## Explain the differences between `==` and `===`.

The `==` operator checks for equality but performs type coercion if the types differ, while `===` checks for both value and type.  
  
Example:  
```javascript  
console.log(5 == "5"); // true (type coercion)  
console.log(5 === "5"); // false (strict comparison)  
```

## What is event delegation, and how does it function in JavaScript?

Event delegation is a technique where a single event listener is added to a parent element to handle events on its child elements. This works because of event bubbling.  
  
Example:  
```javascript  
document.getElementById('parent').addEventListener('click', (event) => {  
 if (event.target.tagName === 'BUTTON') {  
 console.log('Button clicked:', event.target.textContent);  
 }  
});  
```

## Can you provide an illustration of how ES6 has altered the approach to working with `this` in JavaScript?

In ES6, arrow functions do not have their own `this`. Instead, they inherit `this` from their surrounding lexical scope.  
  
Example:  
```javascript  
function Person() {  
 this.age = 0;  
 setInterval(() => {  
 this.age++;  
 console.log(this.age);  
 }, 1000);  
}  
  
new Person();  
```

## Explain the concept of prototypal inheritance.

Prototypal inheritance allows objects to inherit properties and methods from another object.  
  
Example:  
```javascript  
const animal = {  
 speak: function() {  
 console.log("I am an animal");  
 },  
};  
  
const dog = Object.create(animal);  
dog.speak(); // I am an animal  
```

## Differentiate between a variable that is null, undefined, or undeclared.

- `null`: Explicitly assigned to indicate "no value."  
- `undefined`: A variable declared but not assigned a value.  
- `undeclared`: A variable that has not been declared.  
  
Example:  
```javascript  
let a;  
console.log(a); // undefined  
let b = null;  
console.log(b); // null  
console.log(c); // ReferenceError: c is not defined  
```

## Define what a closure is and describe its uses and advantages.

A closure is a function that retains access to its outer scope even after the outer function has executed.  
  
Example:  
```javascript  
function outer() {  
 let count = 0;  
 return function inner() {  
 count++;  
 console.log(count);  
 };  
}  
  
const counter = outer();  
counter(); // 1  
counter(); // 2  
```  
Uses:  
- Data encapsulation  
- Function factories  
- Maintaining state

## Explain the primary distinction between the Array.forEach() loop and Array.map() method, as well as when to choose one over the other.

- `forEach`: Executes a function for each array element but does not return a new array.  
- `map`: Transforms each element and returns a new array.  
  
Example:  
```javascript  
const arr = [1, 2, 3];  
arr.forEach((num) => console.log(num)); // Logs 1, 2, 3  
  
const squared = arr.map((num) => num \*\* 2);  
console.log(squared); // [1, 4, 9]  
```  
Use `forEach` for side effects and `map` when you need a transformed array.

## What is a common scenario for employing anonymous functions?

Anonymous functions are often used for short-lived operations where you do not need to reuse the function elsewhere.  
  
Example:  
```javascript  
setTimeout(function() {  
 console.log('This runs after 1 second');  
}, 1000);  
```

## Distinguish between host objects and native objects.

- \*\*Host objects\*\* are objects provided by the host environment (e.g., browsers or Node.js) like `document`, `window`, and `fs`.  
- \*\*Native objects\*\* are built-in objects in JavaScript like `Object`, `Array`, `Function`, etc.  
  
Example:  
```javascript  
console.log(window); // Host object (browser)  
console.log([]); // Native object (Array)  
```

## Clarify the distinctions among 'function User(){}', 'var user = User()', and 'var user = new User()'.

- `function User(){}`: Defines a constructor function.  
- `var user = User()`: Calls the constructor function but does not create an object (this could return undefined).  
- `var user = new User()`: Creates an instance of the `User` object.  
  
Example:  
```javascript  
function User(name) {  
 this.name = name;  
}  
var user = new User('Alice');  
console.log(user.name); // Alice  
```

## Can you elucidate the purposes of Function.call and Function.apply, along with their notable differences?

- `call`: Calls a function with a specified `this` value and arguments passed individually.  
- `apply`: Similar to `call`, but arguments are passed as an array.  
  
