Clean Code

This document contains a summary of the concepts taught throughout the course. Use it in conjunction with the course lectures.

Naming

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

Indeed - if poor names are chosen - pretty much all other concepts taught throughout the course will **not help that much**.

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 you keep that in mind, coming up with good names should actually be straightforward - though coming up with the **best** name for a given variable/ property/ function/ ... will of course still require some practice and often **multiple iterations**. That's normal though - clean code is written by iterating and improving code over time!

Naming Rules

Variables & Properties

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

Hence **the name should imply which kind of data is being stored**.

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

Alternatively, you could also use a **short phrase with an adjective** - typically for storing **boolean values**. For example: `isValid`, `didAuthenticate`, `isLoggedIn`, `emailExists` etc.

Typically, if you can be more specific, you **should** be more specific.

For example, prefer `customer` over `user` if the code at hand is doing customer-specific operations with that data. This makes your code easier to read and understand.

Functions & Methods

Functions and methods can be called to then **execute some code**. That means that 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**, you could also go for **short phrases with adjectives**. For example: `isValid(...)`, `isEmail(...)`, `isEmpty(...)` etc.

You should try to **avoid** names like `email()`, `user()` etc. These names sound like properties. Prefer `getEmail()` etc. instead.

As with variables and properties, if you can **be more specific**, it typically makes sense to use such more specific names. For example: `createUser()` instead of just `create()`.

Classes

Classes are used to **create objects** (unless it's a static class).

Hence the class name should **describe the kind of object it will create**. Even if it's a static class (i.e. it won't be instantiated), you will still use it as some kind of container for various pieces of data and/ or functionality - so you should then describe that container.

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

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

Avoid Generic Names

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

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

Be Consistent

An important part of using proper names is **consistency**.

If you used `fetchUsers()` in one part of your code, you should also use `fetchProducts()` - and not `getProducts()` - in another part of that same code.

Generally, it doesn't matter if you prefer `fetch...()`, `get...()`, `retrieve...()` or any other term but you should be consistent!

Comments & Formatting

You could think that comments help with code readability. In reality, the **opposite is often the case** though.

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

Bad Comments

There are plenty of bad comments which you can add to your code. 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
// !!!!!!!

const user = new User(...);
```

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

You don't need extra markers for that. They only stop your 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, you stop and spend time reading them - just to learn what you already knew because the code used proper names.

This command could be useful if you had **poor naming**:

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

But of course you 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 you should try to **avoid commenting out code**. Instead: Just **delete** it.

When using **source control** (e.g. Git), you can always bring back old code if you need it - commented-out code just **clutters your 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 actually **mislead** the reader.

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

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

Definitely avoid these kinds of comments.

Good Comments

Whilst comments don't improve your code, there are couple of comments which could make sense.

Legal Information

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

For example:

```
// (c) Academind GmbH
```

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

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

"Required" Explanations

In rare cases, adding extra explanations next to your code does help - **even if you named everything properly**.

A good example would be regular expressions:

```
# Min. 8 characters, at least: one letter, one number, one special character
const passwordRegex = /^(?=.*[A-Za-z])(?=.*\d)(?=.*[@$!%*#?&])[A-Za-z\d@$!%*#?
&]{8,}$/
```

Even though the name `passwordRegex` tells us that this regular expression will be used to validate passwords, it's not clear immediately, which specific rule will apply.

Regular expressions are not that easy to read, hence adding that extra comment doesn't hurt.

Warnings

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

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

Todo Notes

Even though, you shouldn't over-do it, adding "Todo" 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 "Todo" comments - but leaving a "Todo" comment here and there won't hurt, especially since modern IDEs help you find these comments.

Obviously, it will not help readability (and your code in general), if you just have a bunch of "Todo" comments everywhere!

Vertical Formatting

Vertical formatting is all about using the - well - vertical space in your code file. So it's all about adding blank lines but also about grouping related concepts together and adding space between distant 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 to digest. 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 you should **add vertical spacing** to make your code cleaner.

But **where** should you 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 in the database.

These concepts **should be separated by a blank line** therefore.

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

If we had **multiple functions**, then **functions that call other functions should be kept close 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 least if your programming language allows such an ordering.

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

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

Splitting Code Across Files

If your **code file gets bigger** and/ or if you 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 your code. This ensures that your individual code files as a whole stay readable.

Horizontal Formatting

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

Breaking Lines Into Multiple Lines

Nowadays, you can of course fit extremely long lines onto your screen - at least if you can read small text/ fonts.

But just because it fits into a line technically, doesn't mean that it's good code.

Good code should use **relatively short lines** and you 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);
```

