JS Notes

Question 1: What is front-end development and how is it different from back-end development?

Front-end development is the practice of building the user-facing parts of a website or application. It involves everything users see and interact with directly in their web browser. This includes the layout, design, structure, animations, and interactivity. Front-end developers use technologies such as:

- **HTML (HyperText Markup Language):** Defines the structure and content of web pages.
- **CSS (Cascading Style Sheets):** Styles and visually formats the HTML content.
- **JavaScript:** Adds interactivity, dynamic content, and logic to web pages.

Front-end development focuses on:

- Creating responsive layouts that adapt to different devices (mobile, tablet, desktop).
- Ensuring accessibility for users with disabilities.
- Optimizing performance for fast load times and smooth interactions.
- Implementing UI/UX designs provided by designers.

Back-end development refers to the server-side logic and infrastructure that powers web applications. It is responsible for:

- Processing requests from the front-end.
- Managing databases and data storage.
- Handling authentication, authorization, and security.
- Implementing business logic and APIs.

Back-end developers use languages and frameworks such as Node.js, Python (Django, Flask), Java (Spring), Ruby (Rails), PHP, and databases like MySQL, PostgreSQL, or MongoDB.

Key Differences:

Criteria	Front-End	Back-End
Technologies	HTML, CSS, JavaScript, React, etc.	Node.js, Python, Java, SQL, etc.
Role	Interface & User Interaction	Logic, Data Management, API
Executes On	Browser	Server
Tools	Chrome DevTools, Webpack, etc.	Postman, Docker, REST APIs

Example:

- When you visit an e-commerce site, the product listings, images, and shopping cart UI are built by front-end developers. When you place an order, the back-end processes your payment, updates inventory, and sends confirmation emails.

Edge Case:

- **Full-Stack Development:** Some frameworks (like Next.js, Nuxt.js) allow developers to work on both front-end and back-end in a single project, blurring the lines between the two roles.

Question 2: What are variables in JavaScript? How do `var`, `let`, and `const` differ?

Variables in JavaScript are containers for storing data values. They allow you to store, retrieve, and manipulate data in your programs. JavaScript provides three main ways to declare variables: `var`, `let`, and `const`.

`var`

- **Scope:** Function-scoped. Accessible anywhere within the function where it is declared.
- **Hoisting:** Declarations are hoisted to the top of their scope and initialized as `undefined`.
- **Redeclaration:** Can be redeclared within the same scope.

```
function testVar() {
  console.log(x); // undefined (hoisted)
  var x = 10;
  console.log(x); // 10
}
testVar();
```

`let`

- **Scope:** Block-scoped. Only accessible within the block (`{}`) where it is declared.
- **Hoisting:** Declarations are hoisted but not initialized. Accessing before declaration throws a ReferenceError.
- **Redeclaration:** Cannot be redeclared in the same scope.

```
function testLet() {
  // console.log(y); // ReferenceError
  let y = 20;
  console.log(y); // 20
}
testLet();
```

`const`

- **Scope:** Block-scoped, like `let`.
- **Initialization:** Must be initialized at the time of declaration.
- **Reassignment:** Cannot be reassigned, but the contents of objects/arrays can be mutated.

```
const obj = { name: "Alice" };
obj.name = "Bob"; // Allowed
// obj = {}; // Error: Assignment to constant variable
```

Edge Cases:

- `const` only prevents reassignment of the variable binding, not the mutation of the object/array it points to.
- Using `var` in loops with asynchronous code can cause unexpected results due to function scoping.

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000); // Prints 3, 3, 3
}
for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000); // Prints 0, 1, 2
}
```

Best Practice:

- Use `let` and `const` for modern JavaScript. Prefer `const` by default, and use `let` only when you need to reassign the variable.

Question 3: What is closure in JavaScript? Provide a real-life use case.

A **closure** is a feature in JavaScript where an inner function has access to the outer (enclosing) function's variables, even after the outer function has finished executing. Closures allow functions to retain access to their lexical scope.

Key Points:

- Functions "remember" the environment in which they were created.
- Useful for data privacy and creating private variables.

Example:

```
function makeCounter() {
  let count = 0;
  return function () {
    return ++count;
  };
}
const counter = makeCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

Here, the inner function returned by `makeCounter` retains access to the `count` variable, even after `makeCounter` has finished executing.

Real-Life Use Case: Private State

```
function createBankAccount(initialBalance) {
  let balance = initialBalance;
  return {
    deposit(amount) {
      balance += amount;
    },
    withdraw(amount) {
      if (amount <= balance) balance -= amount;
    },
    getBalance() {
      return balance;
    },
  };
}
const myAcc = createBankAccount(1000);
myAcc.deposit(500);
console.log(myAcc.getBalance()); // 1500
```

Diagram (Mermaid):

```
flowchart TD
    A[Global Scope] --> B[makeCounter Scope]
    B --> C[Inner Function]
    C --> D[Access to count = 0]
```

Edge Cases:

- Closures can cause memory leaks if not handled properly (e.g., in event listeners).
- Can be confusing in loops with `var`. Use `let` for block scope.

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000); // 3, 3, 3
}
for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000); // 0, 1, 2
}
```

Question 4: What is the difference between `undefined` and `null` in JavaScript?

Both `undefined` and `null` represent the absence of a value, but they are used in different contexts and have different meanings.

`undefined`

- Automatically assigned to variables that have been declared but not initialized.
- Returned by functions with no explicit return value.
- Indicates "value is missing or not assigned".

```
let a;
console.log(a); // undefined
function test() {}
console.log(test()); // undefined
```

`null`

- Represents intentional absence of any object value.
- Used to explicitly clear or reset a variable.

```
let user = null; // explicitly set to nothing
```

Type Comparison

```
typeof undefined; // "undefined"
typeof null; // "object" (legacy bug)
```

Equality Comparison

```
undefined == null; // true (loose equality)
undefined === null; // false (strict equality)
```

Edge Cases:

- Accessing undeclared variables throws a ReferenceError, not `undefined`.
- Use strict checks: `if (value === null)` for null, `typeof value === 'undefined'` for undefined.

Best Practice:

- Use `undefined` for uninitialized variables, and `null` when you want to intentionally clear a value.

Question 5: What is hoisting in JavaScript? Explain with examples.

Hoisting is JavaScript's behavior of moving declarations (not initializations) to the top of their scope before code execution. This applies to variables and function declarations.

Example: `var`

```
console.log(a); // undefined
var a = 10;
```

JavaScript interprets this as:

```
var a;
console.log(a); // undefined
a = 10;
```

`let` and `const`

- These are also hoisted, but not initialized. Accessing them before declaration results in a ReferenceError (temporal dead zone).

```
console.log(b); // ReferenceError
let b = 20;
```

Function Declarations

- Function declarations are fully hoisted, so you can call them before they are defined.

```
sayHi();
function sayHi() {
  console.log("Hello!");
}
```

Function Expressions

- Not hoisted like declarations.

```
greet(); // TypeError: greet is not a function
var greet = function () {
  console.log("Hi");
};
```

Diagram (Mermaid):

```
graph TD
    A[Code Execution Start] --> B[Hoist Declarations to Top]
    B --> C[Initialize var as undefined]
    B --> D[Throw ReferenceError for let/const if accessed early]
```

Edge Cases:

- Hoisting can create subtle bugs, especially with `var`.
- Always declare variables at the top of their scope for clarity.

Question 6: What is a JavaScript object? Show how to create one.

A JavaScript object is a collection of key-value pairs, where keys are strings (or Symbols) and values can be any data type, including other objects or functions. Objects are used to store related data and functionality.

Example:

```
const person = {
  name: "John",
  age: 30,
  greet: function () {
    console.log(`Hello, my name is ${this.name}`);
  },
};
person.greet(); // Output: Hello, my name is John
```

Properties of an Object in JavaScript:

- **Key-Value Pairs**: Objects consist of properties, which are defined as key-value pairs.
    - e.g., `const car = { make: 'Toyota', model: 'Camry' };`
    - here, `make` and `model` are keys, and `'Toyota'` and `'Camry'` are their respective values. we can access these properties using dot notation (`car.make`) or bracket notation (`car['model']`).
- **Methods**: Functions can be defined as properties of an object, allowing for behavior to be associated with the object.
    - e.g , `const person = { name: 'Alice', greet: function() { console.log('Hello'); } };`
    - here, `greet` is a method of the `person` object that can be called using `person.greet()`.
    - Methods can also be defined using arrow functions, e.g., `const person = { name: 'Alice', greet: () => console.log('Hello'); }`.
    -
- **Dynamic Nature**: Properties can be added, modified, or deleted at runtime.
- e.g., `car.year = 2020;` adds a new property `year` to the `car` object.
- **Prototype-Based Inheritance**: Objects can inherit properties and methods from other objects, allowing for code reuse and organization. e.g , `const animal = { sound: 'roar' }; const lion = Object.create(animal); console.log(lion.sound); // Output: roar` here , `lion` inherits the `sound` property from the `animal` object. but it does not have its own `sound` property. this allows for a hierarchical structure where objects can share functionality. in this case, `lion` can access properties and methods defined in `animal`, promoting code reuse and organization.
- **Nested Objects**: Objects can contain other objects, allowing for complex data structures.
- e.g., `const company = { name: 'TechCorp', address: { city: 'New York', zip: '10001' } };`

    - here, the `address` property is itself an object with its own properties (`city` and `zip`).
    - this allows for organizing related data together, making it easier to manage and access complex information.
    -

- **Accessing Properties**: Properties can be accessed using dot notation or bracket notation.
- e.g., `console.log(car.make); // Output: Toyota` or `console.log(car['model']); // Output: Camry` here, we access the `make` and `model` properties of the `car` object using both dot notation and bracket notation. for example, `car.make` retrieves the value of the `make` property, while `car['model']` retrieves the value of the `model` property. in case of nested objects, we can access properties using dot notation as well, e.g., `console.log(company.address.city); // Output: New York`. or we can use bracket notation for dynamic property access, e.g., `const prop = 'zip'; console.log(company.address[prop]); // Output: 10001`. or we can mix both notations, e.g., `console.log(company['address']['city']); // Output: New York`.
-
- **JSON Representation**: Objects can be represented in JSON format, which is a text-based format for data interchange.
- e.g., `const jsonString = JSON.stringify(person);`
- here, the `person` object is converted to a JSON string representation.
- this allows for easy data exchange between systems, as JSON is widely used in APIs and web services.
- we can also parse a JSON string back into an object using `JSON.parse(jsonString)`, e.g., `const parsedPerson = JSON.parse(jsonString); console.log(parsedPerson.name); // Output: John`.
- this allows for seamless communication between different programming languages and platforms, as JSON is language-agnostic and can be easily parsed and generated in various environments.
- for example, we can send a JSON object from a server to a client, and the client can parse it to access the data. this makes JSON a popular choice for data interchange in web applications and APIs.
- a real use case for JSON representation is when sending data over the network in web applications. for example, when making an API request, we can send a JSON object containing user information, such as name and email, to the server. the server can then process this data and respond with a JSON object containing the requested information, such as user details or status messages. this allows for efficient communication between the client and server, enabling dynamic and interactive web applications.
- e.g. `const jsonString = JSON.stringify({ name: 'Alice', age: 25 });` this will convert the object into a JSON string representation, which can be sent over the network or stored in a file. we can then parse this JSON string back into an object using `JSON.parse(jsonString)`, allowing us to work with the data in a structured manner.
- for example in URL parameters, we can use JSON to encode complex data structures. this allows us to pass objects with nested properties as query parameters in a URL, making it easier to transmit structured data between the client and server. -- for example a search query can be represented as a

JSON object, such as `{ "query": "JavaScript", "filters": { "category": "programming" } }`. this JSON object can then be serialized into a string and appended to the URL as a query parameter, allowing the server to parse it and perform the search accordingly. like this: `const searchQuery = JSON.stringify({ query: 'JavaScript', filters: { category: 'programming' } }); const url =` https://example.com/search?query=${encodeURIComponent(searchQuery)}``

- **Prototypes**: JavaScript objects can inherit properties and methods from other objects through the prototype chain, allowing for shared functionality and behavior.
- e.g., `const animal = { sound: 'roar' }; const lion = Object.create(animal); console.log(lion.sound); // Output: roar`
- here, `lion` inherits the `sound` property from the `animal` object, demonstrating how objects can share properties and methods through inheritance.
- this allows for a hierarchical structure where objects can share functionality. in this case, `lion` can access properties and methods defined in `animal`, promoting code reuse and organization.
- **Immutability**: While object properties can be changed, the reference to the object itself can be made immutable using `Object.freeze()`.
  - e.g., `const obj = Object.freeze({ name: 'Alice' }); obj.name = 'Bob'; // This will not change the name property`
  - this prevents any modifications to the object, ensuring that its properties remain constant throughout its lifetime.
  - this is useful when you want to ensure that an object remains unchanged, such as when passing configuration objects or constants in your code.
- **Destructuring**: JavaScript allows for destructuring of objects, enabling easy extraction of properties into variables.
- e.g., `const { name, age } = person; console.log(name, age); // Output: John 30`
- this allows for concise and readable code when working with objects, as you can extract multiple properties in a single line.
- destructuring can also be used in function parameters, allowing you to pass an object and extract its properties directly in the function signature.
- for example, `function greet({ name }) { console.log(`Hello, ${name}`); }` allows you to pass an object with a `name` property and access it directly in the function body.
- **Spread Operator**: The spread operator (`...`) can be used to create shallow copies of objects or merge multiple objects into one.
- **Object Methods**: JavaScript provides built-in methods like `Object.keys()`, `Object.values()`, and `Object.entries()` to work with objects effectively.
- **Symbol Properties**: Symbols can be used as keys in objects, providing a way to create unique property identifiers that do not conflict with other properties.
- **Computed Property Names**: Objects can have properties defined with dynamic keys using square brackets, allowing for more flexible object creation.
- **Prototype Chain**: Objects can inherit properties and methods from their prototype, allowing for shared functionality across instances.
- **Object Literals**: Objects can be created using object literals, which provide a concise syntax for defining objects with properties and methods.

7. Explain JavaScript objects and provide an example demonstrating prototype-based inheritance.

A JavaScript object is a collection of key-value pairs, where keys are strings (or Symbols) and values can be any data type, including other objects or functions. Objects are fundamental to JavaScript and are used to model real-world entities, store data, and encapsulate behavior.

Prototype-Based Inheritance: JavaScript uses prototype-based inheritance, meaning objects can inherit properties and methods from other objects. Every object has an internal link to another object called its prototype. When you try to access a property that does not exist on the object itself, JavaScript looks up the prototype chain until it finds it or reaches the end.

Example:

```
const animal = {
  eats: true,
  walk() {
    console.log("Animal walks");
  },
};

const dog = Object.create(animal); // dog inherits from animal

dog.barks = true;
dog.walk(); // Output: Animal walks
console.log(dog.eats); // true (inherited)
console.log(dog.barks); // true (own property)
```

- Here, `dog` is created with `animal` as its prototype. It inherits the `eats` property and `walk` method from `animal`.
- You can check inheritance using `isPrototypeOf`:

```
console.log(animal.isPrototypeOf(dog)); // true
```

Constructor Functions and Prototypes: You can also use constructor functions to create objects with shared prototypes:

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function () {
  console.log(`Hello, my name is ${this.name}`);
};
const alice = new Person("Alice");
alice.greet(); // Output: Hello, my name is Alice
```

Edge Cases:

- Modifying the prototype affects all objects inheriting from it.

- You can override inherited properties by defining them on the object itself.

Question 8: What is the purpose of using AJAX in web applications?

The purpose of using AJAX in web applications includes:

1. **Asynchronous Communication:** AJAX allows web applications to communicate with the server asynchronously, meaning that data can be sent and received in the background without interfering with the user's interaction with the page.
2. **Partial Page Updates:** AJAX enables partial page updates, where only a specific portion of the web page is updated with new data, rather than reloading the entire page. This results in a smoother and more responsive user experience.
3. **Improved Performance:** By loading data in the background and updating only the necessary parts of the page, AJAX can significantly improve the performance and speed of web applications.
4. **Dynamic Content Loading:** AJAX allows for dynamic loading of content, such as fetching data from the server based on user actions (e.g., clicking a button, submitting a form) and updating the page with the new content.
5. **Enhanced User Experience:** With AJAX, web applications can provide a more interactive and seamless user experience, similar to that of desktop applications.

Question 9: What command is used to set up a new React application using Create React App?

