Clean Code

This document contains a summary of the Clean Code concepts that can be commonly applied to all languages.

Please refer to a language-specific style guide for more specificity.

Written by Monisha S K (Team Quality Enablement), 2023

Naming

Naming things (= *variables, properties, functions, methods, classes*) correctly and in an understandable way is **an extremely important part of writing clean code.**

Be Descriptive

Names have **one simple purpose**: They should **describe** what's stored in a variable or property or what a function or method does. Or what kind of object will be created when instantiating a class.

If that is kept in the mind, then coming up with good names will be straightforward.
– coming up with the **good** name for a given variable/ property/ function/ ... will require some practice and often **multiple iterations**.
– Clean code is produced by iterating and improving code over time!

Naming Rules

For Variables & Properties

Variables and properties hold values of - numbers, text (strings), Boolean values, objects, lists, arrays, maps etc.

In case of constants, the name should reveal the purpose or intent of the constant,

For e.g., const MINUTES_PER_HOUR = 60; const PATH_SEPARATOR = ':';

and not const SIXTY = 60; const COLON = ':'.

Hence the **name should reveal what purpose the constant's value will be used for.**

Therefore, variables and properties should typically receive a **noun** as a name.
For example: user, product, customer, database, transaction etc.

Alternatively, we could also use a **short phrase with an adjective** - typically for storing**Boolean values**.

For example: isValid, didAuthenticate, isLoggedIn, emailExists etc.

Typically, as far as possible we **should** be more specific.

For example, prefer customer over user if the code at hand is doing customer-specificoperations with that data. This makes the code easier to be read and understood.

For Functions & Methods

Functions and methods can be called to **execute some code** i.e., they perform **tasks and operations**.

Therefore, functions and methods should typically receive a **verb** as a name.

For example:login(), createUser(), database.insert(), log() etc.

Alternatively, functions and methods can also be used to primarily produce values – then, especially when producing **booleans**, we could also go for **short phrases with adjectives**.

For example: isValid(...), isEmail(...), isEmpty(...) etc.

It typically makes sense to use more specific names.

For example: createUser() instead of just create().

For Classes

Classes are used to **create objects.**
The class name should **reflect the purpose/intent of the class to exist**. Even if it's a static class (i.e., it won't be instantiated), we will still use it as some kind of container for various pieces of data and/ or functionality - so we should then describe that container.

Good class names - just like good variable and property names - are therefore **nouns or noun phrases.**

For example: User, Product, Transaction, Payment etc.

Avoid Generic Names

In most situations, we should **avoid generic *names like handle(), process(), data, item etc.***

There can always be situations where it makes sense but typically, we should either make these names more specific (*e.g.* ***processTransaction())*** or go for a different kindof name (e.g. ***product instead of item).***

Be Consistent

An important part of using proper names is **consistency**. If we used ***fetchUsers() in*** one part of code, we should also use ***fetchProducts(***)- and ***not getProducts()***- in another part of that same code. Generally, it doesn't matter if we prefer ***fetch...(), get...(), retrieve...() or*** anyother term but we should be consistent.

The Key for the same **is to respect established conventions**, coming from the language, platform, framework being used, or make it a transparently communicated decision not to follow some convention.

Be **consistent with your domain language**, i.e., your stakeholders are all using the same words for specific things, and the same words/wording is used in the code to name classes, operations, etc...

Comments & Formatting

We usually think that comments help with code readability. The opposite is often the case though.

Proper **code formatting** (i.e., keeping lines short, adding blank lines etc.) on the otherhand **helps a lot with reading** and understanding code.

Bad Comments

There are plenty of bad comments which we can add to our code assuming it to be helpful. In the best case, "bad"means "**redundant**" in the worst case, it means "**confusing**" or even "**misleading**".

Dividers & Markers

```
// !!!!!!!
// CLASSES
// !!!!!!!

class User { ...

} class Product

{ ... }

// !!!!!!!
// MAIN
// !!!!!!!
```

Dividers and markers are redundant. If the code is written in a clean way (i.e., use proper names etc.), it's obvious what our different code parts are about.

We don't need extra markers for that. They only stop reading flow and make analyzing the code file **harder**.