Using Short Names

Names should be descriptive - you learned that.

But you shouldn't waste space and make them harder to read by **being too specific**.

Consider this name:

```
const loggedInUserAuthenticatedByEmailAndPassword = ...
```

This is way too specific! `loggedInUser` would suffice!

Functions (& Methods)

Functions and 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 cheat sheet 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 paramters 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.

Thankfully, 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, **this code should be straightforward** and easy to use and understand:

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

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

On the other hand, you 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 you 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 you do if a function takes too many paramters but needs all that data?

You can replace multiple parameters with a map or an array!

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

This is much more readable!

Keep Functions Small

Besides the number of paramters, 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 you (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 you read this snippet, you will probably understand pretty quickly what it's doing. Because it's a short, simple snippet.

Nonetheless, it'll definitely take you 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, to understand 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?

You could argue it does three things:

- Validate the user input
- Verify the credentials
- Create a session

And of course you 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 concept.

There **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 of the programming language here, we're not establish 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 lower 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! You just should **not mix them with higher level** operations since that can cause confusion and make code harder to read.

And you 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 ob abstraction implied by the function name).

Of course getting this right can be tricky and requires experience (and you'll still not always get it right). But knowing these concepts is an important step towards writing clean 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 same level 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 `login` function 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.

Of course, the line is blurry though.

What about this slightly altered example?

```
function login(email, password) {
  if (inputInvalid(email, password)) {
    showError(email, password);
    return;
  }
  verifyCredentials(email, password);
  createSession();
}
```

Here, we still got relatively high levels of abstraction but we can definitely argue whether all operations are on the same level. `verifyCredentials(...)` seems to be more high-level than doing the if check and caring about the error message manually.

In addition, whilst validation belongs to the tasks kicked of by `login()`, we can question whether `showError(...)` should be called directly inside of `login()`. It seems to be more than one level below the `login()` instruction.

Obviously, this always leaves **room for discussion and interpretation**.

And more granularity also isn't always better (see further down below, "Split Functions Reasonably").

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 in order to read the document.

Of course, you could argue, that this could be split up even more:

```
function printDocument(documentPath) {
  const document = readFromFile(documentPath);
  printFile(document);
}
```

But this new `printFile()` function almost just rephrases the `printDocument` function. So this split could be done but it might not always be a great decision to make it (see below, "Split Functions Reasonably").

Rules Of Thumb

The concept of "levels of abstraction" can be scary and you absolutely should **NOT spend hours** on your code, just to analzye which levels you migth have there.

Instead, there are **two easy rules of thumb** I came up with, which help you 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 user object. `save()` then confirms these changes.

You 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, you implictly removed another problem: Mixed levels 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 to read but it definitely needs more interpretation from your side than just reading `processPayment(...)`.

Hence, you 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.

Split Functions Reasonably

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

This is dangerous - because this actually also can lead to bad code.