The command to set up a new React application using Create React App is:

```
npx create-react-app my-app
```

- `npx` is a package runner tool that comes with Node.js. It allows you to run commands from npm packages without installing them globally.
- `create-react-app` is a command-line tool that sets up a new React project with a pre-configured development environment.
- `my-app` is the name of the directory where the new React app will be created. You can replace it with your desired app name.

After running this command, you can navigate to the project directory and start the development server:

```
cd my-app
npm start
```

This will start the development server, and you can view your new React app in the browser at `http://localhost:3000`.

Section 1: JavaScript Fundamentals

1. Explain front-end development. How does it differ from back-end development?
2. What do you understand by operators in JavaScript? Provide two examples.
3. What are control structures in programming? Explain with a JavaScript example.
4. Describe different types of variables and data types available in JavaScript.
5. How do `var`, `let`, and `const` behave with respect to scope and value retention? Illustrate with examples.
6. What is a JavaScript object? Show how to create one. A JavaScript object is a collection of key-value pairs, where keys are strings (or Symbols) and values can be any data type, including other objects or functions. Objects are used to store related data and functionality. Example:

```
const person = {
  name: "John",
  age: 30,
  greet: function () {
    console.log(`Hello, my name is ${this.name}`);
  },
};
person.greet(); // Output: Hello, my name is John
```

Properties of an Object in JavaScript:

- **Key-Value Pairs**: Objects consist of properties, which are defined as key-value pairs.
  - e.g., `const car = { make: 'Toyota', model: 'Camry' };`
  - here, `make` and `model` are keys, and `'Toyota'` and `'Camry'` are their respective values. we can access these properties using dot notation (`car.make`) or bracket notation (`car['model']`).
- **Methods**: Functions can be defined as properties of an object, allowing for behavior to be associated with the object.
  - e.g , `const person = { name: 'Alice', greet: function() { console.log('Hello'); } };`
  - here, `greet` is a method of the `person` object that can be called using `person.greet()`.
  - Methods can also be defined using arrow functions, e.g., `const person = { name: 'Alice', greet: () => console.log('Hello'); }`.
  -
- **Dynamic Nature**: Properties can be added, modified, or deleted at runtime.
- e.g., `car.year = 2020;` adds a new property `year` to the `car` object.
- **Prototype-Based Inheritance**: Objects can inherit properties and methods from other objects, allowing for code reuse and organization. e.g , `const animal = { sound: 'roar' }; const`

`lion = Object.create(animal); console.log(lion.sound); // Output: roar` here , `lion` inherits the `sound` property from the `animal` object. but it does not have its own `sound` property. this allows for a hierarchical structure where objects can share functionality. in this case, `lion` can access properties and methods defined in `animal`, promoting code reuse and organization.

- **Nested Objects**: Objects can contain other objects, allowing for complex data structures.
- e.g., `const company = { name: 'TechCorp', address: { city: 'New York', zip: '10001' } };`
  - here, the `address` property is itself an object with its own properties (`city` and `zip`).
  - this allows for organizing related data together, making it easier to manage and access complex information.
  -
- **Accessing Properties**: Properties can be accessed using dot notation or bracket notation.
- e.g., `console.log(car.make); // Output: Toyota` or `console.log(car['model']); // Output: Camry` here, we access the `make` and `model` properties of the `car` object using both dot notation and bracket notation. for example, `car.make` retrieves the value of the `make` property, while `car['model']` retrieves the value of the `model` property. in case of nested objects, we can access properties using dot notation as well, e.g., `console.log(company.address.city); // Output: New York`. or we can use bracket notation for dynamic property access, e.g., `const prop = 'zip'; console.log(company.address[prop]); // Output: 10001`. or we can mix both notations, e.g., `console.log(company['address']['city']); // Output: New York`.

- **JSON Representation**: Objects can be represented in JSON format, which is a text-based format for data interchange.
- e.g., `const jsonString = JSON.stringify(person);`
- here, the `person` object is converted to a JSON string representation.
- this allows for easy data exchange between systems, as JSON is widely used in APIs and web services.
- we can also parse a JSON string back into an object using `JSON.parse(jsonString)`, e.g., `const parsedPerson = JSON.parse(jsonString); console.log(parsedPerson.name); // Output: John`.
- this allows for seamless communication between different programming languages and platforms, as JSON is language-agnostic and can be easily parsed and generated in various environments.
- for example, we can send a JSON object from a server to a client, and the client can parse it to access the data. this makes JSON a popular choice for data interchange in web applications and APIs.
- a real use case for JSON representation is when sending data over the network in web applications. for example, when making an API request, we can send a JSON object containing user information, such as name and email, to the server. the server can then process this data and respond with a JSON object containing the requested information, such as user details or status messages. this allows for efficient communication between the client and server, enabling dynamic and interactive web applications.
- e.g. `const jsonString = JSON.stringify({ name: 'Alice', age: 25 });` this will convert the object into a JSON string representation, which can be sent over the network or stored in a file. we can

then parse this JSON string back into an object using `JSON.parse(jsonString)`, allowing us to work with the data in a structured manner.

- for example in URL parameters, we can use JSON to encode complex data structures. this allows us to pass objects with nested properties as query parameters in a URL, making it easier to transmit structured data between the client and server. -- for example a search query can be represented as a JSON object, such as `{ "query": "JavaScript", "filters": { "category": "programming" } }`. this JSON object can then be serialized into a string and appended to the URL as a query parameter, allowing the server to parse it and perform the search accordingly. like this: `const searchQuery = JSON.stringify({ query: 'JavaScript', filters: { category: 'programming' } }); const url =` https://example.com/search?query=${encodeURIComponent(searchQuery)}``
    - **Prototypes**: JavaScript objects can inherit properties and methods from other objects through the prototype chain, allowing for shared functionality and behavior.
    - e.g., `const animal = { sound: 'roar' }; const lion = Object.create(animal); console.log(lion.sound); // Output: roar`
    - here, `lion` inherits the `sound` property from the `animal` object, demonstrating how objects can share properties and methods through inheritance.
    - this allows for a hierarchical structure where objects can share functionality. in this case, `lion` can access properties and methods defined in `animal`, promoting code reuse and organization.
    - **Immutability**: While object properties can be changed, the reference to the object itself can be made immutable using `Object.freeze()`.
        - e.g., `const obj = Object.freeze({ name: 'Alice' }); obj.name = 'Bob'; // This will not change the name property`
        - this prevents any modifications to the object, ensuring that its properties remain constant throughout its lifetime.
        - this is useful when you want to ensure that an object remains unchanged, such as when passing configuration objects or constants in your code.
    - **Destructuring**: JavaScript allows for destructuring of objects, enabling easy extraction of properties into variables.
    - e.g., `const { name, age } = person; console.log(name, age); // Output: John 30`
    - this allows for concise and readable code when working with objects, as you can extract multiple properties in a single line.
    - destructuring can also be used in function parameters, allowing you to pass an object and extract its properties directly in the function signature.
    - for example, `function greet({ name }) { console.log(`Hello, ${name}`); }` allows you to pass an object with a `name` property and access it directly in the function body.
    - **Spread Operator**: The spread operator (`...`) can be used to create shallow copies of objects or merge multiple objects into one.
    - **Object Methods**: JavaScript provides built-in methods like `Object.keys()`, `Object.values()`, and `Object.entries()` to work with objects effectively.
    - **Symbol Properties**: Symbols can be used as keys in objects, providing a way to create unique property identifiers that do not conflict with other properties.
    - **Computed Property Names**: Objects can have properties defined with dynamic keys using square brackets, allowing for more flexible object creation.
    - **Prototype Chain**: Objects can inherit properties and methods from their prototype, allowing for shared functionality across instances.

- **Object Literals**: Objects can be created using object literals, which provide a concise syntax for defining objects with properties and methods.

7. Explain JavaScript objects and provide an example demonstrating prototype-based inheritance.

A JavaScript object is a collection of key-value pairs, where keys are strings (or Symbols) and values can be any data type, including other objects or functions. Objects are fundamental to JavaScript and are used to model real-world entities, store data, and encapsulate behavior.

Prototype-Based Inheritance: JavaScript uses prototype-based inheritance, meaning objects can inherit properties and methods from other objects. Every object has an internal link to another object called its prototype. When you try to access a property that does not exist on the object itself, JavaScript looks up the prototype chain until it finds it or reaches the end.

