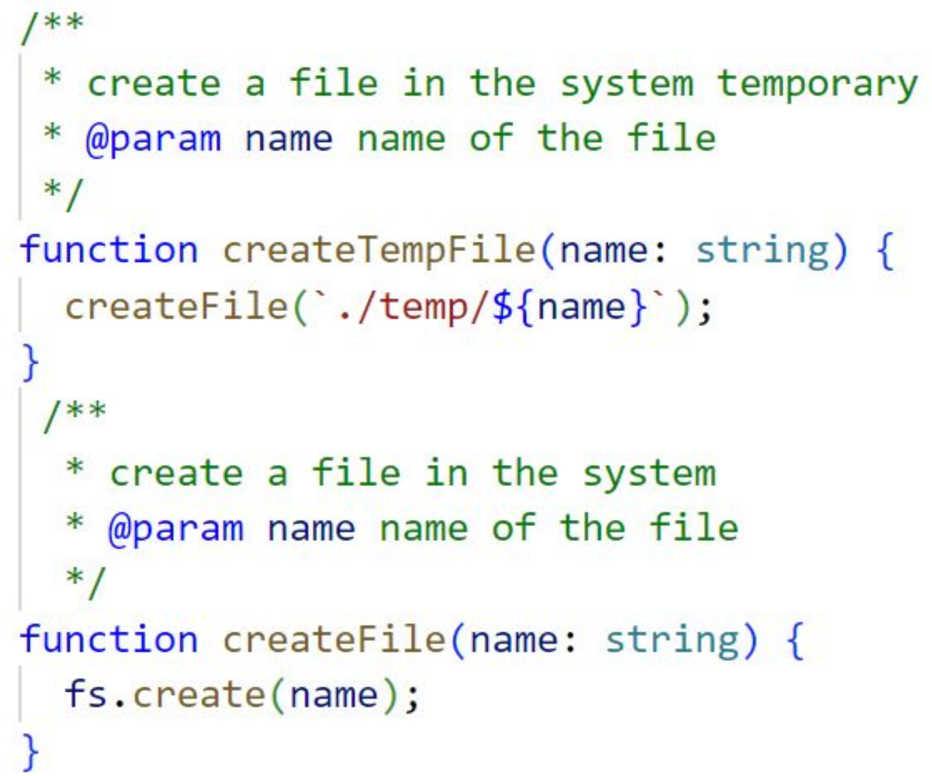


```
/**
 * Checks that number is a natural number.
 * @param possibleNatural number to check if it is a natural number.
 * @returns True if it is a natural number, false otherwise.
 */
function isANaturalNumber(possibleNatural): boolean {
  if (!Number.isInteger(possibleNatural) || possibleNatural < 0) {
    return false;
  }
  return true;
}
```

Don’t use flags as parameters

```
/**
 * create a file in the system
 * @param name string with the name of the file
 * @param temp boolean if the file is temporary or not
 */
function createFile(name: string, temp: boolean) {
  if (temp) {
    fs.create(`./temp/${name}`);
  } else {
    fs.create(name);
  }
}
```

Don’t use flags as parameters



```
/**
 * create a file in the system temporary
 * @param name name of the file
 */
function createTempFile(name: string) {
  createFile(`./temp/${name}`);
}
 /**
  * create a file in the system
  * @param name name of the file
  */
function createFile(name: string) {
  fs.create(name);
}
```