Redundant Information

```
function createUser() { // creating a new user
  ...
}
```

Comments like in this example **add nothing**. Instead, we stop and spend time reading them - just to learn what we already knew because the code used proper names.

```
function build() { // creating a new user
  ...
}
```

But of course, we should **avoid such poor names** in the first place.

Commented Out Code

```
function createUser() {
  ...
}

// function createProduct() {
//   ...
// }
```

We all do that from time to time.

But we should try to **avoid commenting out code**. Instead: Just **delete** it.

When using **source control** (e.g. Git), we can always bring back old code if we need it - commented-out code just **clutters our code file** and makes going through it harder.

Misleading Comments

```
function login() { // create a new user
  ...
}
```

Probably the **worst kind of comments** are comments which mislead the reader.

Is the above function logging a user in (as the name implies) or creating a brand-newuser (as the comment implies).

We don't know - and now we must **analyze the complete function** (and any other functions it might call) to find out.

Definitely we should avoid these kinds of comments.

Good Comments

While comments don't improve our code, there are couple of comments which couldmake sense.

Legal Information

In some projects and/ or companies, we could be required to add legal information toour code files.

For example:

```
// (c) XYZ
```

Obviously, there's little we can do about that. But since the comment is right at the top ofthe code file, it's also not a big issue.

So of course, there's nothing wrong with such kinds of comments.

Warnings

Also in rare cases, warnings next to some code could make sense – for example if a unittest may take a long time to complete or if a certain functionality won't work in certain environments.

```
function fetchTestData() { ... } // requires local dev server
```

To-do Notes

Even though, we **shouldn't over-do it, adding** "To-do" notes can also be okay.

```
function login(email, password) {
  // todo: add password validation
}
```

Of course, it's better to implement a feature completely or not at all - or in incremental steps which don't require "To-do" comments - but leaving a "To-do" comment here andthere won't hurt, especially since modern IDEs help, we find these comments.

Obviously, it will not help readability (and our code in general), if we just have a bunchof "To-do" comments everywhere!

Vertical Formatting

Vertical formatting is all about using the - **well - vertical space in our code file**. So, it's all about **adding blank lines** but also about **grouping related concepts together** and **adding space between distinct concepts.**

Adding Blank Lines

Have a look at this example:

```
function login(email, password) {
  if (!email.includes('@') || password.length <
    7) {throw new Error('Invalid input!');
  }
  const user = findUserByEmail(email);
  const passwordIsValid = compareEncryptedPassword(user.password,
  password);if (passwordIsValid) {
    createSession();
  } else {
    throw new Error('Invalid credentials!');
  }
}
function signup(email, password) {
  if (!email.includes('@') || password.length <
    7) {throw new Error('Invalid input!');
  }
  const user = new User(email, password);
  user.saveToDatabase();
}
```

The above code uses good names and is not overly long - still it can be challenging todigest.

Compare this code snippet:

```
function login(email, password) {
  if (!email.includes('@') || password.length < 7) {
    throw new Error('Invalid input!');
  }

  const user = findUserByEmail(email);

  const passwordIsValid = compareEncryptedPassword(user.password, password);

  if (passwordIsValid) {
    createSession();
  } else {
    throw new Error('Invalid credentials!');
  }
}

function signup(email, password) {
  if (!email.includes('@') || password.length < 7) {
    throw new Error('Invalid input!');
  }

  const user = new User(email, password);
  user.saveToDatabase();
}
```

It's the exact same code but the extra blank lines **help with readability**.

Hence we should **add vertical spacing** to make our code cleaner.

But **where** should we add blank lines? That's related to the concept of "Vertical Density"and "Vertical Distance".

Vertical Density & Vertical Distance

Vertical density simply means that related concepts should be kept closely together.

Vertical distance means that concepts which are not closely related should be separated.

That affects both individual statements inside of a function as well as functions/ methods as a whole.

```
function signup(email, password) {
  if (!email.includes('@') || password.length < 7) {
    throw new Error('Invalid input!');
  }

  const user = new User(email, password);
  user.saveToDatabase();
}
```

Here, we have two main concepts in this function:

Validation and creating the new user inthe database. These concepts **should be separated by a blank line,** therefore.

On the other hand, creating the user object and then calling saveToDatabase()on itbelongs closely together, hence **no blank line** should be added in between.

