1. Explain the concept of closures in JavaScript with an example.

A **closure** is a function that retains access to its lexical scope (variables declared outside of it) even after the function has returned. This allows inner functions to access variables from their outer functions, even after the outer function has finished execution.

Example:

javascript
Copy
```
function outer() {
  let count = 0;

  return function inner() {
    count++;
    console.log(count);
  };
}

const counter = outer();
counter(); // 1
counter(); // 2
counter(); // 3
```

Here, the `inner` function forms a closure because it has access to the `count` variable from the `outer` function.

---

2. What is the difference between '==' and '===' in JavaScript?

- `==`: This is the **loose equality** operator. It compares values for equality but performs **type coercion**, meaning it converts the operands to the same type before comparing them.
- `===`: This is the **strict equality** operator. It compares both the **value and type**, and **does not perform type coercion**.

Example:

javascript
Copy

```
console.log(5 == '5');  // true (because type coercion happens)
console.log(5 === '5'); // false (different types)
```

---

3. How does the JavaScript event loop work?

The **event loop** handles asynchronous code in JavaScript. It allows the single-threaded execution model to run non-blocking tasks by delegating tasks to be executed later. Here's how it works:

1. **Call Stack**: Holds the currently executing code.
2. **Web APIs**: Handle async tasks like `setTimeout`, AJAX requests, etc.
3. **Callback Queue**: Once async tasks are complete, their callbacks are placed in the queue.
4. **Event Loop**: Checks if the call stack is empty. If it is, the event loop pushes the first task in the callback queue to the stack.

Microtasks (Promises) are executed before tasks in the callback queue.

---

4. What is hoisting in JavaScript?

Hoisting is a JavaScript mechanism where variable and function declarations are moved to the top of their containing scope during the compile phase. However, only the declarations are hoisted, not the initializations.

Example:

javascript
Copy
```
console.log(x); // undefined
var x = 5;

function example() {
  console.log(y); // undefined
  var y = 10;
}
```

---

5. Explain 'this' keyword in different contexts.

The value of `this` depends on how the function is called:

- In **global scope**, `this` refers to the global object (`window` in browsers).
- In **object methods**, `this` refers to the object the method is called on.
- In **constructor functions**, `this` refers to the newly created instance.
- In **arrow functions**, `this` is **lexically bound**, meaning it takes the value of `this` from the surrounding context.

Example:

javascript
Copy
```
const obj = {
  name: 'John',
  greet: function() {
    console.log(this.name); // 'this' refers to obj
  }
};

obj.greet(); // Output: John
```

---

6. What is the difference between var, let, and const?

- `var`: **Function-scoped** or **global-scoped**, can be redeclared and updated. Hoisted to the top of the scope.
- `let`: **Block-scoped**, can be updated but not redeclared within the same block.
- `const`: **Block-scoped**, cannot be updated or redeclared. Must be initialized at declaration.

Example:

javascript
Copy
```
let x = 10;
const y = 20;
var z = 30;
```

```
x = 15; // Valid
y = 25; // Error: Assignment to constant variable
z = 35; // Valid
```

---

7. How does prototypal inheritance work in JavaScript?

JavaScript uses **prototypal inheritance**, meaning objects can inherit properties and methods from other objects. Every object has a prototype, and when a property is accessed, JavaScript checks the object itself first, then looks at its prototype chain.

Example:

javascript
Copy
```
const animal = {
  speak: function() {
    console.log('Animal speaks');
  }
};

const dog = Object.create(animal);
dog.speak(); // 'Animal speaks'
```

---

8. What are promises and how are they used?

A **Promise** is an object representing the eventual completion (or failure) of an asynchronous operation. It has three states: `pending`, `resolved`, and `rejected`.

Example:

javascript
Copy
```
const myPromise = new Promise((resolve, reject) => {
  let success = true;
  if(success) {
    resolve("Task succeeded");
  } else {
    reject("Task failed");
  }
});
```

```
myPromise
  .then(result => console.log(result)) // "Task succeeded"
  .catch(error => console.log(error)); // "Task failed"
```

---

9. Explain async/await with examples.

`async/await` simplifies handling asynchronous code, making it look synchronous.

- **async**: Marks a function as asynchronous and returns a Promise.
- **await**: Pauses execution until the Promise resolves.

Example:

javascript
Copy

```
async function fetchData() {
  const response = await fetch('https://api.example.com');
  const data = await response.json();
  console.log(data);
}

fetchData();
```

---

10. What is the difference between synchronous and asynchronous code?

- **Synchronous code** runs in sequence, blocking the execution of subsequent code until it completes.
- **Asynchronous code** runs independently of the main program flow, allowing other code to execute while waiting for completion.