Example:

```
const animal = {
  eats: true,
  walk() {
    console.log("Animal walks");
  },
};

const dog = Object.create(animal); // dog inherits from animal

dog.barks = true;
dog.walk(); // Output: Animal walks
console.log(dog.eats); // true (inherited)
console.log(dog.barks); // true (own property)
```

- Here, `dog` is created with `animal` as its prototype. It inherits the `eats` property and `walk` method from `animal`.
- You can check inheritance using `isPrototypeOf`:

```
console.log(animal.isPrototypeOf(dog)); // true
```

Constructor Functions and Prototypes: You can also use constructor functions to create objects with shared prototypes:

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function () {
  console.log(`Hello, my name is ${this.name}`);
};
const alice = new Person("Alice");
alice.greet(); // Output: Hello, my name is Alice
```

Edge Cases:

- Modifying the prototype affects all objects inheriting from it.
- You can override inherited properties by defining them on the object itself.

Question 8: What is the purpose of using AJAX in web applications?

The purpose of using AJAX in web applications includes:

1. **Asynchronous Communication:** AJAX allows web applications to communicate with the server asynchronously, meaning that data can be sent and received in the background without interfering with the user's interaction with the page.
2. **Partial Page Updates:** AJAX enables partial page updates, where only a specific portion of the web page is updated with new data, rather than reloading the entire page. This results in a smoother and more responsive user experience.
3. **Improved Performance:** By loading data in the background and updating only the necessary parts of the page, AJAX can significantly improve the performance and speed of web applications.
4. **Dynamic Content Loading:** AJAX allows for dynamic loading of content, such as fetching data from the server based on user actions (e.g., clicking a button, submitting a form) and updating the page with the new content.
5. **Enhanced User Experience:** With AJAX, web applications can provide a more interactive and seamless user experience, similar to that of desktop applications.

Question 9: What command is used to set up a new React application using Create React App?

The command to set up a new React application using Create React App is:

```
npx create-react-app my-app
```

- `npx` is a package runner tool that comes with Node.js. It allows you to run commands from npm packages without installing them globally.
- `create-react-app` is a command-line tool that sets up a new React project with a pre-configured development environment.
- `my-app` is the name of the directory where the new React app will be created. You can replace it with your desired app name.

After running this command, you can navigate to the project directory and start the development server:

```
cd my-app
npm start
```

This will start the development server, and you can view your new React app in the browser at `http://localhost:3000`.

Section 1: JavaScript Fundamentals

1. Explain front-end development. How does it differ from back-end development?
2. What do you understand by operators in JavaScript? Provide two examples.
3. What are control structures in programming? Explain with a JavaScript example.
4. Describe different types of variables and data types available in JavaScript.
5. How do `var`, `let`, and `const` behave with respect to scope and value retention? Illustrate with examples.
6. What is a JavaScript object? Show how to create one. A JavaScript object is a collection of key-value pairs, where keys are strings (or Symbols) and values can be any data type, including other objects or functions. Objects are used to store related data and functionality. Example:

```
const person = {
  name: "John",
  age: 30,
  greet: function () {
    console.log(`Hello, my name is ${this.name}`);
  },
};
person.greet(); // Output: Hello, my name is John
```

Properties of an Object in JavaScript:

- **Key-Value Pairs**: Objects consist of properties, which are defined as key-value pairs.
  - e.g., `const car = { make: 'Toyota', model: 'Camry' };`
  - here, `make` and `model` are keys, and `'Toyota'` and `'Camry'` are their respective values. we can access these properties using dot notation (`car.make`) or bracket notation (`car['model']`).
- **Methods**: Functions can be defined as properties of an object, allowing for behavior to be associated with the object.
  - e.g , `const person = { name: 'Alice', greet: function() { console.log('Hello'); } };`
  - here, `greet` is a method of the `person` object that can be called using `person.greet()`.
  - Methods can also be defined using arrow functions, e.g., `const person = { name: 'Alice', greet: () => console.log('Hello'); }`.
  -
- **Dynamic Nature**: Properties can be added, modified, or deleted at runtime.
- e.g., `car.year = 2020;` adds a new property `year` to the `car` object.
- **Prototype-Based Inheritance**: Objects can inherit properties and methods from other objects, allowing for code reuse and organization. e.g , `const animal = { sound: 'roar' }; const`

`lion = Object.create(animal); console.log(lion.sound); // Output: roar` here , `lion` inherits the `sound` property from the `animal` object. but it does not have its own `sound` property. this allows for a hierarchical structure where objects can share functionality. in this case, `lion` can access properties and methods defined in `animal`, promoting code reuse and organization.

- **Nested Objects**: Objects can contain other objects, allowing for complex data structures.
- e.g., `const company = { name: 'TechCorp', address: { city: 'New York', zip: '10001' } };`
  - here, the `address` property is itself an object with its own properties (`city` and `zip`).
  - this allows for organizing related data together, making it easier to manage and access complex information.
  -
- **Accessing Properties**: Properties can be accessed using dot notation or bracket notation.
- e.g., `console.log(car.make); // Output: Toyota` or `console.log(car['model']); // Output: Camry` here, we access the `make` and `model` properties of the `car` object using both dot notation and bracket notation. for example, `car.make` retrieves the value of the `make` property, while `car['model']` retrieves the value of the `model` property. in case of nested objects, we can access properties using dot notation as well, e.g., `console.log(company.address.city); // Output: New York`. or we can use bracket notation for dynamic property access, e.g., `const prop = 'zip'; console.log(company.address[prop]); // Output: 10001`. or we can mix both notations, e.g., `console.log(company['address']['city']); // Output: New York`.

- **JSON Representation**: Objects can be represented in JSON format, which is a text-based format for data interchange.
- e.g., `const jsonString = JSON.stringify(person);`
- here, the `person` object is converted to a JSON string representation.
- this allows for easy data exchange between systems, as JSON is widely used in APIs and web services.
- we can also parse a JSON string back into an object using `JSON.parse(jsonString)`, e.g., `const parsedPerson = JSON.parse(jsonString); console.log(parsedPerson.name); // Output: John`.
- this allows for seamless communication between different programming languages and platforms, as JSON is language-agnostic and can be easily parsed and generated in various environments.
- for example, we can send a JSON object from a server to a client, and the client can parse it to access the data. this makes JSON a popular choice for data interchange in web applications and APIs.
- a real use case for JSON representation is when sending data over the network in web applications. for example, when making an API request, we can send a JSON object containing user information, such as name and email, to the server. the server can then process this data and respond with a JSON object containing the requested information, such as user details or status messages. this allows for efficient communication between the client and server, enabling dynamic and interactive web applications.
- e.g. `const jsonString = JSON.stringify({ name: 'Alice', age: 25 });` this will convert the object into a JSON string representation, which can be sent over the network or stored in a file. we can

then parse this JSON string back into an object using `JSON.parse(jsonString)`, allowing us to work with the data in a structured manner.

- for example in URL parameters, we can use JSON to encode complex data structures. this allows us to pass objects with nested properties as query parameters in a URL, making it easier to transmit structured data between the client and server. -- for example a search query can be represented as a JSON object, such as `{ "query": "JavaScript", "filters": { "category": "programming" } }`. this JSON object can then be serialized into a string and appended to the URL as a query parameter, allowing the server to parse it and perform the search accordingly. like this: `const searchQuery = JSON.stringify({ query: 'JavaScript', filters: { category: 'programming' } }); const url =` https://example.com/search?query=${encodeURIComponent(searchQuery)}``
    - **Prototypes**: JavaScript objects can inherit properties and methods from other objects through the prototype chain, allowing for shared functionality and behavior.
    - e.g., `const animal = { sound: 'roar' }; const lion = Object.create(animal); console.log(lion.sound); // Output: roar`
    - here, `lion` inherits the `sound` property from the `animal` object, demonstrating how objects can share properties and methods through inheritance.
    - this allows for a hierarchical structure where objects can share functionality. in this case, `lion` can access properties and methods defined in `animal`, promoting code reuse and organization.
    - **Immutability**: While object properties can be changed, the reference to the object itself can be made immutable using `Object.freeze()`.
        - e.g., `const obj = Object.freeze({ name: 'Alice' }); obj.name = 'Bob'; // This will not change the name property`
        - this prevents any modifications to the object, ensuring that its properties remain constant throughout its lifetime.
        - this is useful when you want to ensure that an object remains unchanged, such as when passing configuration objects or constants in your code.
    - **Destructuring**: JavaScript allows for destructuring of objects, enabling easy extraction of properties into variables.
    - e.g., `const { name, age } = person; console.log(name, age); // Output: John 30`
    - this allows for concise and readable code when working with objects, as you can extract multiple properties in a single line.
    - destructuring can also be used in function parameters, allowing you to pass an object and extract its properties directly in the function signature.
    - for example, `function greet({ name }) { console.log(`Hello, ${name}`); }` allows you to pass an object with a `name` property and access it directly in the function body.
    - **Spread Operator**: The spread operator (`...`) can be used to create shallow copies of objects or merge multiple objects into one.
    - **Object Methods**: JavaScript provides built-in methods like `Object.keys()`, `Object.values()`, and `Object.entries()` to work with objects effectively.
    - **Symbol Properties**: Symbols can be used as keys in objects, providing a way to create unique property identifiers that do not conflict with other properties.
    - **Computed Property Names**: Objects can have properties defined with dynamic keys using square brackets, allowing for more flexible object creation.
    - **Prototype Chain**: Objects can inherit properties and methods from their prototype, allowing for shared functionality across instances.

- **Object Literals**: Objects can be created using object literals, which provide a concise syntax for defining objects with properties and methods.

7. Explain JavaScript objects and provide an example demonstrating prototype-based inheritance.

A JavaScript object is a collection of key-value pairs, where keys are strings (or Symbols) and values can be any data type, including other objects or functions. Objects are fundamental to JavaScript and are used to model real-world entities, store data, and encapsulate behavior.

Prototype-Based Inheritance: JavaScript uses prototype-based inheritance, meaning objects can inherit properties and methods from other objects. Every object has an internal link to another object called its prototype. When you try to access a property that does not exist on the object itself, JavaScript looks up the prototype chain until it finds it or reaches the end.

Example:

```
const animal = {
  eats: true,
  walk() {
    console.log("Animal walks");
  },
};

const dog = Object.create(animal); // dog inherits from animal

dog.barks = true;
dog.walk(); // Output: Animal walks
console.log(dog.eats); // true (inherited)
console.log(dog.barks); // true (own property)
```

- Here, `dog` is created with `animal` as its prototype. It inherits the `eats` property and `walk` method from `animal`.
- You can check inheritance using `isPrototypeOf`:

```
console.log(animal.isPrototypeOf(dog)); // true
```

Constructor Functions and Prototypes: You can also use constructor functions to create objects with shared prototypes:

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function () {
  console.log(`Hello, my name is ${this.name}`);
};
const alice = new Person("Alice");
alice.greet(); // Output: Hello, my name is Alice
```

Edge Cases:

- Modifying the prototype affects all objects inheriting from it.
- You can override inherited properties by defining them on the object itself.

Question 8: What is the purpose of using AJAX in web applications?

The purpose of using AJAX in web applications includes:

1. **Asynchronous Communication:** AJAX allows web applications to communicate with the server asynchronously, meaning that data can be sent and received in the background without interfering with the user's interaction with the page.
2. **Partial Page Updates:** AJAX enables partial page updates, where only a specific portion of the web page is updated with new data, rather than reloading the entire page. This results in a smoother and more responsive user experience.
3. **Improved Performance:** By loading data in the background and updating only the necessary parts of the page, AJAX can significantly improve the performance and speed of web applications.
4. **Dynamic Content Loading:** AJAX allows for dynamic loading of content, such as fetching data from the server based on user actions (e.g., clicking a button, submitting a form) and updating the page with the new content.
5. **Enhanced User Experience:** With AJAX, web applications can provide a more interactive and seamless user experience, similar to that of desktop applications.

Question 9: What command is used to set up a new React application using Create React App?

The command to set up a new React application using Create React App is:

```
npx create-react-app my-app
```

- `npx` is a package runner tool that comes with Node.js. It allows you to run commands from npm packages without installing them globally.
- `create-react-app` is a command-line tool that sets up a new React project with a pre-configured development environment.
- `my-app` is the name of the directory where the new React app will be created. You can replace it with your desired app name.

After running this command, you can navigate to the project directory and start the development server:

```
cd my-app
npm start
```

This will start the development server, and you can view your new React app in the browser at `http://localhost:3000`.

Section 1: JavaScript Fundamentals

1. Explain front-end development. How does it differ from back-end development?
2. What do you understand by operators in JavaScript? Provide two examples.
3. What are control structures in programming? Explain with a JavaScript example.
4. Describe different types of variables and data types available in JavaScript.
5. How do `var`, `let`, and `const` behave with respect to scope and value retention? Illustrate with examples.
6. What is a JavaScript object? Show how to create one. A JavaScript object is a collection of key-value pairs, where keys are strings (or Symbols) and values can be any data type, including other objects or functions. Objects are used to store related data and functionality. Example:

```
const person = {
  name: "John",
  age: 30,
  greet: function () {
    console.log(`Hello, my name is ${this.name}`);
  },
};
person.greet(); // Output: Hello, my name is John
```

Properties of an Object in JavaScript:

- **Key-Value Pairs**: Objects consist of properties, which are defined as key-value pairs.
  - e.g., `const car = { make: 'Toyota', model: 'Camry' };`
  - here, `make` and `model` are keys, and `'Toyota'` and `'Camry'` are their respective values. we can access these properties using dot notation (`car.make`) or bracket notation (`car['model']`).
- **Methods**: Functions can be defined as properties of an object, allowing for behavior to be associated with the object.
  - e.g , `const person = { name: 'Alice', greet: function() { console.log('Hello'); } };`
  - here, `greet` is a method of the `person` object that can be called using `person.greet()`.
  - Methods can also be defined using arrow functions, e.g., `const person = { name: 'Alice', greet: () => console.log('Hello'); }`.
  -
- **Dynamic Nature**: Properties can be added, modified, or deleted at runtime.
- e.g., `car.year = 2020;` adds a new property `year` to the `car` object.
- **Prototype-Based Inheritance**: Objects can inherit properties and methods from other objects, allowing for code reuse and organization. e.g , `const animal = { sound: 'roar' }; const`

`lion = Object.create(animal); console.log(lion.sound); // Output: roar` here , `lion` inherits the `sound` property from the `animal` object. but it does not have its own `sound` property. this allows for a hierarchical structure where objects can share functionality. in this case, `lion` can access properties and methods defined in `animal`, promoting code reuse and organization.

- **Nested Objects**: Objects can contain other objects, allowing for complex data structures.
- e.g., `const company = { name: 'TechCorp', address: { city: 'New York', zip: '10001' } };`
  - here, the `address` property is itself an object with its own properties (`city` and `zip`).
  - this allows for organizing related data together, making it easier to manage and access complex information.
  -
- **Accessing Properties**: Properties can be accessed using dot notation or bracket notation.
- e.g., `console.log(car.make); // Output: Toyota` or `console.log(car['model']); // Output: Camry` here, we access the `make` and `model` properties of the `car` object using both dot notation and bracket notation. for example, `car.make` retrieves the value of the `make` property, while `car['model']` retrieves the value of the `model` property. in case of nested objects, we can access properties using dot notation as well, e.g., `console.log(company.address.city); // Output: New York`. or we can use bracket notation for dynamic property access, e.g., `const prop = 'zip'; console.log(company.address[prop]); // Output: 10001`. or we can mix both notations, e.g., `console.log(company['address']['city']); // Output: New York`.

- **JSON Representation**: Objects can be represented in JSON format, which is a text-based format for data interchange.
- e.g., `const jsonString = JSON.stringify(person);`
- here, the `person` object is converted to a JSON string representation.
- this allows for easy data exchange between systems, as JSON is widely used in APIs and web services.
- we can also parse a JSON string back into an object using `JSON.parse(jsonString)`, e.g., `const parsedPerson = JSON.parse(jsonString); console.log(parsedPerson.name); // Output: John`.
- this allows for seamless communication between different programming languages and platforms, as JSON is language-agnostic and can be easily parsed and generated in various environments.
- for example, we can send a JSON object from a server to a client, and the client can parse it to access the data. this makes JSON a popular choice for data interchange in web applications and APIs.
- a real use case for JSON representation is when sending data over the network in web applications. for example, when making an API request, we can send a JSON object containing user information, such as name and email, to the server. the server can then process this data and respond with a JSON object containing the requested information, such as user details or status messages. this allows for efficient communication between the client and server, enabling dynamic and interactive web applications.
- e.g. `const jsonString = JSON.stringify({ name: 'Alice', age: 25 });` this will convert the object into a JSON string representation, which can be sent over the network or stored in a file. we can

then parse this JSON string back into an object using `JSON.parse(jsonString)`, allowing us to work with the data in a structured manner.