If we had **multiple functions**, then **functions that call other functions should be keptclose together** – with blank lines in between but not on different ends of the code file.

If functions are **not directly working together** (i.e., not directly calling each other) it is okay if there is **more space** (e.g., other functions) in between.

Ordering Functions & Methods

When it comes to ordering functions and methods, it is a good practice to follow the**"stepdown rule".**

A function A which is called by function B should be (closely) **below function B - at leastif our programming language allows such an ordering.**

```
function login(email, password) {
  validate(email, passsword);
  ...
}

function validate(email, password) {...}
```

Splitting Code Across Files

If our **code file gets bigger** and/ or if we have a lot of different "things" in one file (e.g.,multiple class definitions), **it is considered a good practice to split that code across multiple files and to then use import and export statements to connect our code. This ensures that our individual code files as a whole stay readable.**

Horizontal Formatting

Horizontal formatting of course is about using the horizontal space in our code file -that primarily means that lines should be kept short and readable.

Breaking Lines into Multiple Lines

Good code should use **relatively short lines** and we should consider splitting code across multiple lines if it becomes too long.

```
const loggedInUser = email && password ? login(email, password) :
login(getValidatedEmail(), getValidatedPassword());
```

This is too long - and too much code in one line.

This snippet holds the same logic but is easier to read:

```
if (!email && !password) {
  email = getValidatedEmail();
  password = getValidatedPassword();
}
const loggedInUser = login(email, password);
```

Functions (& Methods)

Functions or methods (I don't differentiate here) are the meat of any code we write. All our code is part of some function or method after all. And we use functions to call other functions, build re-usable functionalities and more.

That's why it's extremely important to write clean functions. Functions

are made up of three main parts:

- Their name
- Their parameters (if any)
- Their body

The naming of functions and methods is covered in the "Naming" section already.

This section focuses on the parameters and body, therefore.

Minimize The Number of Parameters

The **fewer parameters a function has, the easier it is to read and call** (and the easier it is to read and understand statements where a function is called).

See this example:

```
createUser('Max', 'Max', 'test@test.com', 'testers', 31, ['Sports',
'Cooking']);
```

Calling this function is no fun. We have to remember **which parameters** are required and in **which order** they have to be provided.

Reading this code is no fun either – for example, it's not immediately clear why we have two 'Max'values in the list.

How Many Parameters Are, Okay?

Generally, **fewer is better**.

Functions **without parameters are of course very easy to read** and digest.

For example:

```
createSession();

user.save();
```

But "no parameters" is **not always an option** - after all it is the capability to take parameters that makes functions dynamic and flexible.

Functions with **one parameter** are also straightforward:

```
isValid(email);

file.write(data);
```

Functions with **two parameters can be okay –** it really depends on the context and the kind of function.

For example, **the below code should be straightforward** and easy to use and understand:

```
login('test@test.com', 'testers');

createProduct('Carpet', 12.99);
```

On the other hand, we can encounter functions where two parameters can already be confusing and it's not obvious / common sense which value should go where:

```
createSession('abc', 'temp');

sortUsers('email', 'asc');
```

Of course, modern IDEs help us with understanding the expected values and order but having to hover over these functions is an extra step which definitely **hurts code readability**.

More than two parameters should mostly be avoided - such functions can be hard to call and read:

```
createRectangle(10, 9, 30, 12);

createUser('test@test.com', 31, 'max');
```

Reducing The Number Of Parameters

What can we do if a function takes too many paramters but needs all that data?

We can replace multiple parameters with a map or an array Or, when working with OO languages, with objects.!

```
createRectangle({x: 10, y: 9, width: 30, height: 12});
```

This is much more readable!

Keep Functions Small

Besides the number of parameters, the **function body should also be kept small**.

Because a smaller body means **less code to read and understand**. But in addition, it also forces to (ideally) to write highly readable code – for example by extracting other functions which use good naming.

Consider this example:

```
function login(email, password) {
  if (!email.includes('@') || password.length < 7) {
    throw new Error('Invalid input!');
  }

  const existingUser = database.find('users', 'email', '==', email);

  if (!existingUser) {
    throw new Error('Could not find a user for the provided email.');
  }

  if (existingUser.password === password) {
    // create a session
  } else {
    throw new Error('Invalid credentials!');
  }
}
```

If we read this snippet, we will probably understand quickly what it's doing.Because it's a short, simple snippet.

