ES6

Prevent Object Mutation → JavaScript provides a function `Object.freeze` to prevent data mutation.

Use Arrow Functions to Write Concise Anonymous Functions

```
const myFunc = () => {
  const myVar = "value";
  return myVar;
}

//if there is no function body
const myFunc = () => "value";

//Arrow Functions with Parameters
const doubler = (item) => item * 2;

//Set Default Parameters for Your Functions
const greeting = (name = "Anonymous") => "Hello " + name;
```

Use the Rest Parameter with Function Parameters

```
/*The rest parameter eliminates the need to check the args array and allows us to apply ma
p(), filter() and reduce() on the parameters array.*/
function howMany(...args) {
  return "You have passed " + args.length + " arguments.";
}
```

Use the Spread Operator to Evaluate Arrays In-Place

```
/*We had to use Math.max.apply(null, arr) because Math.max(arr) returns NaN. Math.max() ex
pects comma-separated arguments, but not an array. The spread operator makes this syntax m
uch better to read and maintain.*/
var arr = [6, 89, 3, 45];
var maximus = Math.max.apply(null, arr);

//...arr returns an unpacked array. In other words, it spreads the array.
const arr = [6, 89, 3, 45];
const maximus = Math.max(...arr);
const arrCopy = [...arr]
```

Destructuring assignment is special syntax introduced in ES6, for neatly assigning values taken directly from an object.

Use Destructuring Assignment:

1. **to Extract Values from Object**

```
const user = { name: 'John Doe', age: 34 };
const { name, age } = user;
/*the name and age variables will be created
and assigned the values of their respective values from the user object.*/
```

2. **to Assign Variables from Objects**

```
const { name: userName, age: userAge } = user;
```

3. **to Assign Variables from Nested Objects**

```
const user = {
  johnDoe: {
    age: 34,
    email: 'johnDoe@freeCodeCamp.com'
  }
};
const { johnDoe: { age, email }} = user;
const { johnDoe: { age: userAge, email: userEmail }} = user;
```

4. **to Assign Variables from Arrays**

```
const [a, b] = [1, 2, 3, 4, 5, 6];
console.log(a, b); //The console will display the values of a and b as 1, 2.

const [a, b,,, c] = [1, 2, 3, 4, 5, 6];
console.log(a, b, c); //The console will display the values of a, b, and c as 1, 2, 5.
```

5. **with the Rest Parameter to Reassign Array Elements**

```
const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];
//The result is similar to Array.prototype.slice()
```

6. **to Pass an Object as a Function's Parameters**

```
const profileUpdate = ({ name, age, nationality, location }) => {
}
```

Create Strings using Template Literals → Template literals allow you to create multi-line strings and to use string interpolation features to create strings.

```
const person = {
  name: "Zodiac Hasbro",
  age: 56
};

const greeting = `Hello, my name is ${person.name}!
I am ${person.age} years old.`;

console.log(greeting);
```

UpperCamelCase should be used by convention for ES6 class names.

```
//In ES5, we usually define a constructor function and use the new keyword to instantiate
 an object.
var SpaceShuttle = function(targetPlanet){
  this.targetPlanet = targetPlanet;
}
var zeus = new SpaceShuttle('Jupiter');

//The class syntax simply replaces the constructor function creation
class SpaceShuttle {
  constructor(targetPlanet) {
    this.targetPlanet = targetPlanet;
  }
}
const zeus = new SpaceShuttle('Jupiter');
```

Use getters and setters to Control Access to an Object

```
class Book {
  constructor(author) {
    this._author = author;
  }
  // getter
  get writer() {
    return this._author;
  }
  // setter
  set writer(updatedAuthor) {
    this._author = updatedAuthor;
  }
}
const novel = new Book('anonymous');
console.log(novel.writer);
novel.writer = 'newAuthor';
console.log(novel.writer);
```

Create a Module Script → A script that uses this module type can now use the import and export features

```
<html>
<body>
<!-- Only change code below this line -->
<script type="module" src="filename.js"></script>
<!-- Only change code above this line -->
</body>
</html>
```

Use export to Share a Code Block

When you export a variable or function, you can import it in another file and use it without having to rewrite the code.

```
export const add = (x, y) => {
  return x + y;
}

//OR

const add = (x, y) => {
  return x + y;
}

export { add };
```

Reuse JavaScript Code Using import

```
import { add, subtract } from './math_functions.js';

//Use * to Import Everything from a File
import * as myMathModule from "./math_functions.js";
```

Create an Export Fallback with export default

```
export default function add(x, y) {
  return x + y;
}

export default function(x, y) {
  return x + y;
}
```

Import a Default Export

```
import add from './math_functions.js';
```

JavaScript Promise

It takes a function, as its argument, with two parameters - `resolve` and `reject`. These are methods used to determine the outcome of the promise.

```
const myPromise = new Promise((resolve, reject) => {
  if(condition here) {
    resolve("Promise was fulfilled");
  } else {
    reject("Promise was rejected");
  }
});
```

Handle a Fulfilled Promise with then

```
myPromise.then(result => {

});
```

Handle a Rejected Promise with catch

```
myPromise.catch(error => {

});
```

```
const makeServerRequest = new Promise((resolve, reject) => {
  // responseFromServer is set to false to represent an unsuccessful response from a server
  let responseFromServer = false;

  if(responseFromServer) {
    resolve("We got the data");
  } else {
    reject("Data not received");
  }
});

//Handle a Fulfilled Promise with then
makeServerRequest.then(result => {
  console.log(result);
});

//Handle a Rejected Promise with catch****
makeServerRequest.catch(error => {
  console.log(error)
});
```