Example:  
```javascript  
function greet() {  
 console.log(`Hello, ${this.name}`);  
}  
  
const person = { name: 'Bob' };  
  
greet.call(person); // Hello, Bob  
greet.apply(person); // Hello, Bob  
```

## Describe the Function.prototype.bind method.

The `bind()` method creates a new function that, when called, has its `this` set to the provided value.  
  
Example:  
```javascript  
function greet() {  
 console.log(`Hello, ${this.name}`);  
}  
  
const person = { name: 'Alice' };  
const greetAlice = greet.bind(person);  
greetAlice(); // Hello, Alice  
```

## Explain the differences between feature detection, feature inference, and utilizing the User Agent (UA) string.

- \*\*Feature detection\*\* checks if a browser supports a specific feature (recommended approach).  
- \*\*Feature inference\*\* assumes support for features based on other known characteristics.  
- \*\*User Agent (UA) string\*\* checks the browser's user-agent string (not reliable due to its easily spoofable nature).  
  
Example:  
```javascript  
if ('fetch' in window) {  
 console.log('Fetch API supported');  
}  
```

## Define the concept of 'hoisting.'

Hoisting is a JavaScript behavior where variable and function declarations are moved to the top of their containing scope during compile time.  
  
Example:  
```javascript  
console.log(a); // undefined  
var a = 5;  
```  
  
Hoisting only moves the declarations, not the assignments.

## What is type coercion, and what are some common pitfalls associated with relying on it in JavaScript code?

Type coercion is JavaScript's automatic conversion of data types. It can lead to unexpected results.  
  
Example:  
```javascript  
console.log(5 + '5'); // "55" (number coerced to string)  
console.log('5' - 5); // 0 (string coerced to number)  
```  
Pitfall: relying on implicit coercion can lead to errors or bugs in code.

## Describe event bubbling and event capturing.

Event bubbling occurs when an event starts at the target element and bubbles up to its ancestors. Event capturing is the opposite, where the event starts at the root and is captured by ancestor elements.  
  
Example:  
```javascript  
// Event bubbling  
document.querySelector('button').addEventListener('click', function() {  
 alert('Button clicked!');  
});  
  
// Event capturing  
document.querySelector('button').addEventListener('click', function() {  
 alert('Captured button click!');  
}, true);  
```

## JavaScript Data Types

Question: What are the different data types in JavaScript?  
Answer:  
- Number  
- String  
- Boolean  
- Undefined  
- Null  
- Object  
- Symbol  
- BigInt  
  
Example:  
```javascript  
let num = 42; // Number  
let str = "Hello"; // String  
let bool = true; // Boolean  
let und; // Undefined  
let nul = null; // Null  
let obj = { name: "Alice" }; // Object  
let sym = Symbol("id"); // Symbol  
let bigInt = 123n; // BigInt  
```

## Let, Var, and Const

Question: What is the difference between `let`, `var`, and `const`?  
Answer:  
- `var`: Function-scoped, can be redeclared.  
- `let`: Block-scoped, cannot be redeclared.  
- `const`: Block-scoped, cannot be reassigned.  
  
Example:  
```javascript  
var x = 10;  
let y = 20;  
const z = 30;  
```

## Scope

Question: Explain different scopes in JavaScript.  
Answer:  
- Function scope: Variables declared with `var` inside a function are limited to that function.  
- Block scope: Variables declared with `let` and `const` are limited to the block.  
  
Example:  
```javascript  
function test() {  
 var local = "Local Scope";  
 if (true) {  
 let blockScoped = "Block Scope";  
 }  
 console.log(local);  
}  
```

## Callbacks

Question: What is a callback function in JavaScript?  
Answer:  
A callback is a function passed as an argument to another function, to be executed later.  
  