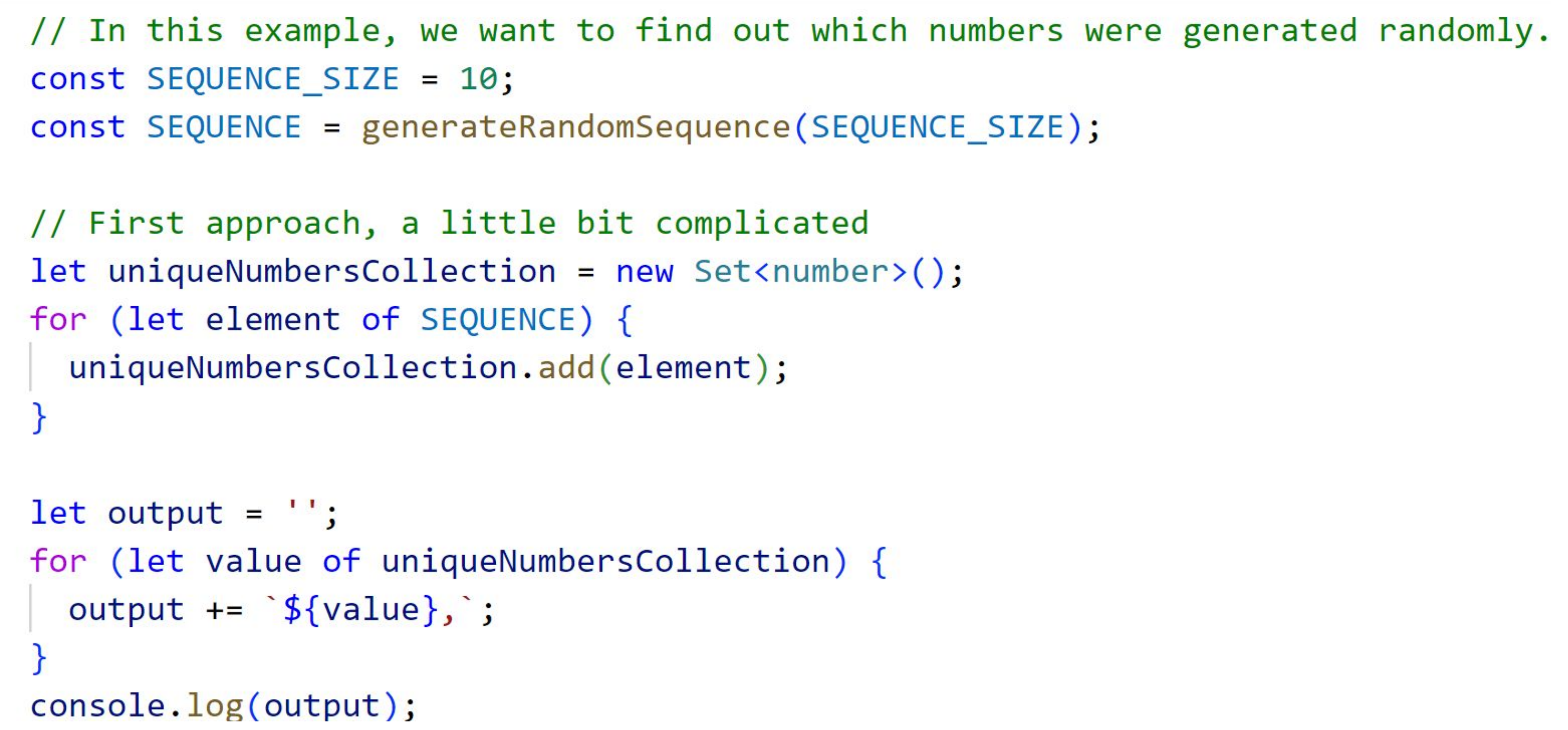
General

- KISS.
- DRY.
- YAGNI.
- Order of visibility in classes.

KISS

- Short for “**K**eep **I**t **S**imple, **S**tupid”.
- Simple systems are usually better than complex ones.
- Use built-in functions, if the problem is complex break it into smaller pieces, etc…

KISS



```
// In this example, we want to find out which numbers were generated randomly.
const SEQUENCE_SIZE = 10;
const SEQUENCE = generateRandomSequence(SEQUENCE_SIZE);

// First approach, a little bit complicated
let uniqueNumbersCollection = new Set<number>();
for (let element of SEQUENCE) {
  uniqueNumbersCollection.add(element);
}

let output = '';
for (let value of uniqueNumbersCollection) {
  output += `${value},`;
}
console.log(output);
```

