ES6

ECMA Script or ES is a standardized version of JS.
ECMA Script ~~and~~ (ES) and JS are interchangable terms.
All major browsers follow this specification.

- Const declaration alone doesn't really protect your data from mutation.
- For this for JS provides a function — Object.freeze
- Any attempt at changing the object will be rejected without an error.
-

(1) Differences b/w var and let keywords:

→ with var keyword, you can even overwrite variable declarations without an error.

Eg- var camper = 'James';
var Camper = 'David';
console.log(camper);
Output → David.

(2) Compare Scopes of var and let keywords:

- When you declare a variable with var keyword, it is declayerd globally, or locally inside a function.
- The to let keywords behaves similarly, but with some extra features
- When you declare a variable with let keyword inside a block, statement or expression, its scope is limited to that block, statement or expression.

(3) Declare read-only variable with const variable.

const have all feature let has, with the added bonus that variables declared using const are read-only. They are a constant value, which means that once a variable is assigned with const, it cannot be reassigned.

- we cannot declare empty const variable.

(4) Mutate an array Declared with Const:

- const declaration have many use cases in JS.
- Some developers prefer to assign all their variables using const by default, unless they know they will need to reassign the value. Only in that case, they use let.

Eg
```
const s = [5, 6, 7];
s = [1, 2, 3];
s[2] = 45;
console.log(s);
```

(5) Prevent Object Mutation:
JS provides function Object.freeze to prevent data mutation.

(6) Arrow functions:

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

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

(7) Just like regular functions, we can pass arguments into arrow f";

```
const doubler = (item) => item * 2;
doubler(4);
```

Output: 8

```
const doubler = item => item * 2;
```

```
const multiplier = (item, multi) => item * multi;
multiplier(4,2);
```

(8) Default parameters for your Functions:
The default parameter kicks in when argument is not specified (it is undefied). As you can see in eg: parameter name will recieve its default value Anonymous when you do not provide a value for parameter. We can add ~~as many~~ default values for as many parameters as we want.

Example:

```
const greeting = (name = "Anonymous") => "Hello " + name;
console.log(greeting("John"));
console.log(greeting());
```

Output:

```
Hello John
Hello Anonymous
```

(9) Use rest Parameter with Function Parameters:
With rest parameter, we can create functions that take variable number of arguments.
These arguments are stored in an array that can be accessed later from inside function.

(10) Use spread Operator:
it allows us to expand arrays and other expressions in places where multiple parameters or elements are expected.

```
var arr = [6, 89, 3, 45];
var maximus = Math.max.apply(null, arr);        maximus = 89.
```

⟹ Math.max.apply(null, arr) because Math.max(arr) returns NaN.
Math.max() expects comma-seperated arguments, but not an array.
The spread operator makes this syntax much better to read and maintain.

```
const arr = [6, 89, 3, 45];
const maximus = Math.max(...arr);
```

However, spread operator only works in-place, like in an argument to a function or in an array literal.

(11) • Use Destructing Assignment to Extract Values from Objects.

```
const user = { name: 'JohnDoe', age: 34 };
const name = user.name;
const age = user.age;
```

```
const { name, age } = user;
```

(12) • to Assign Variables from Objects:
Allows to assign new variable name when extracting values.

```
const user = { name: 'JohnDoe', age: 34 };
const { name: userName, age: userAge } = user;
```

• Assign variables from Arrays:
We can access the values at any index in an array with destructing by using comma op to reach desired index:

```
const [a, b, , , c] = [1, 2, 3, 4, 5, 6];
console.log(a, b, c);
```

Output:

a	b	c
1	2	5

• Reist Parameter to Reasign Array elements:

```
const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];
console.log(a, b);
console.log(...arr);
```

Output

```
...arr
[3, 4, 5, 7]
```

It work only correctly as last elem variable in the list.

• Using Destructing Assignment to Pass an Object as a Function's Parameters.