Example:

javascript
Copy

```
// Synchronous
console.log("Start");
console.log("End");
```

```
// Asynchronous
setTimeout(() => {
  console.log("Hello from setTimeout");
}, 1000);
console.log("End");
```

---

11. What are arrow functions and how do they differ from regular functions?

Arrow functions provide a shorter syntax for writing functions and do not have their own `this`. They inherit `this` from the surrounding context (lexical `this`).

Example:

javascript
Copy

```
const add = (a, b) => a + b;  // Arrow function

function subtract(a, b) {
  return a - b; // Regular function
}
```

---

12. What is a callback function? Give an example.

A **callback function** is a function passed as an argument to another function and executed later. It is typically used in asynchronous operations.

Example:

javascript
Copy

```
function fetchData(callback) {
  setTimeout(() => {
    callback("Data received");
  }, 1000);
}

fetchData(data => {
  console.log(data); // "Data received"
});
```

---

13. How does JavaScript handle memory management and garbage collection?

JavaScript uses **automatic memory management** and performs **garbage collection**. It removes objects that are no longer in use (i.e., objects that are unreachable). The garbage collector works by finding and clearing objects that are not referenced anymore.

---

14. What are IIFEs (Immediately Invoked Function Expressions)?

An **IIFE** is a function that is defined and immediately executed. It helps create a **local scope** to avoid polluting the global namespace.

Example:

javascript
Copy
```javascript
(function() {
  console.log("IIFE executed");
})();
```

---

15. What is the use of bind(), call(), and apply() methods?

- **bind()**: Creates a new function that, when called, has its `this` set to the provided value.
- **call()**: Calls a function with a specified `this` value and arguments passed individually.
- **apply()**: Similar to `call()`, but accepts arguments as an array.

Example:

javascript
Copy
```javascript
const obj = { name: 'John' };
function greet() {
  console.log(`Hello, ${this.name}`);
}
```

```
greet.call(obj);  // "Hello, John"
greet.apply(obj); // "Hello, John"
const boundGreet = greet.bind(obj);
boundGreet();     // "Hello, John"
```

---

16. What are JavaScript modules and how do you export/import them?

JavaScript modules allow you to split your code into smaller pieces. You can use **`export`** to export functions, objects, or values, and **`import`** to bring them into another file.

Example:

javascript
Copy
```
// math.js
export function add(a, b) {
  return a + b;
}

// app.js
import { add } from './math.js';
console.log(add(2, 3)); // 5
```

---

17. What is event delegation and why is it useful?

Event delegation involves attaching a single event listener to a parent element rather than individual child elements. This improves performance and ensures that dynamically added elements can still respond to events.

Example:

javascript
Copy
```
document.getElementById('parent').addEventListener('click',
function(event) {
  if (event.target.matches('button')) {
    console.log('Button clicked');
  }
});
```

---

18. What are higher-order functions in JavaScript?

A **higher-order function** is a function that either takes one or more functions as arguments or returns a function as its result.

Example:

javascript
Copy

```javascript
function greet(name) {
  return function(message) {
    console.log(`${message}, ${name}`);
  };
}

const greetJohn = greet("John");
greetJohn("Hello"); // "Hello, John"
```

---

19. What is the difference between deep copy and shallow copy?

- **Shallow copy**: Copies only the first level of an object. Nested objects are still referenced.
- **Deep copy**: Recursively copies all levels of an object, creating independent copies of nested objects.

Example:

javascript
Copy

```javascript
const obj1 = { a: 1, b: { c: 2 } };
const shallow = Object.assign({}, obj1);
const deep = JSON.parse(JSON.stringify(obj1));

obj1.b.c = 3;
console.log(shallow.b.c); // 3 (shallow copy is affected)
console.log(deep.b.c);    // 2 (deep copy remains unaffected)
```

---

20. Explain debouncing and throttling with examples.

- **Debouncing** ensures that a function is not called repeatedly within a short period. It's useful for scenarios like resizing or scrolling events.
- **Throttling** ensures a function is called at a regular interval, no matter how often the event is triggered.

Debouncing Example:

javascript
Copy
```
let timeout;
function debounce(func, delay) {
  clearTimeout(timeout);
  timeout = setTimeout(func, delay);
}
```

Throttling Example:

javascript
Copy
```
let lastTime = 0;
function throttle(func, delay) {
  const now = new Date().getTime();
  if (now - lastTime >= delay) {
    func();
    lastTime = now;
  }
}
```