KISS

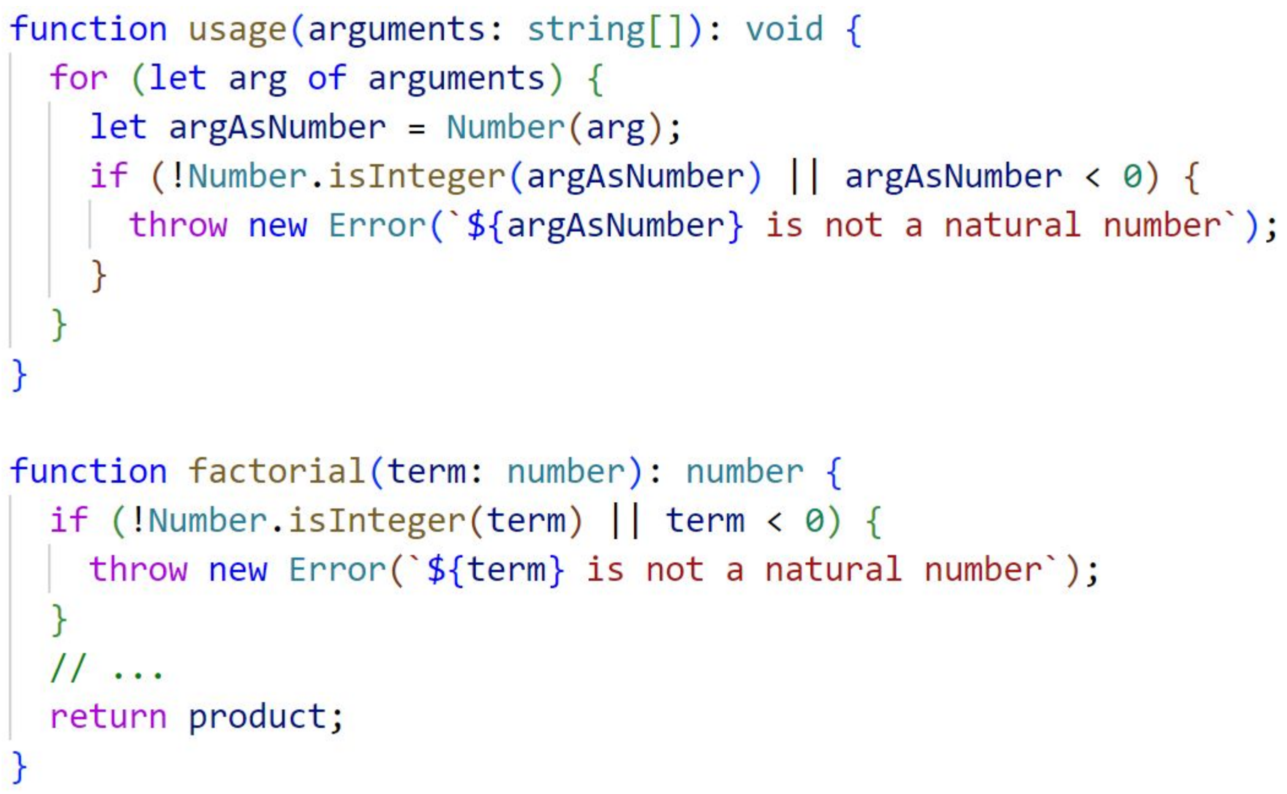
✓

```
// Second approach, simpler
let uniqueNumbersCollection2 = new Set<number>();
SEQUENCE.forEach(value => uniqueNumbersCollection2.add(value));
const SAME_OUTPUT = Array.from(uniqueNumbersCollection2).toString();
console.log(SAME_OUTPUT);
```