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

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

}

(12) Create strings using Template Literals:

```
const person = {
        name : "Zodiac Hasbro",
        age : 56   };
const greeting = `Hello, my name is ${person.name}!
I am ${person.age} years old.`;
console.log(greeting);
```

Output:

Hello, my name is Zodiac Hasbro!
I am 56 years old.

(13) Write concise Object Literal Declarations using Object Property Shorthand:

```
const getMousePosition = (x,y) => ({
                                    x : x,
                                    y : y   });

const getMousePosition = (x,y) => ({ x, y });
```

(14) Use class syntax to Define a Constructor Function:

should be noted that class syntax is just syntax, not a full-fledged class-based implementation of an OOP, unlike in Java, Python, Ruby etc

```
var spaceShuttle = function (targetPlanet) {
    this.targetPlanet = targetPlanet;  }
var Zeus = new SpaceShuttle ('Jupiter');
```

- class syntax simply replaces constructor f^n creation:

```
class SpaceShuttle {
    constructor (targetPlanet) {
        this.targetPlanet = targetPlanet;  }
}
const zeus = new SpaceShuttle ('Jupiter');
```

(15) Use Getters and Setters to control Access to an Object:

- **Getters** f^n's are meant to simply return (get) value of object's private variable to user without the user directly accessing the private variable.
- **Setter** f^n's are meant to modify (set) value of object's private variable based on value passed into setter f^n. This change could involve calculation or even overwriting the previous value completely.

```
class Book {
    constructor(author) {
this._author = author; }

  // getter
  get writer() {
    return this._author; }
  // setter
this._author = updatedAuthor; }

}
const novel = new Book('anonymous');
console.log(novel.writer);
novel.writer = 'newAuthor';
console.log(novel.writer);
```

- Notice the syntax used to invoke getters and setters. They do not look like f^n's, Getters & Setters are imp. because they hide internal implementation details.
- Note: convention to precede name of private variable with an underscore (_). The practice itself doesn't make variable private.

Output:
Anonymous
newAuthor

(16) Create Module Script:

ES6 introduced a way to easily share code among JS files. We need to create a script in my HTML doc. with a type of module.

`<script type="module" src="filename.js"></script>`

A script that uses this module type can now use import & export.

- export to share code block:

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

Imagine file → math-f^n.js

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

```
export { add, subtract. };
```

We can export multiple things by repeating first eg for each thing you want to export, or by placing them all in export statement of 2nd eg.

- **Reuse JS Code Using Import :**

Import allows you to choose which part of a file or module to load.

Here's how we can import it to use in another file.

```
import { add } from './math-fn.js';
```

The ./ tells the import to look for math-fn.js file in same folder as current file.

We can add more than 1 item from file by adding them like :

```
import { add, subtract } from './math-fn.js';
```

- **Use * to Import Everything from a file :**

```
import * as myMathModule from "./math-fn.js";
```

- **Create Export Fallback with export default :**

Another export syntax - export default. It is used only if one value is being exported from file. Also used to create fallback value for file or module.

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

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

first named → function
2nd → anonymous fn.

We can't use export default with var, let or const.

- **Import DEFAULT Export :**

```
import add from "./math-fn.js";
```

(17) Create JS PROMISE :

We use it to do something usually asynchronously. When task completes, we either fulfill your promise or fail to do so. Promise is a constructor fn, so you need to use new keyword to create one.

It takes function, as its argument, with 2 parameters - resolve & reject.

```
const myPromise = new Promise((resolve, reject) => {
});
```

- Complete Promise with resolve and reject:

Promise have 3 states: pending, fulfilled, rejected.

resolve is used → want promise to succeed.
reject is used → want it to fail.

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

- Handle fulfilled Promise with then:

Then method is executed immediately after your promise is fulfilled with resolve.

```
myPromise.then(result => {
});
```

result comes from argument given to resolve method.

- Handle rejected ~~Promise~~ with catch:

Catch is method used when your promise has been rejected. It is executed immediately after a promise's reject method is called.

```
myPromise.catch(~~error~~ error => {
});
```

error is argument passed in ~~t~~ reject method.