# More Advanced JavaScript Concepts

There are quite a few additional JavaScript concepts, which you'll encounter throughout this course and most web projects you'll work one.

This course section (and this document) should act as a "mini-reference" that introduces and explains these concepts, so that they are clear later in the course (and so that you can always come back to this section or document if you want to revisit the concepts).

## Functions & Default Parameter Values (Optional Parameters)

Functions can take parameters - you already know that (hopefully ;-)).

What's new is that they can also take default values for some or all parameters. Those parameters then effectively become optional since they will always assume that default value:

```
function multiply(number, factor = 2) { // factor is optional because it has a default value
  return number * factor;
}
```

If you **don't** assign default values, parameters that are not set when the function is called will actually have a "default default parameter value" of `undefined` - that's a special built-in value in JavaScript which signals that no value exists for this parameter / variable / constant. Typically, you don't want to work with `undefined`, hence you either expect code to pass values for parameters or you set your own default values as shown above.

Parameters with default values always have to come **after** parameters without default values so that the function can be called correctly. Because how would you call the below function if your intention was to **not** set `factor` and use its default?

```
function multiply(factor = 2, number) { // this is wrong!
  return number * factor;
}

multiply(3); // 3 would now be used as a value for 'factor', 'number' would be undefined
```

## Rest Parameters

Sometimes, you need to write a function that does not have a fixed amount of parameter values. For example, a function that sums up all the values (numbers) it receives. Or a function that stores a list of user names in a database.

You could then always accept an array as a value but you can also accept a list of individual parameter values instead by using the "Rest parameters" feature:

```
function sumUp(...numbers) {
  // ... do something ...
}

sumUp(5, -10, 30, 21, 8); // NOT an array of numbers but a list of individual numbers
```

The `...` annotation turns `numbers` into an automatically-created array that groups all individual parameter values together.

So inside of the function, you work with an array, outside of the function, when you call it, you pass in a comma-separated list of values (**not** an array).

Why would you do that over just using a single array in the first place? Well, it's simply some extra flexibility you get for your functions. For example, there also is the built-in `Math.min()` function that also takes a list of values, instead of an array (e.g. `Math.min(4, 2, 1)`).

## The Spread Operator

`...` has another meaning, besides constructing rest paramters. You can also use it as an operator outside of function definitions (i.e. outside of the place where you define the parameters a function should receive).

Then, `...` acts as the so-called "Spread operator" and it does what its name says: It "spreads" the values of an array or object into a list of values (or key-value pairs, in case of an object).

You can use the `...` operator to convert an array into a comma-separated list of values, in cases where such a list might be required:

```
const numbers = [1, 5, -3]; // an array
console.log(Math.min(...numbers)); // using ... because Math.min wants a list, NOT an
array
```

You can also use the spread operator to create a copy of an array or object:

```
const person = { name: 'Max', age: 32 };
const numbers = [1, 2, 3];

const copiedPerson = { ...person };
const copiedNumbers = [ ...numbers ];
```

## Functions Are Objects

It is worth pointing out, that functions are in the end just objects, under the hood.

Special kinds of objects, since you can execute them but still objects - with properties and methods, though you rarely use that feature (i.e. you rarely add methods or properties to your functions).

But you will sometimes encounter this fact when working with built-in functions or using third-party functions.

For example, when we used the ExpressJS package, we could use the imported "express thing" in various ways:

```
const express = require('express');

const app = express(); // executing "express" as a function

app.use(express.static('public')); // executing a method on the express function object
```

This works because functions are objects and the developers of the Express package simply decided to make express a function **and** add more features (e.g. the `static` method) to that function object to enable you to use those features in your code.

## Template Literals

When constructing strings, especially longer strings which might be a mixture of hard-coded text and dynamic values, JavaScript has a feature that helps you shorten your code: Template literals.

```
const firstName = 'Max';
const lastName = 'Schwarzmüller';

const greeting1 = 'Hi, I am ' + firstName + ' ' + lastName + '!'; // long way
const greeting2 = `Hi, I am ${firstName} ${lastName}!`; // shorter way
```

Template literals produce regular strings but you create them by using the special "backtick" character on your keyboard (`) instead of the single or double quote character.

Inside of template literals, you can "inject" results of JS expressions (e.g. values stored in variables or constants) via the `${}` syntax.