DRY

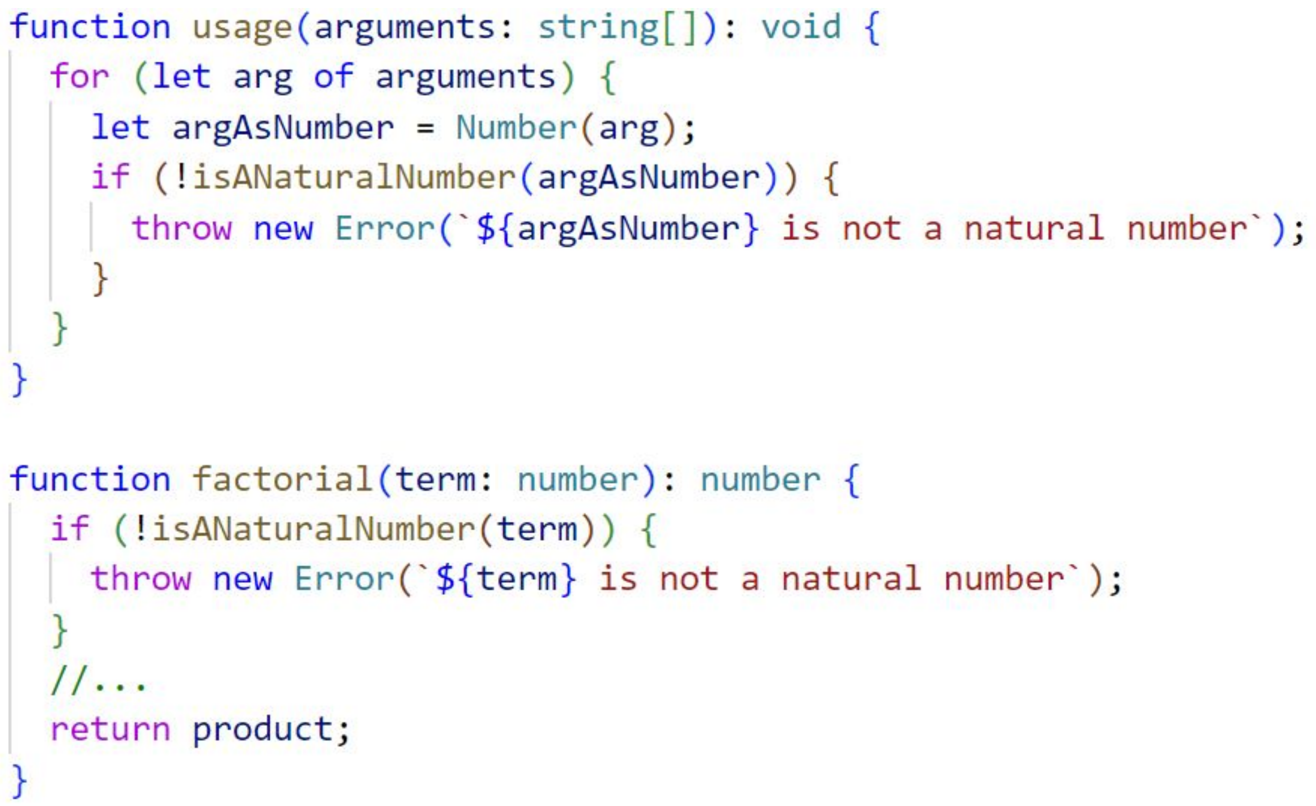
- Short for “**D**on't **R**epeat **Y**ourself”.
- It means don’t write the same code repeatedly.
- Can be a potential source of future bugs.
- Improve the maintainability of code during all phases of its lifecycle.
- Single Source Of Truth (SSOT).

DRY



```
function usage(arguments: string[]): void {
  for (let arg of arguments) {
    let argAsNumber = Number(arg);
    if (!Number.isInteger(argAsNumber) || argAsNumber < 0) {
      throw new Error(`${argAsNumber} is not a natural number`);
    }
  }
}

function factorial(term: number): number {
  if (!Number.isInteger(term) || term < 0) {
    throw new Error(`${term} is not a natural number`);
  }
  // ...
  return product;
}
```

