Q.1 What's a Constructor and Its Purpose?

A constructor is a special method in object-oriented programming languages like JavaScript that is used to initialize and create objects of a class. It is typically called when an object is created using the `new` keyword. The constructor has the same name as the class and can be used to set initial values for the object's properties or perform any other necessary setup.

The purpose of a constructor is to ensure that newly created objects have a valid initial state by initializing their properties or performing any required setup. Constructors allow you to define the behavior that should occur when an object is created, such as initializing variables or setting default values. They help in creating objects with consistent and predictable initial states, making it easier to work with them throughout the program.

Q.2 Explain the `this` Keyword and Its Purpose?

The `this` keyword in JavaScript is a special keyword that refers to the current instance of an object or the context in which a function is called. Its purpose is to access and manipulate the properties and methods of the current object within its scope.

The value of `this` is determined dynamically at runtime based on how a function is called. When a function is called as a method of an object, `this` refers to the object itself. However, when a function is called in a different context or without any specific context, `this` can have different values. In such cases, `this` can refer to the global object (e.g., `window` in a browser) or be undefined, depending on whether strict mode is enabled.

The `this` keyword allows for dynamic and flexible code, enabling objects and functions to work with different data and contexts based on how they are invoked. It plays a crucial role in object-oriented programming by providing access to the object's properties and methods within its own scope.

Q.3 What's the `call`, `apply`, and `bind` methods, and what is the difference between them?

In JavaScript, the `call`, `apply`, and `bind` methods are used to manipulate the context (the value of `this`) within which a function is called.

1. The `call` method allows you to invoke a function with a specified `this` value and individual arguments passed as separate arguments. For example:
   ```javascript
   function sayHello(name) {
     console.log(`Hello, ${name}! I'm ${this.name}.`);
   }

   const obj = { name: 'John' };
   sayHello.call(obj, 'Alice');
   ```

```
// Output: Hello, Alice! I'm John.
```

2. The `apply` method is similar to `call`, but it accepts arguments as an array or an array-like object instead of individual arguments. For example:

```javascript
function sayHello(name) {
  console.log(`Hello, ${name}! I'm ${this.name}.`);
}

const obj = { name: 'John' };
sayHello.apply(obj, ['Alice']);
// Output: Hello, Alice! I'm John.
```

3. The `bind` method returns a new function with a bound `this` value and, optionally, some initial arguments. It allows you to create a function that, when called, will have a specified context and, optionally, some predefined arguments. For example:

```javascript
function sayHello(name) {
  console.log(`Hello, ${name}! I'm ${this.name}.`);
}

const obj = { name: 'John' };
const boundFunction = sayHello.bind(obj, 'Alice');
boundFunction();
// Output: Hello, Alice! I'm John.
```

The main difference between `call` and `apply` is how they accept arguments. `call` accepts arguments as individual arguments, whereas `apply` accepts them as an array or an array-like object. The `bind` method returns a new function with the bound `this` value and partially applied arguments, without immediately invoking the function.

Q.4 Explain OOPS?

OOPS stands for Object-Oriented Programming Paradigm. It is a programming paradigm that organizes data and behavior into reusable structures called objects. The fundamental principles of OOPS are:

1. Encapsulation: It is the bundling of data and related behaviors (methods) into a single unit called an object. Encapsulation hides the internal implementation details of an object and provides a public interface to interact with the object. It helps in achieving data abstraction and information hiding.

2. Inheritance: It is a mechanism that allows objects to inherit properties and methods from other objects. Inheritance enables the creation of a hierarchy of classes, where subclasses inherit the characteristics of their parent classes. It promotes code reuse and the creation of specialized classes based on existing ones.

3. Polymorphism: It refers to the ability of objects of different classes to respond to the same message or method call. Polymorphism allows objects to be treated as instances of their own class or as instances of their parent class. It enables code flexibility and extensibility by providing a common interface for multiple objects.

4. Abstraction: It involves representing complex real-world entities as simplified models within the program. Abstraction focuses on the essential properties and behaviors of an object while ignoring the unnecessary details. It helps in managing program complexity and provides a high-level view of the system.

Q.5 What's Abstraction and Its Purpose?

Abstraction is a fundamental concept in object-oriented programming that involves representing complex real-world entities as simplified models within the program. It focuses on the essential properties and behaviors of an object while hiding unnecessary details.