Nonetheless, it'll definitely take we a few moments. Now

consider this snippet which does the same thing:

```
function login(email, password) {
  validateUserInput(email, password);

  const existingUser = findUserByEmail(email);

  existingUser.validatePassword(password);
}
```

This is way shorter and way easier to digest, right?

And that's the goal! Having short, focused functions which are **easy to read, tounderstand and to maintain**!

Do One Thing

In order to be small, **functions should just do one thing**. Exactly one thing. This

ensures that a function doesn't do too much.

But what is "one thing"?

What is "One Thing"?

The concept of functions doing "just one thing" can be confusing.

Have a look at this function:

```
function login(email, password) {
  validateUserInput(email, password);
  verifyCredentials(email, password);
  createSession();
}
```

Is this function doing one thing? We

could argue it does three things:

- Validate the user
- inputVerify the credentials
- Create a session

And of course, we would be right - it does all these things.

The idea of a function doing "one thing" is linked to another concept: The **levels of abstraction** the various operations in a function have.

A function is considered to do just one thing if all operations in the function body are on the same **level of abstraction** and **one level below** the function name.

Levels of Abstraction

Levels of abstraction can be confusing but, in the end, it's quite straightforward.

There are **high-level and low-level operations in programming** - and then a huge bandwidth **between** these two extremes.

Consider this example:

```
function connectToDatabase(uri) {
  if (uri === '') {
    console.log('Invalid URI!');
    return;
  }
  const db = new Database(uri);
  db.connect();
}
```

Calling db.connect()is a high level operation – we're not dealing with the internals ofthe programming language here, we're not establishing any network connections in great detail. We just call a function which then does a bunch of things under the hood.

console.log(...) on the other hand, just like making that **uri === ''** comparison is a lo-level operation. A higher-level equivalent would be code like this:

```
if (uriIsValid(uri)) {
  showError('Invalid URI!');
  return;
}
```

Now the implementation details are "**abstracted away**".

Low levels of abstraction aren't bad though! we just should **not mix them with higherlevel** operations since that can cause confusion and make code harder to read.

And we should try to write functions where **all operations are on the same level** of abstraction which then in turn should be exactly **one level below the function name** (i.e.,the level of abstraction implied by the function name).

Knowing these concepts is an important step towards writingclean functions.

Operations Should Be One Level Below The Function Name

In one of the above examples, we can see a couple of operations which are on the samelevel of abstraction – which then is one level below the level implied by the function name:

```
function login(email, password) {
  validateUserInput(email, password);
  verifyCredentials(email, password);
  createSession();
}
```

The loginfunction clearly wants to do all the steps that are required to log a user in. That definitely includes input validation, credential verification and then the creation of some session, token or anything like that.

And our function does exactly that!

All three operations are on the same level of abstraction (pretty high levels in this case)and one level below the function name.

Avoid Mixing Levels Of Abstraction

As mentioned above, levels of abstractions shouldn't be mixed, since that decreases readability and can cause confusion.

Consider this example:

```
function printDocument(documentPath) {
  const fsConfig = { mode: 'read', onError: 'retry' };
  const document = fileSystem.readFile(documentPath, fsConfig);
  const printer = new Printer('pdf');
  printer.print(document);
}
```

It's not a lot of code but it mixes levels of abstractions. Configuring the readFile()operation and executing all these individual steps side-by-side with the pretty high-level printing operations adds unnecessary complexity to this function.

This version is cleaner:

```
function printDocument(documentPath) {
  const document = readFromFile(documentPath);
  const printer = new Printer('pdf');
  printer.print(document);
}
```

Here, readFromFile() can take care about the exact steps that need to be performed inorder to read the document.

We need to analyze which levels we might help to improve the readability.

There are **two easy rules of thumb** I came up with, which help to decide when to split:

1. **Extract code that works on the same functionality / is closely related.**
2. **Extract code that requires more interpretation than the surrounding code**

Here's an **example for rule #1:**

```
function updateUser(userData) {
  validateUserData(userData);
  const user = findUserById(userData.id);
  user.setAge(userData.age);
  user.setName(userData.name);
  user.save();
}
```

setAge() and setName()have the same goal / functionality: They update data in the userobject. save() then confirms these changes.

