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.