Example:  
```javascript  
function fetchData(callback) {  
 setTimeout(() => {  
 console.log("Data fetched");  
 callback();  
 }, 1000);  
}  
  
fetchData(() => {  
 console.log("Callback executed");  
});  
```

## Call, Apply, and Bind

Question: Explain the difference between `call`, `apply`, and `bind` in JavaScript.  
Answer:  
- `call`: Invokes a function with a given `this` context and arguments passed individually.  
- `apply`: Similar to `call`, but arguments are passed as an array.  
- `bind`: Returns a new function with a given `this` context and arguments.  
  
Example:  
```javascript  
const person = {  
 name: "Alice",  
 greet: function (age) {  
 console.log(`Hello, my name is ${this.name}, and I am ${age} years old.`);  
 },  
};  
  
const anotherPerson = { name: "Bob" };  
  
person.greet.call(anotherPerson, 25); // Hello, my name is Bob, and I am 25 years old.  
person.greet.apply(anotherPerson, [30]); // Hello, my name is Bob, and I am 30 years old.  
  
const boundGreet = person.greet.bind(anotherPerson, 35);  
boundGreet(); // Hello, my name is Bob, and I am 35 years old.  
```

## Try-Catch

Question: How does `try-catch` work in JavaScript?  
Answer:  
The `try-catch` block is used to handle exceptions gracefully.  
  
Example:  
```javascript  
try {  
 let result = 10 / 0;  
 console.log(result);  
 throw new Error("An error occurred");  
} catch (err) {  
 console.error(err.message);  
} finally {  
 console.log("Execution completed");  
}  
```

## Property Descriptors

Question: What are property descriptors in JavaScript?  
Answer:  
Property descriptors define the attributes of an object property (writable, configurable, enumerable).  
  
Example:  
```javascript  
const obj = {};  
Object.defineProperty(obj, "prop", {  
 value: 42,  
 writable: false,  
 configurable: true,  
 enumerable: true,  
});  
  
console.log(obj.prop); // 42  
obj.prop = 50; // No effect since writable is false  
console.log(obj.prop); // 42  
```

## Loops and Control Structures

Question: Explain common loops and control structures in JavaScript.  
Answer:  
1. \*\*`for` loop\*\*: Iterates a fixed number of times.  
2. \*\*`while` loop\*\*: Runs while a condition is true.  
3. \*\*`do...while` loop\*\*: Runs at least once before checking the condition.  
4. \*\*`if-else`\*\*: Conditional branching.

5 for—in for--of  
  
Example:  
```javascript  
for (let i = 0; i < 3; i++) {  
 console.log(i); // 0, 1, 2  
}  
  
let x = 0;  
while (x < 3) {  
 console.log(x); // 0, 1, 2  
 x++;  
}  
  
let y = 0;  
do {  
 console.log(y); // 0, 1, 2  
 y++;  
} while (y < 3);  
  
if (x > 2) {  
 console.log("Greater");  
} else {  
 console.log("Smaller");  
}  
```

## Array Methods

Question: Explain some commonly used array methods., what is difference between these map,foreeach?  
Answer:  
1. \*\*`forEach`\*\*: Iterates through elements without returning a value.  
2. \*\*`map`\*\*: Returns a new array by transforming each element.  
3. \*\*`filter`\*\*: Returns a new array of elements that satisfy a condition.  
4. \*\*`reduce`\*\*: Reduces an array to a single value.  
  
Example:  
```javascript  
const arr = [1, 2, 3];  
  
arr.forEach(num => console.log(num)); // 1, 2, 3  
  
const squared = arr.map(num => num \*\* 2); // [1, 4, 9]  
  
const evens = arr.filter(num => num % 2 === 0); // [2]  
  
const sum = arr.reduce((acc, num) => acc + num, 0); // 6  
```

## Object Methods: `keys`, `entries`, `assign`, etc.

Question: What are common object methods in JavaScript?  
Answer:  
1. \*\*`Object.keys`\*\*: Returns an array of keys.  
2. \*\*`Object.entries`\*\*: Returns an array of key-value pairs.  
3. \*\*`Object.assign`\*\*: Copies properties from source objects to a target object.  
4. \*\*`Object.create`\*\*: Creates a new object with a specified prototype.  
  