DRY ✓

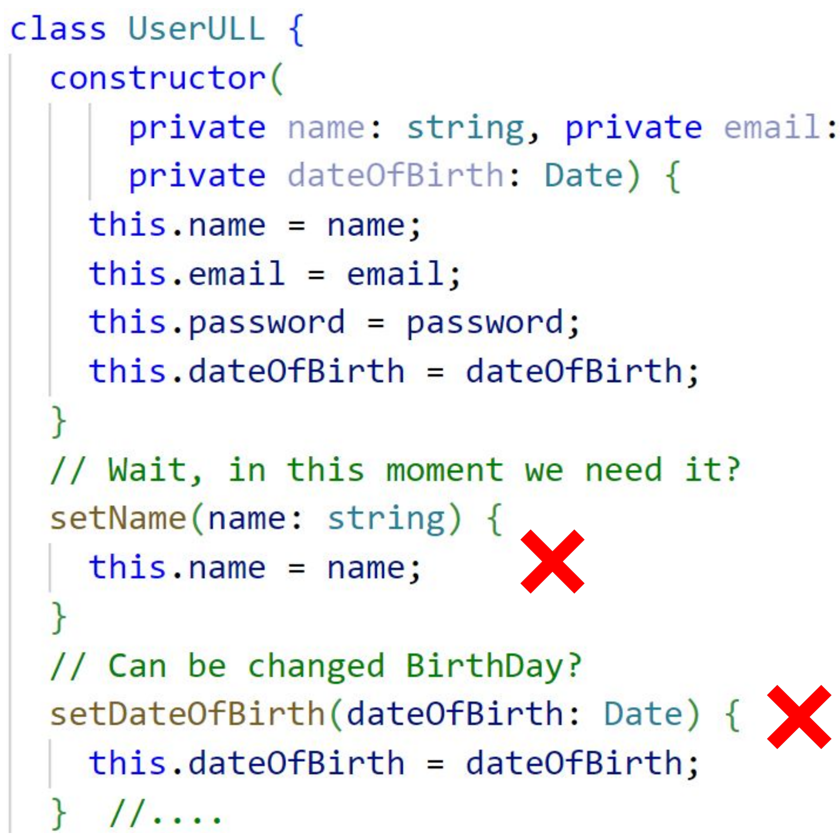


```
function usage(arguments: string[]): void {
  for (let arg of arguments) {
    let argAsNumber = Number(arg);
    if (!isANaturalNumber(argAsNumber)) {
      throw new Error(`${argAsNumber} is not a natural number`);
    }
  }
}

function factorial(term: number): number {
  if (!isANaturalNumber(term)) {
    throw new Error(`${term} is not a natural number`);
  }
  //...
  return product;
}
```

YAGNI

- Short for “**Y**ou **A**ren't **G**onna **N**eed **I**t”.
- Should not add functionality until deemed necessary.
- Spend time analyzing a testing feature.

YAGNI



```
class UserULL {
  constructor(
      private name: string, private email: string, private password: string,
      private dateOfBirth: Date) {
    this.name = name;
    this.email = email;
    this.password = password;
    this.dateOfBirth = dateOfBirth;
  }
  // Wait, in this moment we need it?
  setName(name: string) {      ❌
    this.name = name;
  }
  // Can be changed BirthDay?
  setDateOfBirth(dateOfBirth: Date) {      ❌
    this.dateOfBirth = dateOfBirth;
  }  //....
```

Order of visibility in classes

- According to "Clean Code" by Robert C. Martin, you should follow the standard Java convention:

1. Public static constants.
2. Private static constants.
3. Private instance variables (there's no public ones).
4. Public methods (constructors first).
5. Private methods.

What is Refactoring

- Systematic process of improving code without creating new functionality.

- In other words, rewriting our code.

When to refactor

- Rule of 3. If you do something 3 times, refactor.
- While adding new features (Boy Scout Rule).
- When there are bugs in the program.

Refactoring

- There’s a checklist you have to go through after refactoring:
  - Is the code cleaner?
  - Is the functionality the same?
  - Are the tests still passing?

Refactoring techniques

- **Substituting an algorithm. Steps:**

1. Simplify the existing algorithm as much as possible. If needed, use other methods.
2. Create the new algorithm in a new method and test it.
3. Test don’t pass? Compare to the old one and check for discrepancies.
4. Test pass? Delete the old algorithm for good.

Refactoring techniques

- **Extract class.**

  You have a class that does more than one thing.

1. Create a new class with the other functionality.
2. Create the relationship between the two classes.
3. Move the methods and attributes from the old to the new.
4. Think about the visibility of the new class from the outside.

Conclusions

“Rules are for the guidance of wise men and the obedience of fools” -Douglas Bader.

Conclusions

- In a best case scenario (Clean Code) a function should take no parameters.
- Functions need parameters.
- 1, 2, 3 parameters are acceptable, but more than three are… A code smell…

Conclusions

- There’s lots of code smells and good practices to learn about.
- Keep practicing and reading code to improve.
- “Rules are for the guidance of wise men and the obedience of fools” -Douglas Bader.

References

Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin.

97 Things Every Programmer Should Know by Kevlin Henney.

https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/code-smell/.
https://refactoring.guru/es/refactoring.

https://people.apache.org/~fhanik/kiss.html.

https://www.oreilly.com/library/view/97-things-every/9780596809515/ch30.html.

https://wiki.c2.com/?CodeSmell.

Questions?

Thanks for your attention!

Jose Fenic Peiteado Padilla

fenic.peiteado.20@ull.edu.es

Pablo Santana González

pablo.santana.gonzalez.15@ull.edu.es