We could therefore split the function:

```
function updateUser(userData) {
  validateUserData(userData);
  applyUpdate(userData);
}

function applyUpdate(userData) {
  const user = findUserById(userData.id);
  user.setAge(userData.age);
  user.setName(userData.name);
  user.save();
}
```

Just by following that rule of thumb, we implicitly removed another problem: Mixedlevels of abstraction in the original function.

Here's an **example for rule #2:**

```
function processTransaction(transaction) {
  if (transaction.type === 'UNKNOWN') {
    throw new Error('Invalid transaction type.');
  }
  if (transaction.type === 'PAYMENT') {
    processPayment(transaction);
  }
}
```

The validation for whether the transaction type is 'UNKNOWN'is of course not difficult toread but it definitely needs more interpretation from our side than just reading processPayment(...).

Hence, we could refactor this to:

```
function processTransaction(transaction) {
  validateTransaction(transaction);
  if (isPayment(transaction)) {
    processPayment(transaction);
  }
}
```

This is now all very readable, and no step requires extra interpretation from the reader's side.

Again, "behind the scenes", we got rid of mixed levels of abstraction and a too big distance between the level implied by the function name and some code in that function.

Warning: **Split Functions Reasonably**

With all these rules, and because we of course definitely don't want to write bad code,we can get into a habit of extracting everything into new functions.

How do we know that an extraction doesn't make sense?There

are **three main signals:**

1. We’re just **renaming the operation.**
2. We suddenly need to **scroll way more**, just to follow the line of thought of a simple function.
3. We **can't come up with a reasonable name** for the extracted function, which **hasn't already been taken.**

Control Structures

No matter which kind of application we're building - we will most likely use control structures in our code: **if statements**, **for loops**, maybe also **while loops** or **switch-case** statements.

Control structures are extremely important to coordinate code flow and of course we should use them.

But control structures **can also lead to bad or suboptimal code** and hence play an important role when it comes to writing clean code.

There are three main areas of improvement, we should be aware of:

1. **Prefer positive checks.**
2. **Avoid deep nesting.**
3. **Embrace errors.**

Side-note: *"Avoid deep nesting" is heavily related to clean code practices we already knowfrom writing **clean functions** in general. Still, there are some control-structure specific concepts, that are covered in this document and course section.*

Prefer Positive Checks

This is a simple one. It can make sense to use positive wording in **if** checks instead of negative wording.

Consider this example:

```
if (isEmpty(blogContent)) {
  // throw error
}

if (!hasContent(blogContent)) {
  // throw error
}
```

The first snippet is quite readable and requires zero thinking.
The second snippet uses the '!' operator to check for the opposite - slightly more thinking and interpretation is required from the reader.

Hence option #1 is preferrable.

However, sometimes, the negative version adds more sense:

```
if (!isOpen(transaction)) {
  // throw error
}

if (isClosed(transaction)) {
  // throw error
}
```

On first sight, it looks like option #2 is better.

And it generally might be. But what if we didn't just have 'Open' and 'Closed' transactions? What if we also had 'Unknown'?

```
if (!isOpen(transaction)) {
  // throw error
}

if (isClosed(transaction) || isUnknown(transaction)) {
  // throw error
}
```

This quickly adds up! The more possible options we have, the more checks we need tocombine.

Or we simply check for the opposite - in this example, simply for the transaction NOT being open.

Avoid Deep Nesting

This is very important! We should **absolutely avoid deeply nested control structures.** since such code is highly unreadable, hard to maintain and often error prone.

There are a couple of techniques that can help with getting rid of deeply nested control structures and code duplication:

1. **Use guards and fail fast.**
2. **Extract control structures and logic into separate functions**
3. **Polymorphism & Factory Functions**
4. **Replace if checks with errors.**

Use Guards & Fail Fast

Guards are a great concept! Often, we can extract a nested **if** check and **move it right tothe start of a function** to **fail fast** if some condition is (not) met and only continue with the rest of the code otherwise.

Here's an example with a guard and failing fast:

```
function messageUser(user, message) {
  if (!user || !message || !user.acceptsMessages) {
    return;
  }
  user.sendMessage(message);
  if (success) {
    console.log('Message sent!');
  }
}
```

By extracting and combining conditions, three **if** checks could be merged into one which leads to the function not continuing **if** one condition fails.