Example:  
```javascript  
const obj = { a: 1, b: 2 };  
  
console.log(Object.keys(obj)); // ['a', 'b']  
console.log(Object.entries(obj)); // [['a', 1], ['b', 2]]  
console.log(Object.assign({}, obj, { c: 3 })); // { a: 1, b: 2, c: 3 }  
  
const protoObj = { greet() { console.log("Hello"); } };  
const newObj = Object.create(protoObj);  
newObj.greet(); // Hello  
```

## What distinguishes an 'attribute' from a 'property'?

- \*\*Attribute\*\* is the initial value set in HTML, like `id` or `class`.  
- \*\*Property\*\* is the value that JavaScript interacts with, typically an object value.  
  
Example:  
```html  
<input id="myInput" type="text" value="initial value">  
```  
  
```javascript  
const input = document.getElementById('myInput');  
console.log(input.value); // property  
console.log(input.getAttribute('value')); // attribute  
```

What is Hoisting?

Answer:

Hoisting is JavaScript's default behavior of moving variable and function declarations to the top of their containing scope during the compile phase. Only the declarations are hoisted, not the initializations.

console.log(a); // undefined

var a = 5;

hoistedFunction(); // "Hoisted!"

function hoistedFunction() {

  console.log("Hoisted!");

}

// Not hoisted

console.log(b); // ReferenceError: Cannot access 'b' before initialization

let b = 10;

What is a Closure?

Answer:

A closure is a function that "remembers" the variables from its outer scope even after the outer function has finished executing. This allows the function to access and manipulate variables in its lexical environment.

function outerFunction() {

    let counter = 0;

    return function innerFunction() {

      counter++;

      console.log(counter);

    };

  }

  const increment = outerFunction();

  increment(); // 1

  increment(); // 2

  increment(); // 3

  What is preventDefault()?

Answer:

preventDefault() prevents the browser's default behavior for a specific event.

document.getElementById("myForm").addEventListener("submit", (event) => {

    event.preventDefault(); // Prevents the form from submitting

    console.log("Form submission prevented");

  });

  Explain Synchronization and Asynchronization in JavaScript

  Answer:

Synchronous Code: Executes sequentially. Each line waits for the previous one to finish.

Asynchronous Code: Allows multiple tasks to run without waiting for each other to complete, using mechanisms like callbacks, promises, or async/await.

console.log("Start");

console.log("End");

// Output: Start, End

console.log("Start");

setTimeout(() => {

  console.log("Inside Timeout");

}, 1000);

console.log("End");

// Output: Start, End, Inside Timeout

What is this in JavaScript?

Answer:

The this keyword refers to the object that is executing the current function. Its value depends on how the function is invoked.

Key Scenarios:

Global Scope:

console.log(this); // In browsers, 'this' refers to the global object (window).

Object Method:

const obj = {

  name: "Alice",

  greet() {

    console.log(this.name); // "Alice"

  }

};

obj.greet();

Arrow Functions:

Arrow functions do not bind this but inherit it from their enclosing lexical scope.

const obj = {

    name: "Bob",

    greet: () => {

      console.log(this.name); // undefined (lexical 'this')

    }

  };

  obj.greet();

Using call, apply, or bind:

function greet() {

    console.log(this.name);

  }

  const person = { name: "test" };

  greet.call(person); // "test"

  greet.apply(person); // "test"

  const boundGreet = greet.bind(person);

  boundGreet(); // "test"

  How JavaScript Works: Execution Flow

JavaScript executes code using the Execution Context and the Call Stack.

Execution Context (EC):

An environment where JavaScript code is executed. Two types:

Global Execution Context (GEC): Created when the script starts.

this points to the global object.

Global variables and functions are stored here.

Function Execution Context (FEC): Created whenever a function is called.

Contains its own scope, arguments, and this.

Phases of Execution Context:

Creation Phase (Memory Allocation):

Variables declared with var are hoisted and initialized to undefined.

Functions are hoisted with their definitions.

Execution Phase (Code Execution):

Code runs line by line, and variables are assigned values.