## Primitive vs Reference Values

Theory time! In JavaScript, you have two kinds of values you're working with: Primitive values and reference values.

And the rules are simple: **All** objects are reference values, all other kinds of values (e.g. numbers, strings, booleans but also `undefined` or `null`) are primitive values.

What's the difference?

Primitive values are, as the name implies, rather simple values. They are stored in a simpler kind of computer memory and in your code, you can think of those values being stored directly in your variables.

I.e. `const number = 32` means that the value `32` is directly stored in `number`.

The implication of that is that if you assign that to a new variable or constant, the value is **copied**.

```
const number = 32;
let newNumber = number;
newNumber = 10;
console.log(number); // still outputs 32
console.log(newNumber); // outputs 10
```

Reference values are a bit more complex. Objects (which are the reference values) can become complex and quite "big" in memory - that's why they're stored in a "special kind of computer memory" and in your code, you can think of only the address of the value in memory being stored in your variables and constants, **not** the value itself.

Hence if you re-assign reference values, you only copy the address, not the value - and this can lead to unexpected behaviors:

```
const person = { name: 'Max', age: 32 };
const newPerson = person;
newPerson.age = 35;
console.log(newPerson.age); // outputs 35
console.log(person.age); // also outputs 35
```

In this example, `person.age` is `35` because when setting `newPerson = person`, we **don't** copy the person value (i.e. we **don't** create a new object in memory) but instead only the address is copied. Therefore, when you change `newPerson`, JavaScript uses the address to look up the value in memory and that value gets changed. Therefore `person` got changed as well (because it's one and the same object value in memory!).

That also explains why you can manipulate objects and arrays (arrays are also just objects!), even if they're stored as a `const`.

You can do that because you don't change the address, which is the actual value stored in the constant, but only the value in memory.

## Error Handling with try / catch

Sometimes, things go wrong.

In an Express app, you can use the built-in default error handling middleware to then handle errors and send back error responses (e.g. a `500` error page).

But sometimes you need more fine-grained control over errors - maybe because you're **not** in an Express app or because you might want to retry the operation or execute some fallback code (instead of sending back an error response).

For all these scenarios, you have `try` / `catch` in JavaScript:

```
try {
  someCodeThatCouldFail();
} catch (error) {
  console.log('An error occurred!');
  console.log(error); // logging the generated error object
}
```

This allows you to wrap some code that could fail with `try` and put your fallback code into `catch`.

Whenever an error is generated, an value (typically an object) describing that error is generated automatically - you can get access to this object after `catch` by kind of accepting it as a parameter `catch (error)`.

You can also throw your own errors:

```
function doSomething() {
    // do something ...
    throw { message: 'Something went wrong! };
}
```

That's a bit more advanced but that is in the end what all these built-in functions and methods do, if they cause an error.

**Very important**: You should **not** wrap all your code with `try`! This might sound like a good idea first, but you should really only wrap code that could fail, where you as a developer can't prevent possible failures (e.g. erroneous user input, file access that might fail because the file is temorarily not accessible...).

Other kinds of errors ("bugs") that your code might contain should be detected by you during development!

## Variable Scope & Variable Shadowing

Variables, constants and functions are "scoped" in JavaScript - which simply means: You can't use them everywhere.

Generally, there is a simple rule: You can use the variables and constants in the block, where you defined them and in "nested blocks". A block is created with curly braces (except for when you create an object, that's not a block). `if`, `for`, `try` and functions all created their own blocks which limit the availability of variables and constants.

```
const firstName = 'Max'; // defined globally - hence available everyhwere

function greet(mode) { // mode is scoped to this function
  let phrase; // scoped to this function
  if (mode === 'friendly') { // creates a nested block and hence scope
    phrase = 'Amazing that you are here '; // "phrase" can be used in the nested block
  } else {
    phrase = 'Good that you are here ';
  }
  console.log(phrase + firstName); // "firstName" can be used here
}

console.log(phrase); // this would fail, "phrase" is defined in a different scope
```

## Object Blueprints with Classes

Objects are a key feature of JavaScript (and actually a lot of other programming languages as well!).

You can create objects like this:

```
const person = { name: 'Max', age: 32 }; // use curly braces like this, to create an
object
```

That's great if you create an object to group some data together.

