var, let, const: var - function-scoped, declares a variable globally; let - block-scoped, can be reassigned; const - block-scoped, can not be reassigned, elements of arrays and objects can be reassigned.
Null vs undefined: Null is intentional absence of any value. Undefined is a value of a variable which is declared and not initialized.

JS prototype:

```
function Person(name) {
  this.name = name;
}
Person.prototype.sayHello = function() {
  return `My name is ${this.name}.`;
};
let person1 = new Person('Jack');
console.log(person1.sayHello()); // My name is Jack.

String.prototype.upper = function() {
  return this.toUpperCase();
};
console.log("abc".upper()); // "ABC"

Array.prototype.first3 = function() {
  return this.slice(0, 3);
};
console.log([1,2,3,4,5].first3()); // [1,2,3]
```

JS curried function:

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

let add5 = add(5);
console.log(add5(3)); // 8
```

JS OOP:

```
class Car {
  constructor(make) {
    this.make = make;
    this.isOn = false;
  }
  start() {
    this.isOn = true;
    console.log(`${this.make} started.`);
  }
  stop() {
    this.isOn = false;
    console.log(`${this.make} stopped.`);
  }
  isRunning() {
    return this.isOn;
  }
}
class ElectricCar extends Car {
  constructor(make, batteryCapacity) {
    super(make);
    this.batteryCapacity = batteryCapacity;
  }
  charge() {
    console.log(`${this.make} is charging.`);
  }
}
let myElCar = new ElectricCar('EV', '100 kWh');
myElCar.start(); // EV started.
console.log(myElCar.isRunning()); // true
myElCar.stop(); // EV stopped.
console.log(myElCar.isRunning()); // false
myElCar.charge(); // EV is charging.
```

Spread ...

```
let nums = [1, 2];
let newNums = [...nums, 3]; // newNums: [1, 2, 3]
let obj = { a: 1 };
let newObj = {...obj, b: 2}; // newObj: {a:1, b:2}
let obj2 = { a: 1, b: 2 };
let updObj = {...obj1, b: 5}; // updObj: {a:1, b:5}
```

Optional Chaining (?.): Allows to access properties that may not exist without causing an error.

```
let person = {name: 'Jack', address: {city: 'York'}};
console.log(person?.address?.city); // Output: 'York'
console.log(person?.age); // Output: undefined
let arr = [[1, 2], [3, 4]];
console.log(arr[0]?.[1]); // Output: 2
console.log(arr[2]?.[0]); // Output: undefined
```

Nullish Coalescing (??): Set default value if left operand is null or undefined.

```
let defaultValue = 'default';
let value1 = null;
let value2 = undefined;
let value3 = 'not null';
console.log(value1 ?? defaultValue); // default
console.log(value2 ?? defaultValue); // default
console.log(value3 ?? defaultValue); // not null
```

Object Destructuring: Extracts properties from objects and binds them to variables.

```
let person = { name: 'J', age: 30 };
let { name, age } = person; // name='J', age=30
```

Arrow Functions: A concise way to write function expressions.

```
let add = (a, b) => a + b;
```

Template Literals: Allow embedding expressions into string literals.

```
let name = 'Jack';
let greeting = `Hi, ${name}!`;
```

Object.keys(): Returns an array of a given object's own enumerable property names.

```
let person = { a: 'Jack', b: 30 };
let keys = Object.keys(person); // keys=['a','b']
```

Object.values(): Returns an array of a given object's own enumerable property values.

```
let person = { name: 'J', age: 30 };
let vals = Object.values(person); // vals=['J', 30]
```

Object.entries(): Returns an array of an object's own enumerable property [key, value] pairs.

```
let obj = { a: 1, b: 2 };
let e = Object.entries(obj); // e=[['a',1],['b',2]]
```

forEach(): Executes a function once for each array element.

```
[1,2,3].forEach(num => console.log(num)); // 1,2,3
```

map(): Transforms each element in an array and returns a new array.

```
let dbl = [1,2].map(num => num * 2); // dbl=[2,4]
```

filter(): Returns a new array with elements that meet a condition.

```
let even = [1,2,3].filter(n => n%2===0); // even=[2]
```

reduce(): Reduces an array to a single value.