- for example in URL parameters, we can use JSON to encode complex data structures. this allows us to pass objects with nested properties as query parameters in a URL, making it easier to transmit structured data between the client and server. -- for example a search query can be represented as a JSON object, such as `{ "query": "JavaScript", "filters": { "category": "programming" } }`. this JSON object can then be serialized into a string and appended to the URL as a query parameter, allowing the server to parse it and perform the search accordingly. like this: `const searchQuery = JSON.stringify({ query: 'JavaScript', filters: { category: 'programming' } }); const url =` https://example.com/search?query=${encodeURIComponent(searchQuery)}``
    - **Prototypes**: JavaScript objects can inherit properties and methods from other objects through the prototype chain, allowing for shared functionality and behavior.
    - e.g., `const animal = { sound: 'roar' }; const lion = Object.create(animal); console.log(lion.sound); // Output: roar`
    - here, `lion` inherits the `sound` property from the `animal` object, demonstrating how objects can share properties and methods through inheritance.
    - this allows for a hierarchical structure where objects can share functionality. in this case, `lion` can access properties and methods defined in `animal`, promoting code reuse and organization.
    - **Immutability**: While object properties can be changed, the reference to the object itself can be made immutable using `Object.freeze()`.
        - e.g., `const obj = Object.freeze({ name: 'Alice' }); obj.name = 'Bob'; // This will not change the name property`
        - this prevents any modifications to the object, ensuring that its properties remain constant throughout its lifetime.
        - this is useful when you want to ensure that an object remains unchanged, such as when passing configuration objects or constants in your code.
    - **Destructuring**: JavaScript allows for destructuring of objects, enabling easy extraction of properties into variables.
    - e.g., `const { name, age } = person; console.log(name, age); // Output: John 30`
    - this allows for concise and readable code when working with objects, as you can extract multiple properties in a single line.
    - destructuring can also be used in function parameters, allowing you to pass an object and extract its properties directly in the function signature.
    - for example, `function greet({ name }) { console.log(`Hello, ${name}`); }` allows you to pass an object with a `name` property and access it directly in the function body.
    - **Spread Operator**: The spread operator (`...`) can be used to create shallow copies of objects or merge multiple objects into one.
    - **Object Methods**: JavaScript provides built-in methods like `Object.keys()`, `Object.values()`, and `Object.entries()` to work with objects effectively.
    - **Symbol Properties**: Symbols can be used as keys in objects, providing a way to create unique property identifiers that do not conflict with other properties.
    - **Computed Property Names**: Objects can have properties defined with dynamic keys using square brackets, allowing for more flexible object creation.
    - **Prototype Chain**: Objects can inherit properties and methods from their prototype, allowing for shared functionality across instances.

- **Object Literals**: Objects can be created using object literals, which provide a concise syntax for defining objects with properties and methods.

7. Explain JavaScript objects and provide an example demonstrating prototype-based inheritance.

A JavaScript object is a collection of key-value pairs, where keys are strings (or Symbols) and values can be any data type, including other objects or functions. Objects are fundamental to JavaScript and are used to model real-world entities, store data, and encapsulate behavior.

Prototype-Based Inheritance: JavaScript uses prototype-based inheritance, meaning objects can inherit properties and methods from other objects. Every object has an internal link to another object called its prototype. When you try to access a property that does not exist on the object itself, JavaScript looks up the prototype chain until it finds it or reaches the end.

Example:

```
const animal = {
  eats: true,
  walk() {
    console.log("Animal walks");
  },
};

const dog = Object.create(animal); // dog inherits from animal

dog.barks = true;
dog.walk(); // Output: Animal walks
console.log(dog.eats); // true (inherited)
console.log(dog.barks); // true (own property)
```

- Here, `dog` is created with `animal` as its prototype. It inherits the `eats` property and `walk` method from `animal`.
- You can check inheritance using `isPrototypeOf`:

```
console.log(animal.isPrototypeOf(dog)); // true
```

Constructor Functions and Prototypes: You can also use constructor functions to create objects with shared prototypes:

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function () {
  console.log(`Hello, my name is ${this.name}`);
};
const alice = new Person("Alice");
alice.greet(); // Output: Hello, my name is Alice
```

Edge Cases:

- Modifying the prototype affects all objects inheriting from it.
- You can override inherited properties by defining them on the object itself.

Question 8: What is the purpose of using AJAX in web applications?

The purpose of using AJAX in web applications includes:

1. **Asynchronous Communication:** AJAX allows web applications to communicate with the server asynchronously, meaning that data can be sent and received in the background without interfering with the user's interaction with the page.
2. **Partial Page Updates:** AJAX enables partial page updates, where only a specific portion of the web page is updated with new data, rather than reloading the entire page. This results in a smoother and more responsive user experience.
3. **Improved Performance:** By loading data in the background and updating only the necessary parts of the page, AJAX can significantly improve the performance and speed of web applications.
4. **Dynamic Content Loading:** AJAX allows for dynamic loading of content, such as fetching data from the server based on user actions (e.g., clicking a button, submitting a form) and updating the page with the new content.
5. **Enhanced User Experience:** With AJAX, web applications can provide a more interactive and seamless user experience, similar to that of desktop applications.

Question 9: What command is used to set up a new React application using Create React App?

The command to set up a new React application using Create React App is:

```
npx create-react-app my-app
```

- `npx` is a package runner tool that comes with Node.js. It allows you to run commands from npm packages without installing them globally.
- `create-react-app` is a command-line tool that sets up a new React project with a pre-configured development environment.
- `my-app` is the name of the directory where the new React app will be created. You can replace it with your desired app name.

After running this command, you can navigate to the project directory and start the development server:

```
cd my-app
npm start
```

This will start the development server, and you can view your new React app in the browser at `http://localhost:3000`.

Section 1: JavaScript Fundamentals

1. Explain front-end development. How does it differ from back-end development?
2. What do you understand by operators in JavaScript? Provide two examples.
3. What are control structures in programming? Explain with a JavaScript example.
4. Describe different types of variables and data types available in JavaScript.
5. How do `var`, `let`, and `const` behave with respect to scope and value retention? Illustrate with examples.
6. What is a JavaScript object? Show how to create one. A JavaScript object is a collection of key-value pairs, where keys are strings (or Symbols) and values can be any data type, including other objects or functions. Objects are used to store related data and functionality. Example:

```
const person = {
  name: "John",
  age: 30,
  greet: function () {
    console.log(`Hello, my name is ${this.name}`);
  },
};
person.greet(); // Output: Hello, my name is John
```

Properties of an Object in JavaScript:

- **Key-Value Pairs**: Objects consist of properties, which are defined as key-value pairs.
  - e.g., `const car = { make: 'Toyota', model: 'Camry' };`
  - here, `make` and `model` are keys, and `'Toyota'` and `'Camry'` are their respective values. we can access these properties using dot notation (`car.make`) or bracket notation (`car['model']`).
- **Methods**: Functions can be defined as properties of an object, allowing for behavior to be associated with the object.
  - e.g , `const person = { name: 'Alice', greet: function() { console.log('Hello'); } };`
  - here, `greet` is a method of the `person` object that can be called using `person.greet()`.
  - Methods can also be defined using arrow functions, e.g., `const person = { name: 'Alice', greet: () => console.log('Hello'); }`.
  -
- **Dynamic Nature**: Properties can be added, modified, or deleted at runtime.
- e.g., `car.year = 2020;` adds a new property `year` to the `car` object.
- **Prototype-Based Inheritance**: Objects can inherit properties and methods from other objects, allowing for code reuse and organization. e.g , `const animal = { sound: 'roar' }; const`

`lion = Object.create(animal); console.log(lion.sound); // Output: roar` here , `lion` inherits the `sound` property from the `animal` object. but it does not have its own `sound` property. this allows for a hierarchical structure where objects can share functionality. in this case, `lion` can access properties and methods defined in `animal`, promoting code reuse and organization.

- **Nested Objects**: Objects can contain other objects, allowing for complex data structures.
- e.g., `const company = { name: 'TechCorp', address: { city: 'New York', zip: '10001' } };`
  - here, the `address` property is itself an object with its own properties (`city` and `zip`).
  - this allows for organizing related data together, making it easier to manage and access complex information.
  -
- **Accessing Properties**: Properties can be accessed using dot notation or bracket notation.
- e.g., `console.log(car.make); // Output: Toyota` or `console.log(car['model']); // Output: Camry` here, we access the `make` and `model` properties of the `car` object using both dot notation and bracket notation. for example, `car.make` retrieves the value of the `make` property, while `car['model']` retrieves the value of the `model` property. in case of nested objects, we can access properties using dot notation as well, e.g., `console.log(company.address.city); // Output: New York`. or we can use bracket notation for dynamic property access, e.g., `const prop = 'zip'; console.log(company.address[prop]); // Output: 10001`. or we can mix both notations, e.g., `console.log(company['address']['city']); // Output: New York`.

- **JSON Representation**: Objects can be represented in JSON format, which is a text-based format for data interchange.
- e.g., `const jsonString = JSON.stringify(person);`
- here, the `person` object is converted to a JSON string representation.
- this allows for easy data exchange between systems, as JSON is widely used in APIs and web services.
- we can also parse a JSON string back into an object using `JSON.parse(jsonString)`, e.g., `const parsedPerson = JSON.parse(jsonString); console.log(parsedPerson.name); // Output: John`.
- this allows for seamless communication between different programming languages and platforms, as JSON is language-agnostic and can be easily parsed and generated in various environments.
- for example, we can send a JSON object from a server to a client, and the client can parse it to access the data. this makes JSON a popular choice for data interchange in web applications and APIs.
- a real use case for JSON representation is when sending data over the network in web applications. for example, when making an API request, we can send a JSON object containing user information, such as name and email, to the server. the server can then process this data and respond with a JSON object containing the requested information, such as user details or status messages. this allows for efficient communication between the client and server, enabling dynamic and interactive web applications.
- e.g. `const jsonString = JSON.stringify({ name: 'Alice', age: 25 });` this will convert the object into a JSON string representation, which can be sent over the network or stored in a file. we can

then parse this JSON string back into an object using `JSON.parse(jsonString)`, allowing us to work with the data in a structured manner.

- for example in URL parameters, we can use JSON to encode complex data structures. this allows us to pass objects with nested properties as query parameters in a URL, making it easier to transmit structured data between the client and server. -- for example a search query can be represented as a JSON object, such as `{ "query": "JavaScript", "filters": { "category": "programming" } }`. this JSON object can then be serialized into a string and appended to the URL as a query parameter, allowing the server to parse it and perform the search accordingly. like this: `const searchQuery = JSON.stringify({ query: 'JavaScript', filters: { category: 'programming' } }); const url =` https://example.com/search?query=${encodeURIComponent(searchQuery)}``
    - **Prototypes**: JavaScript objects can inherit properties and methods from other objects through the prototype chain, allowing for shared functionality and behavior.
    - e.g., `const animal = { sound: 'roar' }; const lion = Object.create(animal); console.log(lion.sound); // Output: roar`
    - here, `lion` inherits the `sound` property from the `animal` object, demonstrating how objects can share properties and methods through inheritance.
    - this allows for a hierarchical structure where objects can share functionality. in this case, `lion` can access properties and methods defined in `animal`, promoting code reuse and organization.
    - **Immutability**: While object properties can be changed, the reference to the object itself can be made immutable using `Object.freeze()`.
        - e.g., `const obj = Object.freeze({ name: 'Alice' }); obj.name = 'Bob'; // This will not change the name property`
        - this prevents any modifications to the object, ensuring that its properties remain constant throughout its lifetime.
        - this is useful when you want to ensure that an object remains unchanged, such as when passing configuration objects or constants in your code.
    - **Destructuring**: JavaScript allows for destructuring of objects, enabling easy extraction of properties into variables.
    - e.g., `const { name, age } = person; console.log(name, age); // Output: John 30`
    - this allows for concise and readable code when working with objects, as you can extract multiple properties in a single line.
    - destructuring can also be used in function parameters, allowing you to pass an object and extract its properties directly in the function signature.
    - for example, `function greet({ name }) { console.log(`Hello, ${name}`); }` allows you to pass an object with a `name` property and access it directly in the function body.
    - **Spread Operator**: The spread operator (`...`) can be used to create shallow copies of objects or merge multiple objects into one.
    - **Object Methods**: JavaScript provides built-in methods like `Object.keys()`, `Object.values()`, and `Object.entries()` to work with objects effectively.
    - **Symbol Properties**: Symbols can be used as keys in objects, providing a way to create unique property identifiers that do not conflict with other properties.
    - **Computed Property Names**: Objects can have properties defined with dynamic keys using square brackets, allowing for more flexible object creation.
    - **Prototype Chain**: Objects can inherit properties and methods from their prototype, allowing for shared functionality across instances.

- **Object Literals**: Objects can be created using object literals, which provide a concise syntax for defining objects with properties and methods.

7. Explain JavaScript objects and provide an example demonstrating prototype-based inheritance.

A JavaScript object is a collection of key-value pairs, where keys are strings (or Symbols) and values can be any data type, including other objects or functions. Objects are fundamental to JavaScript and are used to model real-world entities, store data, and encapsulate behavior.

Prototype-Based Inheritance: JavaScript uses prototype-based inheritance, meaning objects can inherit properties and methods from other objects. Every object has an internal link to another object called its prototype. When you try to access a property that does not exist on the object itself, JavaScript looks up the prototype chain until it finds it or reaches the end.

Example:

```
const animal = {
  eats: true,
  walk() {
    console.log("Animal walks");
  },
};

const dog = Object.create(animal); // dog inherits from animal

dog.barks = true;
dog.walk(); // Output: Animal walks
console.log(dog.eats); // true (inherited)
console.log(dog.barks); // true (own property)
```

- Here, `dog` is created with `animal` as its prototype. It inherits the `eats` property and `walk` method from `animal`.
- You can check inheritance using `isPrototypeOf`:

```
console.log(animal.isPrototypeOf(dog)); // true
```

Constructor Functions and Prototypes: You can also use constructor functions to create objects with shared prototypes:

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function () {
  console.log(`Hello, my name is ${this.name}`);
};
const alice = new Person("Alice");
alice.greet(); // Output: Hello, my name is Alice
```

Edge Cases:

- Modifying the prototype affects all objects inheriting from it.
- You can override inherited properties by defining them on the object itself.

Question 8: What is the purpose of using AJAX in web applications?

The purpose of using AJAX in web applications includes:

1. **Asynchronous Communication:** AJAX allows web applications to communicate with the server asynchronously, meaning that data can be sent and received in the background without interfering with the user's interaction with the page.
2. **Partial Page Updates:** AJAX enables partial page updates, where only a specific portion of the web page is updated with new data, rather than reloading the entire page. This results in a smoother and more responsive user experience.
3. **Improved Performance:** By loading data in the background and updating only the necessary parts of the page, AJAX can significantly improve the performance and speed of web applications.
4. **Dynamic Content Loading:** AJAX allows for dynamic loading of content, such as fetching data from the server based on user actions (e.g., clicking a button, submitting a form) and updating the page with the new content.
5. **Enhanced User Experience:** With AJAX, web applications can provide a more interactive and seamless user experience, similar to that of desktop applications.

Question 9:

Edge Cases:

- Always check `e.target` to ensure the correct element is handled.
- Some events (like `focus`) do not bubble.

Question 26: What is the difference between synchronous and asynchronous code in JavaScript?

Synchronous code executes line by line, blocking further execution until the current operation completes.
Asynchronous code allows other operations to run while waiting for a task (like a network request) to finish.

Synchronous Example:

```
console.log("A");
alert("Pause here");
console.log("B"); // Runs after alert is closed
```

Asynchronous Example:

```
console.log("A");
setTimeout(() => console.log("B"), 1000);
console.log("C"); // C prints before B
```

Explanation:

- Synchronous: Each line waits for the previous one.
- Asynchronous: Some operations (timers, AJAX, Promises) run in the background.

Edge Cases:

- Asynchronous code can lead to callback hell if not managed well.
- Use Promises or async/await for better readability.

Question 27: What is a callback function? Provide an example.

A **callback function** is a function passed as an argument to another function, to be executed later (after an event, or when a task completes).