Consider this example:

```
function createUser(email, password) {
  validateInput(email, password);

  saveUser(email, password);
}

function validateInput(email, password) {
  if (!isEmail(email) || isInvalidPassword(password)) {
    throwError('Invalid input');
  }
}

function isEmail(email) { ... }

function isInvalidPassword(password) { ... }

function throwError(message) {
  throw new Error(message);
}

function saveUser(email, password) {
  const user = buildUser(email, password);
  user.save();
}

function buildUser(email, password) {
  return new User(email, password);
}
```

And now compare it to this version:

```
function createUser(email, password) {
  validateInput(email, password);

  saveUser(email, password);
}
```

```
function validateInput(email, password) {
  if (!isEmail(email) || isInvalidPassword(password)) {
    throw new Error('Invalid input');
  }
}

function isEmail(email) { ... }

function isInvalidPassword(password) { ...

function saveUser(email, password) {
  const user = new User(email, password);
  user.save();
}
```

Which version is easier to understand?

I would argue, the second version is. Even though (or actually: because) we have **less function extractions** there.

Splitting functions and keeping them short is important! But pointless extractions lead nowhere - you shouldn't extract just for the extraction's sake.

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

There are **three main signals:**

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

In the example above, both the `throwError()` and the `buildUser()` functions in the end just renamed the operation they contained. For `buildUser()`, coming up with a good name was hard because `createUser()` was already taken - and does more than just create a single user object.

Avoid Unexpected Side Effects

In addition to everything covered above, there's one last important aspect which you should keep in mind to write clean functions.

You should **avoid unexpected side effects in functions**.

What's a Side Effect?

A side effect is simply an operation which **changes the state** (data, system status etc.) of the application.

Connecting to a database is a side effect. Sending an Http request is one. Printing output to the console or changing data saved in memory - all these things are side effects.

Side effects are a **normal thing in development** - after all, we DO build applications in order to derive results and change things.

Problems arise when a side effect is unexpected.

Unexpected Side Effects

A side effect is unexpected, when the name and or context of a function doesn't imply it.

Consider this example:

```
function validateUserInput(email, password) {
  if (!isEmail(email) || passwordIsInvalid(password)) {
    throw new Error('Invalid input!');
  }

  createSession();
}
```

This function has an **unexpected side-effect**: `createSession()`.

Creating a session (which has an impact on data in memory, maybe even on data in a database or files) is definitely a side-effect.

We might expect this kind of side-effect in a function named `login()`, but in `validateUserInput()`, it's definite **not expected**. And that's a problem.

Hence you should **move this side-effect into another function**, or - if it makes sense - **rename the function** to imply that this side-effect will be caused.

Control Structures

No matter which kind of application you're building - you will most likely also use control structures in your 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 you 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, you 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 you already know from 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 your `if` checks instead of negative wording.

Though - in my opinion at least - sometimes a short negative phrase is better than a constructed positive one.

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, I do prefer the negative version:

```
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 to combine.

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

Avoid Deep Nesting

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

You can see an example for deeply nested control structures in the section videos - it's quite long, which is why I didn't paste it into this document.

There are a couple of techniques that can help you 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, you can extract a nested `if` check and **move it right to the 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 without a guard:

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

Here's the improved version, using 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 funtion **not** continuing if one condition fails.

Guards are these `if` checks right at the start of your functions.

You simply take your nested checks, **invert the logic** (i.e. check for the user **not** accepting messages etc.) and you 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.

Consider this example:

```
function connectDatabase(uri) {
  if (!uri) {
    throw new Error('An URI is required!');
  }

  const db = new Database(uri);
  let success = db.connect();
  if (!success) {
    if (db.fallbackConnection) {
      return db.fallbackConnectionDetails;
    } else {
      throw new Error('Could not connect!');
    }
  }
  return db.connectionDetails;
}
```

This code could be improved by applying what we learned about functions:

```
function connectDatabase(uri) {
  validateUri(uri);

  const db = new Database(uri);
  let success = db.connect();
  let connectionDetails;
  if (success) {
    connectionDetails = db.connectionDetails;
  else {
    connectionDetails = connectFallbackDatabase(db);
  }
  return connectionDetails;
}

function validateUri(uri) {
  if (!uri) {
    throw new Error('An URI is required!');
  }
}

function connectFallbackDatabase(db) {
  if (db.fallbackConnection) {
    return db.fallbackConnectionDetails;
  } else {
    throw new Error('Could not connect!');
  }
}
```

You might be able to optimize this code even more, but you can already see that the nested control structure was removed by extracting a separate `connectFallbackDatabase()` function.

Polymorphism & Factory Functions

Sometimes, you 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 you.

Before we dive into these concepts, 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 of twice, 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 were **NOT executed yet** (note, that the **`()` are missing** after `processCreditCardPayment` etc.).

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: The same problem can also be solved by using classes and inheritance - this will be **covered in the "Classes" course section**!*

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 to generate an error.

Once an error is "on its way", it'll bubble up through the entire call stack and **cancel any function 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 out of the `createUser()` function.

```
function handleSignupRequest(request) {
  try {
    createUser(request.email, request.password)
  } catch (error) {
    console.log(error.message);
  }
}

function createUser(email, password) {
  validateInput(email, password);
  // ... 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!');
  }
}
```

Indeed, **error handling should typically be considered to 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

This course does not focus on object-oriented programming, yet objects - and also classes - are of course an important aspect of programming in general, even if you don't follow a purely object-orient style.

Some concepts discussed in this document (and the respective course section) DO come from the object-oriented world and are focused on object-oriented programming.

Nonetheless, I included all concepts that help with writing classes and working with objects in general (even when using a mixed programming style) and/ or which also are interesting when not focusing on OOP.

When it comes to working with classes and objects and writing clean classes and objects, there are a **couple of rules and concepts you 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 you can create objects without (actively) using classes - for example JavaScript. And there are languages where you MUST use classes for pretty much everything (e.g. Java).

Either way, objects allow you to **group related data** (properties) and **functionalities** (methods) **together**. And objects typically expose a public API of methods which can be used anywhere in your 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**.

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 `public`ly.

An **object on the other hand hides it's data from the public and instead exposes a public API** 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 you should **avoid mixing the types.**

Why does this matter?

When writing clean code, 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 absolutely 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 you re-use code (e.g. call the same method) **multiple times** but that the code will do **different things**, depending on the object type.

It's a powerful concept which can be utilized to avoid code duplication.

Here's an example that does not use polymorphism:

```
class Delivery {
  private purchase: Purchase;

  constructor(purchase: Purchase) {
    this.purchase = purchase;
  }

  deliverProduct() {
    if (this.purchase.deliveryType === 'express') {
      Logistics.issueExpressDelivery(this.purchase.product);
    } else if (this.purchase.deliveryType === 'insured') {
      Logistics.issueInsuredDelivery(this.purchase.product);
    } else {
      Logistics.issueStandardDelivery(this.purchase.product);
    }
  }

  trackProduct() {
    if (this.purchase.deliveryType === 'express') {
      Logistics.trackExpressDelivery(this.purchase.product);
    } else if (this.purchase.deliveryType === 'insured') {
      Logistics.trackInsuredDelivery(this.purchase.product);
    } else {
      Logistics.trackStandardDelivery(this.purchase.product);
    }
    }
  }
```

The problem with this code is that we have the same `if` checks with different logic inside of the `if` statements. So it's some code duplication but not everything is duplicated.

And we need these different cases because a product can of course be sent and tracked (and maybe we could even come up with additional methods).

Here's how this issue could be solved in a polymorphic way:

```
class Delivery {
  protected purchase: Purchase;

  constructor(purchase: Purchase) {
    this.purchase = purchase;
  }
}

class ExpressDelivery extends Delivery {
  deliverProduct() {
    Logistics.issueExpressDelivery(this.purchase.product);
  }

  trackProduct() {
    Logistics.trackExpressDelivery(this.purchase.product);
  }
}

class InsuredDelivery extends Delivery {
  deliverProduct() {
    Logistics.issueInsuredDelivery(this.purchase.product);
  }

  trackProduct() {
    Logistics.trackInsuredDelivery(this.purchase.product);
  }
}

class StandardDelivery extends Delivery {
  deliverProduct() {
    Logistics.issueStandardDelivery(this.purchase.product);
  }

  trackProduct() {
    Logistics.trackStandardDelivery(this.purchase.product);
  }
}
```

Above, there are three specialized classes (`ExpressDelivery` etc.) which all inherit from the `Delivery` base class to share common functionalities.

All these specialized classes implement the same `deliverProduct()` and `trackProduct()` methods - though the exact same code in these methods differs.

That's the idea behind polymorphism!

We can now write a factory function or class method that creates instances of these classes when needed. It's the factory function which holds the `if` check from above.

Hence we only need the check once instead of once per method.

```
function createDelivery(purchase) {
  if (purchase.deliveryType === 'express') {
    return new ExpressDelivery(purchase);
  } else if (purchase.deliveryType.deliveryType === 'insured') {
    return new InsuredDelivery(purchase);
  } else {
    return new StandardDelivery(purchase);
  }
}
```

You can now always run code like this and the exact result will depend on the `deliveryType` used for creating the objects:

```
const expressDelivery = createDelivery({deliveryType: 'express', ...});
expressDelivery.deliverProduct();
```

Classes Should Be Small

Just like functions, classses 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 **single object all tasks should be related to**.

A `User` class might for example 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** (*also see "Single Responsibility Principle", further down below*).

The responsibility of a `User` class should be to handle user-related tasks, for example `login()`, `logout()` etc.

Classes & Cohesion

Clean classes have a high cohesion.

But what is "cohesion"?

Cohesion basically describes the amount by which methods of a class use class properties.

If a class has 2 properties and 3 methods and every method uses both properties, this class would 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 maybe just be a data container / data structure instead of an object with a public API (since that public API, the methods, doesn't interact with the internal properties).

In reality, you will rarely achieve maximum cohesion but **high cohesion should definitely be your goal**.

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

Hence, if you care about cohesion (and you should!), you will automatically end up with smaller classes - which is good.

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 you 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**

In this case, `lastPurchase` is a property (an attribute) of the `customer` object, which in turn is a property of the class the executing method belongs to. It's a different object, maybe based on some `Purchase` class. And that class then has a `date` property.

Side-note: Here I directly access properties - it's the same if you would be using getters (`customer.getLastPurchase().getDate()`).

Accessing this `date` property through `lastPurchase` violates the Law of Demeter.

The problem with code like this is that you can end up with very long statements, full of "chaining" (`.something.somethingElse.more()` is called "chaining").

This creates strong dependencies and whenever the `Purchase` class would change (e.g. `date` is renamed to `paymentDate`) all code snippets that look like above need to be adjusted.

The above code would better be implemented like this:

```
this.customer.getLastPurchaseDate();
```

Now the extra dependency on `lastPurchase.date` was removed.

Even this solution is not optimal though - it would be better to "**tell, not ask**". That means, it depends on what you actually needed to do with that date.

Let's say you needed the date to send the purchased products to the customer.

```
const date = this.customer.lastPurchase.date;
this.warehouse.deliverPurchasesByDate(this.customer, date);
```

Instead of getting the date and then checking for the products, the following code would've been a better way of avoiding the "Law of Demeter" violation:

```
this.warehouse.deliverPurchase(customer.lastPurchase);
```

As you can see, it's not just about moving some code around - sometimes a different solution (working with different objects) is the cleanest way of getting something done.

It's also important to note that the "Law of Demeter" really only applies to **property / attribute chaining**.

The following code would be fine (as long as these **methods are real methods and not just getters** wrapped around properties):

