ES.Next

A small project to showcase the possibility of ES.Next and its setup process.

Setup a new Project

1. Create a new folder `mkdir edu_esnext` and move there `cd edu_esnext`
2. Initiate NPM `npm init -y`. Creates a `package.json` file.
3. Install necessary packages `npm install --save-dev webpack`
4. Install CLI tools for webpack `npm install webpack-cli --global`
5. Create a `webpack.config.js` file and add the following

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

module.exports = {
    mode: 'development',
    entry: { app: './src/index.js' },
    output: {
        filename: '[name].js',
        path: path.resolve(__dirname, 'public')
    }
};
```

7. Create `./src/index.js` file
8. Create `./public/index.html` and add the output script to it.

```
    <script src="app.js"></script>
```

9. Run `npx webpack` to trigger the webpack build
10. Start developing!

ESNext features

const / let / var

Unlike var, const and let are block scoped variables. The difference between those is:

```
for(var i=0;i<10;i++) {}
console.log(i) // output: 10

for(let n=0;n<10;n++) {}
console.log(n) // output: "Error: n is not defined"
```

Const variables can only be set once and are not reassignable

```
const PI = 3.141592653589793
PI = 42 // output: "SyntaxError: "PI" is read-only"
```

Template literal strings

Instead of construction a string like this:

```
"Helo " + name + ", nice to meet you!"
```

We can now use:

```
`Hello ${name}, nice to meet you!`
```

Arrow functions

Arrow function expressions provide a shorter syntax than noraml function expressions and does not have its own `this` scope.

Before function expressions were used like this:

```
var that = this
setTimeout(function() {
    that.doSomething()
}, 1000)
```

We can now use:

```
setTimeout(() => {
    this.doSomething()
}, 1000)
```

Binding `this` is not needed anymore because the arrow functions don't have their own `this` scope. Arrow functions are usually used for non-method functions. Those are functions that are not connected within a class.

Promises

A promise object represents the eventual outcome of an asynchronous operation. And it's used like this:

```
const getUserProfile = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('Felix Saaro')
```

```
    }, 1000)
})

getUserProfile.then(profile => {
    console.log(profile) // output: "Felix Saaro"
})
```

Async / Await

Async provides a reliable way of building asynchronous functions and it's used like this:

```
const getUserProfile = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('Felix Saaro')
    }, 1000)
})

const loadUserProfile = async () => {
    const profile = await getUserProfile()
    console.log(profile) // output: "Felix Saaro"
}
```

Instead of using `.then()` after we call the promise, we can now use the keyword `await`. This way we have less code and nesting.

Classes and inherticance

To create a class use the `class` key word.

```
class App {
    constructor() {
        console.log('init app class')
    }
}

// Directly initiate the app class if the file is loaded
new App()
```

To make the class available to other classes use the `export default`.

```
export default class Header {}
```

To make use of an exported class use import statement.

```
import Header from './header.js'
```

With ESNext we can also use inhertiance. This allows us to build object oriented application structures.

```
import Header from './header.js'

export default class MobileHeader extends Header {
    constructor() {
        super()
    }
}
```

The keyword `super` allows us to call functions on the parent class. Using it in the constructor allows us to inisiate the parents constructor.

Object Rest/Spread operator

these operators allow us to make easy changes to objects and arrays. If we want to combine two objects we can do it in two ways.

```
let object1 = {a: 1}
let object2 = {b: 2, c: {d: 3}}

let objectCombinded = Object.assign(object1, object2)

console.log(objectCombinded) //output: { a:1, b:2, c: { d: 3 } }
```

```
let object1 = {a: 1}
let object2 = {b: 2, c: {d: 3}}

let objectCombinded = {
    ...object1,
    ...object2
}

console.log(objectCombinded) //output: { a:1, b:2, c: { d: 3 } }
```