```
let s = [1,2,3].reduce((acc,curr)=>acc+curr,0);//s=6
```

find(): Returns the first element in an array that meets a condition.

```
let lessThan3 = [1,2,3].find(num => num < 3); //
lessThan3 is 1
```

some(): Checks if at least 1 element in an array meets a condition.

```
let hasLessThan2 = [1,2,3].some(num => num < 2); //
hasLessThan2 is true
```

every(): Checks if all elements in the array meet a condition.

```
let allPos = [1,2,3].every(n => n > 0);//allPos=true
```

flat(): Flatten the array.

```
let nested = [1, 2, [3, 4, [5, 6]]];
let flat = nested.flat(); // flat=[1,2,3,4,[5,6]]
```

slice(): Returns a shallow copy of a portion of an array.

```
let sliced = [1,2,3].slice(0, 2); // sliced: [1,2]
```

indexOf(): searches for a specific value in the array via ===. Returns index or -1 if not found.

```
let index = [1,2,3].indexOf(3); // index is 2 (index
of the value 3)
```

findIndex(): finds the first element in the array via a condition. Returns index or -1 if not found.

```
let index = [1, 2, 3].findIndex(num => num > 2); //
index=2 (index of the first element greater than 2)
```

push(): Adds one or more elements to the end of an array and returns the new array length.

```
let nums = [1, 2];
let len = nums.push(7); // len=3, nums=[1, 2, 7]
```

pop(): Removes the last element from an array and returns it.

```
let nums = [1, 2, 3];
let last = nums.pop(); // last=3, nums=[1,2]
```

shift(): Removes the first element from an array and returns it.

```
let nums = [1, 2, 3];
let first = nums.shift(); // first=1, nums=[2,3]
```

unshift(): Adds one or more elements to the start of an array and returns the new array length.

```
let nums = [2, 3];
let len = nums.unshift(1); // len=3, nums=[1,2,3]
```

sort(): Sorts the elements of an array.

```
let nums = [3, 1, 2];
nums.sort(); // nums is [1, 2, 3]
```

join(): Joins all elements of an array into a string.

```
let joined = [1,2,3].join(', '); // joined="1,2,3"
```

includes(): Checks if an array includes a certain value.

```
let hasTwo = [1,2,3].includes(2); // hasTwo is true
```

reverse(): Reverses an array in place.

```
let nums = [1, 2, 3];
nums.reverse(); // nums=[3, 2, 1]
```

toString(): Returns a string representing the specified array and its elements.

```
let str = [1,2,3].toString(); // str="1,2,3"
```

replace(): Replaces a specified value with another value in a string.

```
let upd = '1233'.replace('3', '7'); // upd="1273"
```

querySelector(): Access DOM element.

```
let btnElem = document.querySelector('.btn');
let targetElem = document.querySelector('.target');

targetElem.style.backgound = "red";
targetElem.style.padding = "10px 20px";
targetElem.style.marginRight = "10px";

btnElem.addEventListener('click', function() {
    targetElem.innerText = 'New Text';
});
```

setTimeout(): It will only alert Hello after a 3 second delay.

```
let sayHello = () => {
  alert('Hello');
};
setTimeout(sayHello, 3000);
```

setInterval(): It will say Hello every 3 seconds.

```
let sayHello = (message) => {
  alert(message);
};
setInterval(() => {
  sayHello('Hello');
}, 3000);
```

Async/Await: A simpler syntax for writing asynchronous code.

```
function delay(ms) {
  return new Promise(resolve =>
setTimeout(resolve,ms));
}

async function asyncFunction() {
  console.log("Start");
  await delay(2000); // Pause execution for 2 seconds
  console.log("End");
}

asyncFunction();

// 2nd example:
async function fetchData() {
  let response = await
fetch('https://api.example.com/data');
  let data = await response.json();
  return data;
}
```

Closure: is a function that has access to the outer (enclosing) function's variables. Closure "remember" the environment in which they were created even though outerFunction has already completed execution.

```
function outerFunction() {
    let outerVariable = 'From the outer func';

    return function innerFunction() {
        console.log(outerVariable);
    }
}

let closureExample = outerFunction();
closureExample(); // Output: "From the outer func"
```