# 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.