```
this.customer.sendMessage(message).retry(2);
```

Why would this be okay?

Because now we **utilize the public interfaces** of various classes instead of "abusing" their internal data structure.

If you are only working with a couple of **data containers**, you 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, since it **enforces smaller, and therefore more readable, classes.**

Here's an example for a class that violates the SRP:

```
class ReportDocument {
  generateReport(data: any) { ... }

  createPDF(report: any) { ... }
}
```

The actual code in the methods is missing, because it's not needed - it's the structure and API of the class which matters here.

This `ReportDocument` class violates the SRP because it needs to change because of at least two reasons:

- If anything related to how a report is generated changes
- If anything related to how a report is printed as a PDF changes

It should be fair to assume that in most applications, these two features are probably not directly connected. The people deciding on how the PDF should look like are not necessarily the same people who decide how the report should be generated.

This class would therefore better be split - e.g. into `Report` and `PDFPrinter` classes.

The SRP 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 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.*

Whilst the above sentence might sound cryptic, this is actually a straightforward principle.

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

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

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

Here's an example of a class that violates the OCP:

```
class Printer {
  printPDF(data: any) {
    // ...
  }

  printWebDocument(data: any) {
    // ...
  }

  printPage(data: any) {
    // ...
  }

  verifyData(data: any) {
    // ...
  }
}
```

This `Printer` class has different methods for printing a web document (e.g. generating a HTML file), a "normal" page and a PDF.

Whenever we add a new type of document (e.g. a Word or Excel document), we need to add a new method - probably with a decent junk of code duplication.

This problem might sound familiar - we solved it with "Polymorphism" earlier!

And indeed, that's how we can fix the issue here as well (and follow the OCP).

This `Printer` class is currently **not closed** because we need to come back to it and edit it whenever a new type of document should be printed.

To follow the OCP, we should close it and instead extend it whenever we want to add a new type of document.

```
interface Printer {
  print(data: any);
}

class PrinterImplementation {
  verifyData(data: any) {}
}

class WebPrinter extends PrinterImplementation implements Printer {
  print(data: any) {
    // print web document
  }
}

class PDFPrinter extends PrinterImplementation implements Printer {
  print(data: any) {
    // print PDF document
  }
}

class PagePrinter extends PrinterImplementation implements Printer {
  print(data: any) {
    // print real page
  }
}
```

This snippet shows how we could fix the issue. Now, the base class `PrinterImplementation` (named like this because we have an interface named `Printer`) is closed. It doesn't need to change just because we want to support a new type of document.

Instead, we extend it - we add new subclasses which are based on `PrinterImplementation` (which adds common functionality like verification) and implement `Printer` (which forces all implementing classes to have a `print()` method).

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 want for our classes from a clean code perspective!

L: Liskov Substitution Principle (LSP)

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

The above sentence should be quite clear. Here's an example of a baseclass being replaced with a subclass of it:

```
class Bird {
  fly() {
    console.log('Fyling...');
  }
}

class Eagle extends Bird {
  dive() {
    console.log('Diving...');
  }
}

const bird = new Bird();
bird.fly();

// We can also replace Bird() with Eagle()
const eagle = new Eagle();
eagle.fly();
```

In this snippet, the `Bird` class is the superclass to the `Eagle` class and indeed, we can substitute the `Bird` which we construct with an `Eagle` without changing our code.

Now, let's add a new subclass which actually violates the principle:

```
class Penguin extends Bird {
  // Problem: Can't fly!
}
```

We can't use `Penguin` as drop-in replacement for `Bird`, because penguins can't fly!

So actually, as soon as we have a case like this, we **violated the LSP**.

A better way of modelling our data then would be to do it like this:

```
class Bird {}

class FlyingBird extends Bird {
  fly() {
    console.log('Fyling...');
  }
}

class Eagle extends FlyingBird {
  dive() {
    console.log('Diving...');
  }
}

const eagle = new Eagle();
eagle.fly();
eagle.dive();

class Penguin extends Bird {
  // Good! Birds don't need to fly!
}
```

Now we added a new `FlyingBird` class which is based on `Bird`. And we can now choose whether we have other birds which are "just birds" (like `Penguin`) or "flying birds" (like `Eagle`).

Now we follow the LSP.

Hence, it's fair to say that the LSP in the end has the goal of **forcing us to model our data correctly** and think carefully about our models and the entities we work with in our code.

The LSP is definitely an important and popular principle, but it doesn't have that big of an effect on our code if we look at it from a clean code perspective.

Yes, it could lead to smaller classes but it's really mostly focused on forcing us to model our data correctly.

I: Interface Seggragation Principle (ISP)

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

Reading sentences like the one above can always be confusing, but the ISP is actually also quite straightforward.

When working with classes and in an OOP way, you often also work with "Interfaces". Not all programming languages support interfaces but many do.

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

Here's an example of code that violates the ISP:

```
interface Database {
  storeData(data: any);
  connect(uri: string);
}

class SQLDatabase implements Database {
  connect(uri: string) {
    // connecting...
  }

  storeData(data: any) {
    // Storing data...
  }
}

class InMemoryDatabase implements Database {
  connect(uri: string) {
    // Needs a connect method (because of interface)
    // but hasn't anything to connect to :(
  }

  storeData(data: any) {
    // Storing data...
  }
}
```

In this example, we have a `Database` interface and then we got a couple of database classes for different kinds of databases.

The problem with that code is, that the `Database` interface is too generic - "too general purpose".

It forces the `InMemoryDatabase` to add a `connect()` method, even though this type of database has nothing to connect to (i.e. there is no extra database server or anything like that).

Hence it would be better to work with multiple, more specialized interfaces instead of the general purpose one. And that's exactly what the ISP says!

```
interface Database {
  storeData(data: any);
}

interface RemoteDatabase {
  connect(uri: string);
}

class SQLDatabase implements Database, RemoteDatabase {
  connect(uri: string) {
    // connecting...
  }

  storeData(data: any) {
    // Storing data...
  }
}

class InMemoryDatabase implements Database {
  storeData(data: any) {
    // Storing data...
  }
}
```

Now, with this code, we got **two interfaces** intead of one and every class can pick the interfaces that make sense for it. Hence our `InMemoryDatabase` is able to just implement `Database`, which forces it to have a `storeData()` method and ignore the `RemoteDatabase` interface, which would force it to add a `connect()` method.

The ISP is definitely an important and popular principle, but it doesn't have that big of an effect on our code if we look at it from a clean code perspective.

D: Dependecy Inversion Principle (DIP)

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

This last principle is a principle which you will often already follow, if you follow the other SOLID principles.

It basically is all about not being too specific in your code.

Sounds strange?

Here's an example of code which we could improve (this code assumes that we have database classes as shown for the ISP available):

```
class App {
  private database: Database | RemoteDatabase;
  constructor(database: Database | RemoteDatabase) {
    if (database.connect) {
      database.connect('my-url');
    }
    this.database = database;
  }

  saveSettings() {
    this.database.storeData('Some data');
  }
}


const sqlDatabase = new SQLDatabase();
const app = new App(sqlDatabase);
```

In this snippet, we use the databases we created for the ISP. In the `constructor()` of our `App` class, we check whether we're getting a `RemoteDatabase` (i.e. if we have a `connect()` method) and if we do, we do connect to the database.

The problem with this code is, that we're very specific in our `constructor()` method. The code we execute depends on which kind of database we're getting.

Hence we "depend on a concretion" in that example.

This is suboptimal because it means that we need to make this check whereever we get such a database and we also need to change our code whenever the database implementation changes.

Here's how we could restructure the code to follow the ISP:

```
class App {
  private database: Database;

  constructor(database: Database) {
    this.database = database;
  }

  saveSettings() {
    this.database.storeData('Some data');
  }
}

const sqlDatabase = new SQLDatabase();
sqlDatabase.connect('my-url');
const app = new App(sqlDatabase);
```

This snippet shows us why it's called the "Dependency Inversion Principle".

We ensure that we **depend on an abstraction** ("we just get some database") **instead of a concretion** ("we need to check whether we need to connect"). And we do that by **inverting the dependency**.

Instead of depending on the kind of database inside of `constructor()`, we "force" whoever is creating an `App` to give us a database which is already connected (if required).

Use Common Sense

Just as in the "Functions" section of this course, I really want to emphase one important thing: Use your common sense.

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

This is NOT the goal and definitely NOT clean code!

Yes, classes should be small - just like methods and functions.

But "small" does not mean "almost empty".

It is okay to do work in classes and methods. And whilst methods indeed probably shouldn't be dozens of lines long, the idea behind classes is to group related data and concepts together.

Don't destroy that by breaking up everything!

Instead, consider the various concepts and rules explained here. Apply them as good as you can but don't start breaking up everything because of them.

Clean Code - Checklist

Naming

- ◯ Use **descriptive** and meaningful names
    - ◯ **Variables & Properties**: Nouns or short phrases with adjectives
    - ◯ **Functions and Methods**: Verbs or short phrases with adjectives
    - ◯ **Classes**: Nouns
- ◯ Be as **specific** as necessary and possible
- ◯ Use **yes/ no** "questions" for booleans (e.g. `isValid`)
- ◯ **Avoid misleading** names
- ◯ Be **consistent** with your names (e.g. stick to `get...` instead of `fetch...`)

Comments & Formatting

- ◯ **Most comments are bad** - avoid them!
- ◯ Some good comments are **acceptable**
    - ◯ **Legal** comments
    - ◯ **Warnings**
    - ◯ **Helpful explanations** (e.g. for Regex)
    - ◯ **Todos** (don't overdo it though)
- ◯ Use vertical formatting:
    - ◯ Keep related concepts close to each other (**vertical density**)
    - ◯ Add spacing / distance (e.g. blank linkes) between concepts that are not directly related (**vertical distance**)
    - ◯ Write code **top to bottom**: Called functions should come below calling functions (if possible)
- ◯ Use **horizontal** formatting:
    - ◯ **Avoid long lines** - break them into multiple lines instead
    - ◯ Use **indentation** to express scope

Functions

- ◯ **Limit the number of parameters** your functions use - less is better!
- ◯ Consider using objects, dictionaries or arrays to **group multiple parameters into one parameter**
- ◯ Functions should be **small and do one thing**
  - ◯ Levels of abstraction inside the function body should be **one level below the level implied by the function name**
  - ◯ **Avoid mixing levels** of abstractions in functions
  - ◯ But: **Avoid redundent splitting!**
- ◯ Stay **DRY** (Don't Repeat Yourself)
- ◯ **Avoid unexpected side effects**

Control Structures & Errors

- ◯ Prefer **positive checks**
- ◯ Avoid **deep nesting**
  - ◯ Consider using "**Guard**" statements
  - ◯ Consider using **polymorphism** and **factory functions**
  - ◯ **Extract control structures** into separate functions
- ◯ Consider using **"real" errors** (with error handling) instead of "synthetic errors" built with `if` statements

Objects & Classes

- ◯ Focus on building "real objects" **or** data containers / structures
- ◯ Build **small classes** - focus on a **single responsibility** (which does **not** mean "single method"!)
- ◯ Build classes with **high cohesion**
- ◯ Follow the "**Law of Demeter**" for "real objects" (avoid `this.customer.lastPurchase.date`)
- ◯ Especially when doing OOP: Follow the SOLID principles
- ◯ Especially **SRP and OCP** will help a lot with writing clean code (= readable code)