Guards are these **if** checks right at the start of our functions.

We simply take nested checks, **invert the logic** (i.e., check for the user **not** accepting messages etc.) and then return or throw an error to **avoid that the rest of the function executes**.

Extract Control Structures & Logic Into New Functions

We already learned that splitting functions and keeping functions small is important.Applying this knowledge is always great, it also helps with removing deeply nested control structures.

Polymorphism & Factory Functions

Sometimes, we end up with duplicated **if** statements and duplicated checks just because the code inside of these statements differs slightly.

In such cases, **polymorphism** and **factory functions** can help.

Before we dive into these concepts, let us have a look at this example:

Consider this example:

```
function processTransaction(transaction) {
  if (isPayment(transaction)) {
    if (usesCreditCard(transaction)) {
      processCreditCardPayment(transaction);
    }
    if (usesPayPal(transaction)) {
      processPayPalPayment(transaction);
    }
  } else {
    if (usesCreditCard(transaction)) {
      processCreditCardRefund(transaction);
    }
    if (usesPayPal(transaction)) {
      processPayPalRefund(transaction);
    }
  }
}
```

In this example, we repeat the usesCreditCard() and usesPayPal() checks because we run different code depending on whether we have payment or refund.

We can solve this by writing a **factory function which returns a polymorphic object:**

```
function getProcessors(transaction) {
  let processors = {
    processPayment: null,
    processRefund: null
    };


   if (usesCreditCard(transaction)) {
     processors.processPayment =
     processCreditCardPayment;processors.processRefund =
     processCreditCardRefund;
   }
   if (usesPayPal(transaction)) {
     processors.processPayment =
     processPayPalPayment;processors.processRefund =
     processPayPalRefund;
   }
}
 function processTransaction(transaction)  {
   const processors =
   getProcessors(transaction);if
   (isPayment(transaction)) {
     processors.processPayment(transaction);
   } else {
     processors.processRefund(transaction
     );
   }
}
```

The repeated checks for whether a credit card or PayPal was used was now outsourced into the getProcessors()function which now only runs these checks once (instead oftwice, as before).

getProcessors()is a **factory function**. It produces objects - and that's the definition of a factory function: **A function that produces objects**.

The getProcessors()function returns an object with two functions stored inside –functions which oure **NOT executed yet** (note, that the **() are missing** after processCreditCardPaymentetc.).

The object returned by getProcessors() is **polymorphic** because we always use it in the same way (we can call processPayment() and processRefund()) but the **logic that will be executed is not always the same**.

Side-note*: There are multiple ways to solve problem. The same problem can also be solved by using classes and inheritance.*

Embrace Errors

Errors are another nice way of getting rid of redundant **if** checks. They allow us to utilize mechanisms built into the programming language to handle problems in the place where they should be handled (and cause them in the place where they should be caused...).

Consider this example:

```
function createUser(email, password) {
  const inputValidity = validateInput(email, password);

  if (inputValidity.code === 1 || inputValidity === 2) {
    console.log(inputValidity.message);
    return;
  }
  // ... continue
}

function validateInput(email, password) {
  if (!email.includes('@') || password.length < 7) {
    return { code: 1, message: 'Invalid input' };
  }
  const existingUser = findUserByEmail(email);
  if (existingUser) {
    return { code: 2, message: 'Email is already in use!' };
  }
}
```

Here, the **validateInput()** function does not directly log to the console and also not just return true or false. Instead, it returns an object / map with more information about the validation result. This is not an unrealistic scenario, but it's implemented in a suboptimal way.

In the end, the example code produces a "synthetic error". But because it's synthetic, we can't handle it with normal error handling tools, instead, if checks are used.

Here's a better version - embracing built-in error support which pretty much all programming languages offer:

```
function createUser(email,
  password) {try {
    validateInput(email, password);
  } catch (error) {
    console.log(error.
    message);
  }
  // ... continue
}

function validateInput(email, password) {
  if (!email.includes('@') ||
    password.length < 7) {throw new
    Error('Input is invalid!');
  }
  const existingUser =
  findUserByEmail(email);if
  (existingUser) {
    throw new Error('Email is already taken!');
  }
  }
```

Throw is a keyword in JavaScript (and many other languages) which can be used togenerate an error.