Sometimes though, you also have scenarios where you need multiple objects that all have the same shape but different data.

Like this:

```
const developer = { title: 'Developer', location: 'New York' };
const cook = { title: 'Cook', location: 'Munich' };
const instructor = { title: 'Teacher', location: 'Online' };
```

You can still create objects like this, but for scenarios like that, using some kind of blueprint could be helpful. A blueprint that "locks in" a certain schema and allows you to then set different values for different instances of that blueprint.

And JavaScript has such a feature: Classes.

```
class Job {
  constructor(title, place) { // constructor is a special, reserved method
    this.title = title; // add a title property
    this.location = place; // add a location property
  }
}
```

With `class`, you create a blueprint. The name should start with an uppercase character and describe the kinds of objects that will be created based on that blueprint.

You then have a special `constructor` method that you add to that `class` - this will be called automatically, whenever the class is "instantiated" (= objects are created based on that class).

In the `constructor`, you can initialize the to-be-created object instance and add properties to it - with help of the `this` keyword which refers to the created object instance. You typically accept parameters for that (so that different values can be provided, for different objects).

```
const developer = new Job('Developer', 'New York');
const cook = new Job('Cook', 'Munich');
const instructor = new Job('Teacher', 'Online');
```

Objects are instantiated by calling the class like a function (`Job()`) and adding the special `new` keyword in front of it. That tells JS that a new object based on that blueprint should be created.

There also are built-in classes like `Date` that come with certain built-in functionalities.

Because that's also important: You can not just use classes to create blueprints for objects with properties, but you can also add methods to those objects. All objects created based on a class will then have these methods.

```
class Job {
  constructor(title, place) { // constructor is a special, reserved method
    this.title = title; // add a title property
    this.location = place; // add a location property
  }

  greet() {
    console.log('Hi there, I am a ' + this.title);
  }
}
```

Methods are added as shown in the above example. You can use the `this` keyword in methods to get hold of the concrete object instances on which this method then will be executed (i.e. to get access to the properties of the object).

## Destructuring Objects & Arrays

When working with objects and arrays, you sometimes want to read certain values stored in these objects and arrays.

You can of course do that with the dot notation for objects and with help of the index for arrays.

But sometimes, you want to store certain values in separate constants or variables - and you can easily do that with help of "destructuring".

```
const person = { name: 'Max', age: 32 };
const inputData = ['Max', 'Schwarzmüller'];

const { age: userAge } = person; // extract "age" property value and name constant
"userAge"
const [firstName, lastName] = inputData; // extract both elements into separate
constants
```

For object destructuring, you refer to the properties (by name) which you want to extract and you can assign a custom name (alias) as shown above (i.e. `const { age } = person` would **not** add an alias, `const { age: userAge} = person` does).

For array destructuring, the position matters (since array values are accessed by index, which describes the position) and you can simply extract values by position and store them in constants or variables with any names of your choice.

## Introducing Asynchronous Code

When writing JavaScript programs, you sometimes have operations that may take a bit longer - e.g. storing data in a file, reading a file, sending data to a database etc.

Such operations could potentially block your entire program if you always need to wait for them to finish, in order to move on to the next line of code.

That's why JavaScript knows the concept of "asynchronous code". With JS, you're able to "hand off" those tasks to some system process and continue with the next line without waiting for the asynchronous task to finish. Instead, you just define some function (a so-called "callback function") that should be exeucted once the task is done.

```
fs.readFile('some-file.txt', function(error, data) { // error is undefined if it worked
  console.log('Done reading the file!'); // executed only once the file was read
})
console.log('This is NOT blocked by reading the file!');
```

In the above example, we use the asynchronous version of the `readFile` method (i.e. **not** `readFileSync`, which is the synchronous, blocking alternative). Therefore, the `console.log` after `readFile` executes **before** the code in the callback function executes. This allows you to write performant programs which are not blocked unnecesssarily.

Of course this also means that all operations that depend on the file data have to go into the callback function because that's the only place where you know that reading the file will have finished and the file data is available.