Example:

```
function greet(name, callback) {
  console.log("Hello, " + name);
  callback();
}
```

```
function sayBye() {
  console.log("Goodbye!");
}
greet("Alice", sayBye); // Output: Hello, Alice\nGoodbye!
```

Common Use Case:

- Asynchronous operations (e.g., setTimeout, event handlers, AJAX).

```
setTimeout(function () {
  console.log("Timer done!");
}, 1000);
```

Edge Cases:

- Callback hell: deeply nested callbacks can make code hard to read.
- Use Promises or async/await to avoid this.

Question 28: What is a Promise in JavaScript? How does it work?

A **Promise** is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value.

States:

- Pending: Initial state, neither fulfilled nor rejected.
- Fulfilled: Operation completed successfully.
- Rejected: Operation failed.

Example:

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Done!"), 1000);
});
promise.then((result) => console.log(result)); // Output after 1s: Done!
```

Chaining:

```
promise.then((data) => data + "!").then((str) => console.log(str)); // Output:
Done!!
```

Error Handling:

```
promise.catch((error) => console.error(error));
```

Edge Cases:

- A Promise can only be settled once.
- Unhandled rejections can cause issues; always use `.catch()`.

Question 29: Explain async/await in JavaScript with an example.

async/await is syntactic sugar over Promises, making asynchronous code look synchronous and easier to read.

Example:

```
function fetchData() {
  return new Promise((resolve) => setTimeout(() => resolve("Data!"), 1000));
}
async function getData() {
  const result = await fetchData();
  console.log(result);
}
getData(); // Output after 1s: Data!
```

Explanation:

- `async` marks a function as asynchronous, returning a Promise.
- `await` pauses execution until the Promise resolves.

Edge Cases:

- `await` can only be used inside `async` functions.
- Errors thrown inside async functions can be caught with `try...catch`.

Question 30: What is local storage in JavaScript? How do you use it?

Local storage is a web storage API that allows you to store key-value pairs in the browser, persisting even after the browser is closed.

Usage:

```
// Set item
localStorage.setItem("username", "Alice");
// Get item
const user = localStorage.getItem("username");
console.log(user); // Alice
// Remove item
localStorage.removeItem("username");
```

```
// Clear all
localStorage.clear();
```

Edge Cases:

- Only stores strings. Use `JSON.stringify`/`JSON.parse` for objects.
- Storage limit (~5MB per domain).
- Not available in some privacy modes.

Question 31: What is the difference between `==` and `===` in JavaScript?

- `==` is the **loose equality** operator. It compares values after type coercion.
- `===` is the **strict equality** operator. It compares both value and type.

Examples:

```
0 == false; // true
0 === false; // false
"5" == 5; // true
"5" === 5; // false
null == undefined; // true
null === undefined; // false
```

Best Practice:

- Use `===` to avoid unexpected type coercion.

Question 32: What is JSON? How do you parse and stringify it in JavaScript?

JSON (JavaScript Object Notation) is a lightweight data-interchange format, easy for humans to read and write, and easy for machines to parse and generate.

Parsing:

```
const jsonString = '{"name":"Alice","age":25}';
const obj = JSON.parse(jsonString);
console.log(obj.name); // Alice
```

Stringifying:

```
const user = { name: "Bob", age: 30 };
const str = JSON.stringify(user);
console.log(str); // {"name":"Bob","age":30}
```

Edge Cases:

- Only serializes data, not functions or undefined.
- Circular references cause errors.

Question 33: What is destructuring in JavaScript? Give examples for arrays and objects.

Destructuring allows you to unpack values from arrays or properties from objects into distinct variables.

Array Destructuring:

```
const arr = [1, 2, 3];
const [a, b, c] = arr;
console.log(a, b, c); // 1 2 3
```

Object Destructuring:

```
const person = { name: "Alice", age: 25 };
const { name, age } = person;
console.log(name, age); // Alice 25
```

Edge Cases:

- You can set default values: `const [x = 10] = []`.
- Nested destructuring is possible.

Question 34: What is the spread operator (`...`) in JavaScript? Provide use cases.

The **spread operator** (`...`) expands iterable elements (like arrays or objects) into individual elements.

Examples:

- **Array Copy:**

  ```
  const arr = [1, 2, 3];
  const copy = [...arr];
  ```

- **Array Merge:**

  ```
  const a = [1, 2];
  const b = [3, 4];
  ```

```
const merged = [...a, ...b]; // [1,2,3,4]
```

- **Object Copy/Merge:**

```
const obj1 = { a: 1 };
const obj2 = { b: 2 };
const merged = { ...obj1, ...obj2 };
```

- **Function Arguments:**

```
function sum(x, y, z) {
  return x + y + z;
}
const nums = [1, 2, 3];
sum(...nums); // 6
```

Edge Cases:

- Only shallow copies.
- For objects, later properties overwrite earlier ones.

Question 35: What is a template literal in JavaScript? Show its advantages.

Template literals (backticks ``) allow embedded expressions and multi-line strings.

Example:

```
const name = "Alice";
const msg = `Hello, ${name}!`;
console.log(msg); // Hello, Alice!
```

Advantages:

- Multi-line strings:

```
const text = `Line 1
Line 2`;
```

- Expression interpolation: `${expression}`
- Easier to read and maintain than string concatenation.

Question 36: What is a higher-order function? Give an example.

A **higher-order function** is a function that takes another function as an argument or returns a function.

Example:

```
function multiplyBy(factor) {
  return function (num) {
    return num * factor;
  };
}
const double = multiplyBy(2);
console.log(double(5)); // 10
```

Array methods like `map`, `filter`, and `reduce` are higher-order functions.

Question 37: What is the difference between `null` and `undefined`?

- `undefined`: A variable declared but not assigned a value.
- `null`: An assignment value that represents no value or object.

Examples:

```
let a;
console.log(a); // undefined
let b = null;
console.log(b); // null
```

Type:

- `typeof undefined` is "undefined"
- `typeof null` is "object" (quirk)

Question 38: What is the DOM? How does JavaScript interact with it?

The **DOM (Document Object Model)** is a tree-like structure representing the HTML elements of a web page. JavaScript can access and manipulate the DOM to change content, structure, and styles dynamically.

Example:

```
document.getElementById("demo").textContent = "Changed!";
```

Interactions:

- Select elements

- Change content/styles
- Add/remove elements
- Handle events

Question 39: What is event bubbling in JavaScript?

Event bubbling is the process where an event starts from the target element and bubbles up to its ancestors in the DOM tree.

Example:

```
<div id="parent">
  <button id="child">Click Me</button>
</div>
<script>
  document
    .getElementById("parent")
    .addEventListener("click", () => alert("Parent clicked"));
  document
    .getElementById("child")
    .addEventListener("click", () => alert("Button clicked"));
</script>
```

Explanation:

- Clicking the button triggers both the button and parent handlers (button first, then parent).

Edge Cases:

- Use `event.stopPropagation()` to prevent bubbling.

Question 40: What is the use of `preventDefault()` in JavaScript?

The `preventDefault()` method stops the default action of an event (like submitting a form or following a link).

Example:

```
<form id="myForm">
  <button type="submit">Submit</button>
</form>
<script>
  document.getElementById("myForm").addEventListener("submit", function (e) {
    e.preventDefault();
    alert("Form submission prevented!");
  });
</script>
```

Use Cases:

- Prevent form submission, link navigation, drag-and-drop defaults, etc.

Question 41: What is a closure? Give a practical use case.

A **closure** is a function that retains access to its lexical scope even after the outer function has finished executing.

Example:

```
function makeCounter() {
  let count = 0;
  return function () {
    return ++count;
  };
}
const counter = makeCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

Practical Use Case:

- Data privacy: Encapsulate private variables.
- Event handlers: Maintain state between calls.

Question 42: What is the difference between `forEach`, `map`, and `filter` in JavaScript?

- `forEach`: Executes a function for each array element. Does not return a new array.

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

- `map`: Creates a new array by applying a function to each element.

  ```
  const doubled = [1, 2, 3].map((x) => x * 2); // [2, 4, 6]
  ```

- `filter`: Creates a new array with elements that pass a test.

  ```
  const even = [1, 2, 3, 4].filter((x) => x % 2 === 0); // [2, 4]
  ```

Edge Cases:

- `forEach` cannot be chained (returns undefined).
- `map` and `filter` return new arrays and can be chained.