Once an error is "on its way", it'll bubble up through the entire call stack and **cancel anyfunction execution until it's handled** (via **try-catch**).

This removes the need for extra **if** checks and return statements.

And we could even move the entire error handling logic into a separate func

Indeed, **error handling should typically be "one thing"** (remember: functions should do one thing), so moving it up into a separate function is a good idea.

Classes, Objects & Data Containers

When it comes to working with classes and objects and writing clean classes and objects,there are a **couple of rules and concepts we should be aware of**.

1. We can **differentiate between objects and data containers / data structures.**
2. **Consider using Polymorphism!**
3. **Classes should be small.**
4. **Classes should have a high cohesion.**
5. **Respect the "Law of Demeter"**
6. **Write SOLID classes.**

Objects vs Data Containers / Data Structures

Classes are blueprints for objects.

There also are programming languages where we can create objects without (actively)using classes – for example JavaScript. And there are languages where we MUST use classes for pretty much everything (e.g. Java).

Either way, objects allow us to **group related data** (properties) and **functionalities** (methods) **together**. And objects typically expose a public API of methods which can beused anywhere in our code to interact with these objects.

For example:

```
customer.message(someMessage);
```

Even though we're technically always dealing with objects, we can differentiate between"**real objects**" (i.e. used as objects with a public API) and mere **data containers** (or data structures, though that term is also used in different ways).

A **data container is really just that - an object which holds a bunch of data**. These are often **called as records or structure**.Here's an example:

```
class UserData {
  public name: string;
  public age: number;
}

const userData = new UserData();
userData.name = 'Max';
userData.age = 31;
```

This class has no methods and both properties are exposed publicly.

An **object on the other hand hides its data from the public and instead exposes a publicAPI** in the form of methods:

```
class User {
  private name: string;
  private age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hi! I'm ${this.name} and I'm ${this.age} years old.`);
  }
}

const userData = new UserData('Max', 31);
userData.greet();
```

Both are absolutely valid types of objects - it's just important to use the **right kind of object for** the right job. And we should **avoid mixing the types.**

Why does this matter?

We **shouldn't try to access some internals of an object**. This bares the danger of the object changing these internals and our code breaking because of that.

In addition, it makes code harder to read, if we have to interpret the meaning and values of properties of other classes.

Calling something like greet ()on the other hand is straightforward.

Of course, when working with a data container in a place where a data container is needed, this is fine:

```
function validateInput(credentials) {
  if (!isEmail(credentials.email) || !isPassword(credentials.password))   {
    // ...
  }
}
```

In this example, it's clear that **credentials** just holds some data which we can extract and work with.

Polymorphism

Polymorphism is a fancy term but in the end, it just means that we re-use code (e.g. callthe same method) **multiple times** but that the code will do **different things**, depending on the object type.

That's the idea behind polymorphism!

.

Classes Should Be Small

Just like functions, classes should be **small**. Also, just like functions, classes should kind of "do one thing", though here "one thing" is not a single task but **instead a well-defined set of responsibilities (maybe one) exclusively assigned and taken care of by this one class.**

For Ex: A **User** class might have a **login**() method. It might have a couple of other methods that do user-typical things but a **refund**() method would be strange.

refund() sounds more like a payment-related method so it would make more sense in a Payment class for example.

Hence a User class which also handles payments would be too big - even if it would only be made up of a couple of lines of code.

The **size of a class is therefore defined by its number of responsibilities**. And clean classes should **only have one responsibility.**

The way to approach it would be by asking few questions like

- Why does the class User exist?
- What should it be doing?
- How is it going to be used? What for?

For e.g., if we're talking about some web application that allows some user to login and this is designed to just be some model for a user, maybe holding some real name, nickname, email address, then there might be some operations acting on this data, but the login and logout operations are probably realized completely differently, e.g. by having some services (along the use cases) for logging in and logging out some user.

Classes & Cohesion

Clean classes have a high cohesion.

But what is "cohesion"?

Cohesion basically describes **the amount by which methods and attributes of a class belong to the same (defining) responsibility of that class.**

If a class has 2 properties and 3 methods and every method uses both properties, this class will have the **highest possible cohesion**.