## Introducing Promises

Asynchronous code is an important concept and it's fairly trivial in the end. But when working with callback functions as shown above, you can end up with situations where you have a lot of nested asynchronous code with nested callback functions - something that's often called "callback hell":

```
fs.readFile('some-file.txt', function(error, data) {
  // do something ...
  storeDataInDatabase(data, function() { // another async task is started
    // do something once storing in the database finished ...
    getConfirmationData(function() { // another dependent async process is started
      // do more work ...
    });
  });
})
```

Code like this can easily become unreadable and harder to maintain.

That's why we have a feature called "Promises" - a concept built-into JavaScript that allows you to deal with asynchronous code in a more readable manner.

```
// special import, promises need to be made available by the package you're using
// not all packages do that - the package documentation will tell you
const fs = require('fs/promises');

fs.readFile('some-file.txt')
  .then(function(data) { // then() registers a callback that is executed in success
case
    return storeDataInDatabase(data); // returning a function that yields a promise
  })
  .then(function() { // you can "chain" then() blocks after each other
    // this function will execute once "storeDataInDatabase" is done
    return getConfirmationData();
  })
  .then(function() {
    // do more work ...
  });
```

This is far more readable and structured!

You can build "promise chains" of `then` blocks to execute step after step. If the execution of the next `then` block should depend on some other asynchronous task in the previous `then` block, you simply return that task (assuming it also supports promises) and the next `then` block will only execute once the previous task finished.

Might sound complex, but it actually works pretty straightforward - you'll see it again later in the course!

## Working with async / await

Promises are a great feature that can make your code more readable.

But in cases where you only have operations that should execute after each other (i.e. operations that depend on each other), we might prefer a more "synchronous look" for our code.

In rare cases, like with `readFileSync`, a package offers a synchronous version of its functionalities for such cases. In most cases, you always have to work with a promise - even if you'd like to block code execution because the next lines of code depend on the result of the asynchronous task anyways.

For such scenarios (i.e. the vast majority), you have a nice feature built-into JavaScript: `async` / `await`.

```
async function doSomething() {
  const fileData = await fs.readFile('some-file.txt'); // due to await, execution is
blocked
  await storeInDatabase(fileData); // again blocking next line
  await getConfirmationData();
}
```

This is essentially the same program as in the previous snippet, but it's even more readable - because of `async` / `await`.

You use this feature by adding the `async` keyword in front of any function in which you want to use it. Then, inside of that function, you can use the `await` keyword.

Two important notes:

- Functions that are annotated with `async` **always** return a promise - even if you didn't add any `return` statement!
- `await` doesn't actually block code execution, instead it takes the code in the following lines and moves it into a `then` block automatically, behind the scenes

## Asynchronous Code Error Handling

Things can go wrong, also (and especially) when performing asynchronous operations like reading a file.

When relying on callback functions or promises without `async` / `await`, you **can't** use `try` / `catch`!

Instead, you have different strategies there:

```
fs.readFile('some-file.txt', function(error, data) { // error is undefined if it worked
  if (error) { // error is truthy - i.e. not undefined - if something went wrong
    console.log('Something went wrong!'); // error handling code goes here
    console.log(error.message);
  }
})
```

When relying on "pure callback functions" (i.e. without promises), the asynchronous operations will typically provide you an extra parameter that is `undefined` if everything worked but contains error data otherwise. Simply check if that parameter holds a value other than `undefined` and you know that something failed.

When working with promises, this works differently:

```
fs.readFile('some-file.txt')
  .then(function(data) { // then() registers a callback that is executed in success
case
```

```
    return storeDataInDatabase(data); // returning a function that yields a promise
  })
  .then(function() { // you can "chain" then() blocks after each other
    // this function will execute once "storeDataInDatabase" is done
    return getConfirmationData();
  })
  .then(function() {
    // do more work ...
  })
  .catch(function (error) {
    // this code executes if ANY of the previous steps failed
    console.log(error.message);
  });
```

When using promises, you can add a special `catch` method in your "promise chain" and it will execute the provided callback function whenever any error is thrown in any of the previous `then` callbacks.

When using `async` / `await`, you can again use `try` / `catch` and it's in the end converted to the `catch` block under the hood:

```
async function doSomething() {
  try {
    const fileData = await fs.readFile('some-file.txt');
    await storeInDatabase(fileData); // again blocking next line
    await getConfirmationData();
  } catch (error) {
    console.log('Something went wrong!');
    console.log(error.message);
  }
}
```