# Quick JavaScript Refresher

Refresher.js

```
(function life() {
    code();
    debug();
    learn();
    repeat();
})();
```

#Flashcards #Programming #javascript

# Variables

Variables store data values, and JavaScript has three ways to declare them: **var**, let, and const, out of which **let** and **const** are block-scoped

Variables.js

```
let name = "Alice"; // Can be changed later
const age = 30; // Cannot be changed
```

# Data Types

Common data types include:

- **String**: Text data.
- **Number**: Numeric values.
- **Boolean**: true or false.
- **Null**: A value that represents no value.
- **Undefined**: A variable that has been declared but not assigned a value.

Datatypes.js

```
let isAdult = true; // Boolean
let score = 100;    // Number
```

# Functions

Functions allow you to group code together and execute it whenever needed.

Functions.js

```
// Declare the function
function sayHello() {
  console.log("Hello, World!");}

// Execute the function
sayHello();
```

```
Hello, World!
```

# Arrow Function

Arrow functions provide a shorter syntax.

Arrow_Function.js

```
const add = (a, b) => a + b;
console.log(add(2, 3));
```

5

# Control Flow

Control flow statements help you make decisions and repeat actions in your code.

Control_Flow.js

```
// if-else Execute code based on conditions.
let age = 18;
if (age >= 18) {
  console.log("Adult");
} else {
  console.log("Minor");
}
```

```
Adult
```

# Control Flow

Control flow statements help you make decisions and repeat actions in your code.

Control_Flow.js

```
// for, while: Execute code repeatedly.

for (let i = 0; i < 3; i++) {
  console.log(i);
}
```

```
0, 1, 2
```

# Objects

**Objects**: Store data as key-value pairs.

Objects.js

```
const person = { name: "Alice", age: 30 };
console.log(person.name);
```

```
Alice
```

# Arrays

**Arrays:** Store lists of items.

Common methods include .push(), .map(), and .filter().

Arrays.js

```
const numbers = [1, 2, 3];
console.log(numbers[0]);

numbers.push(4);// Adds 4 to the end
const doubled = numbers.map(n => n * 2);
console.log(doubled);
```

```
1
4
[ 2, 4, 6, 8 ]
```

# Template Literals

Template literals make it easier to work with strings. Use **backticks (`)** and **${}** to interpolate variables.

Template_Literals.js

```
let name = "Alice";
let greeting = `Hello, ${name}!`;
console.log(greeting);
```

```
Hello, Alice!
```

# Destructuring

Extract values from arrays or objects easily.

Destructuring.js

```
const person = { name: "Alice", age: 30 };
const { name, age } = person;
console.log(name);
```

```
Alice!
```

# Spread Operator

Expands elements of an array or object.

Spread_Operator.js

```
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5];
console.log(arr2);
```

```
[ 1, 2, 3, 4, 5 ]
```

# DOM Manipulation (Basics)

DOM_Manipulation.js

```
<h1 id="myHeading"></h1>
<script>
const element=  document.getElementById("myHeading");
element.innerText = "Old Text";
element.addEventListener('click', () => {
  element.innerText = "New Text";
});
</script>
```

**Old Text**

**New Text**

# Promise

Promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. A promise can be in one of three states:

- Pending: The operation has not finished yet.
- Resolved (Fulfilled): The operation completed successfully.
- Rejected: The operation failed.

Promise.js

```
const myPromise = new Promise(resolve => {
  setTimeout(() => resolve("Done!"), 1000);
});
myPromise.then(result => console.log(result));
```

```
Promise { <pending> }
'Done!'
```

# async/await

Modern way to write asynchronous code, which is easier to read and maintain. How async/await Works

- **async** functions automatically return a promise.
- **await** is used to pause the execution of the function until the promise resolves.

async_await.js

```
async function run() {
  await new Promise(resolve => setTimeout(resolve, 1000));
  console.log("Done!");
}
run();
```

```
Promise { <pending> }
'Done!'
```