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 distanct 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!