If the same class would look such that no method uses the properties, the class would have **no cohesion** – the properties would be worthless for the methods.

Low cohesion is a clear sign for a class that should may just be a data container / dataStructure

We will rarely achieve maximum cohesion, **but high cohesion should be our goal.**

Once we note that cohesion decreases, it might be time to **split a class into smaller,more focused classes and smaller classes.**

The Law of Demeter

When working with objects, the following code is considered to be bad / not clean:

```
this.customer.lastPurchase.date;
```

This code violates the **Law of Demeter** which states that we shouldn't access the internals of another object through another object.

This is also called the "**principle of least knowledge**".

Code in a method may only access direct internals (properties and methods) of the object

- it **belongs to**
- objects that are **stored in properties of that object.**
- objects which are **received as method parameters.**
- objects which are **created in the method**.

If we are only working with a couple of **data containers**, we would **not be violating the"Law of Demeter"** when chaining their various properties together. Because the entire idea behind data containers **IS** that they expose their properties publicly – they got nothing else besides that!

SOLID Classes

When it comes to creating clean classes, there are a couple of helpful rules, laws and concepts (e.g. "Law of Demeter", see above). But especially the SOLID principles are often cited.

Below, all five principles are explained and put into the context of writing clean code.

S: Single-Responsibility Principle (SRP)

> *A class should only have a single responsibility - it should only change for this one responsibility.*

The SRP simply states that a class should be **focused on one core responsibility**. Only if that responsibility requires changes to the class code, such changes are acceptable.

If a class needs to change because of different responsibilities, it's too big and should be **split into multiple smaller classes.**

The SRP is an **important principle when it comes to writing clean code**. Because it typically leads to smaller and more **focused classes therefore more readable–** which is in line with what we want for our classes from a clean code perspective!

O: Open-Closed Principle (OCP)

> *A class should be open for extension but closed for modification.*

Once we decided how a class should look like (which public API / methods it has), we should **"close" the class for modification -** i.e., the code doesn't change anymore.

Of course, that's not true though. We still **should continue working** on the class and especially fix errors.

But we should **not** edit the class all the time, just to handle certain new features or -worse – variants of features.

We can use OOPS concept like Polymorphism or Inheritance where we can extend and add new subclasses The OCP is an **important principle when it comes to writing clean code**. Because it typically leads to smaller and more focused classes – which is in line with what we wantfor our classes from a clean code perspective!

L: Liskov Substitution Principle (LSP)

> *Objects in a program should be replaceable with instances of their subtypes.*

This means that the child class must implement everything that's in the parent class. The parent class serves the class has the base members that child classes extend from.

For example, if we want to implement classes for a bunch of shapes, we can have a parent Shape class, which are extended by all classes by implementing everything in the Shape class.

Any inheritance model that adheres to the Liskov Substitution Principle **will implicitly follow the Open/Closed principle and helps us model good inheritance hierarchies** thereby increasing code understandability.

I: Interface Seggragation Principle (ISP)

> *Many client-specific interfaces are better than one general-purpose interface*

Interfaces are basically contracts which force implementing classes to implement certain behaviors (methods and properties).

This means that **we shouldn't impose the implementation of something if it's not needed**.

D: Dependecy Inversion Principle (DIP)

> *One should depend upon abstractions, not concretions.*

This principle states that high-level modules shouldn't depend on low-level modules and they both should depend on abstractions, and abstractions shouldn't depend upon details. Details should depend upon abstractions.

This means that we shouldn’t have to know any **implementation details of our dependencies.** If we do, then we violated this principle.

We need this principle because if we do have to reference the code for the implementation details of a dependency to use it, then when the dependency changes, there’s going to be lots of breaking changes to our own code.

As software gets more complex, if we don’t follow this principle, **then our code will break a lot.**

Use Common Sense

It's easy to treat every tiny step as a single responsibility and single thing and if we do that, we would end up with projects with 100s of classes that all contain only one method.

This is NOT the goal and definitely NOT clean.

*****Thank You *****

References

1. https://yourlearning.ibm.com/activity/UDEMY-3611296

2. https://sayansingha.medium.com/the-s-o-l-i-d-principles-clean-code-9fcca658dab5