The purpose of abstraction is to manage program complexity and provide a high-level view of the system. It allows programmers to create classes or objects that abstract away the complexity of their internal implementation, exposing only the relevant information and functionality to the outside world. By hiding internal details, abstraction provides a simplified interface for interacting with objects, making it easier to use and understand them.

Abstraction helps in modularizing code, as it allows programmers to work with higher-level concepts and entities rather than dealing with low-level implementation details. It promotes code maintainability, reusability, and scalability by encapsulating complexity and providing a clear separation between the interface and the implementation.

Q.6 What's Polymorphism and the Purpose of It?

Polymorphism is a concept in object-oriented programming that allows objects of different classes to be treated as instances of a common superclass or interface. It enables objects to respond to the same message or method call in different ways, based on their specific class implementations.

The purpose of polymorphism is to provide code flexibility and extensibility. By treating objects of different classes as instances of a common superclass or interface, you can write code that

operates on the shared interface without having to know the specific class of the object. This promotes code reusability and allows for more generic and flexible programming.

Polymorphism allows you to write code that can work with a variety of objects, as long as they adhere to a common interface. It simplifies code maintenance and enhances scalability by allowing new classes to be added without modifying

existing code. Polymorphism is a powerful tool for designing modular and extensible systems.

Q.7 What's Inheritance and the Purpose of It?

Inheritance is a mechanism in object-oriented programming that allows objects or classes to inherit properties and methods from other objects or classes. It enables the creation of a hierarchy of classes, where subclasses inherit the characteristics (fields and methods) of their parent classes.

The purpose of inheritance is to promote code reuse and the creation of specialized classes based on existing ones. It allows you to define common attributes and behaviors in a base class and extend or modify them in derived classes. Inheritance provides a way to establish a relationship between classes, where the child class (subclass) inherits the properties and methods of the parent class (superclass).

By inheriting from a superclass, a subclass can reuse the code and behaviors defined in the superclass, reducing code duplication and promoting modularity. Inheritance allows for hierarchical organization of classes, making the code more organized and easier to maintain. It also enables polymorphism, as objects of different classes can be treated as instances of their common superclass.

Q.8 What's Encapsulation and the Purpose of It?

Encapsulation is a fundamental concept in object-oriented programming that involves bundling data and related behaviors (methods) into a single unit called an object. It is the practice of hiding the internal implementation details of an object and providing a public interface to interact with the object.

The purpose of encapsulation is to achieve data abstraction and information hiding. By encapsulating data within an object, you can control access to it and enforce constraints on how it is manipulated. The internal state of an object is hidden from external code, which can only interact with the object through its public interface (methods).

Encapsulation provides several benefits:

1. Modularity: Encapsulation allows you to modularize code by grouping related data and behaviors into a single object. This promotes code organization and makes it easier to understand and maintain.

2. Data Protection: By hiding the internal state of an object, encapsulation protects the data from being directly manipulated or accessed inappropriately. The object's methods provide controlled access to the data, ensuring proper validation and manipulation.

3. Code Flexibility: Encapsulation allows you to change the internal implementation of an object without affecting the code that uses the object. As long as the public interface remains the same, the external code can remain unchanged.

4. Code Reusability: Encapsulation promotes code reuse by encapsulating commonly used behaviors within objects. Objects can be instantiated and used in different parts of the program without duplicating the code.

Q.9 Explain Class in JavaScript?

In JavaScript, a class is a blueprint or a template for creating objects. It is a syntactical addition to JavaScript introduced in ECMAScript 2015 (ES6) to provide a more familiar object-oriented programming style similar to other programming languages.

A class defines the properties and methods that objects created from it will have. It encapsulates the data (properties) and behaviors (methods) that are common to all instances of that class. The class serves as a blueprint, and each object created from the class is an instance of that class.

Here's an example of a class in JavaScript:

```javascript
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  sayHello() {
    console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);
  }
}

// Creating an object (instance) of the Person class
```

```
const john = new Person('John', 25);

// Calling the method on the object
john.sayHello();
// Output: Hello, my name is John and I'm 25 years old.
```

In the above example, the `Person` class has a constructor method, which is a special method that is automatically called when a new object is created from the class. The constructor is used to initialize the object's properties.

The class also has a `sayHello` method, which can be called on instances of the class to display a greeting message.

Classes in JavaScript provide a convenient and intuitive way to define objects and their behavior, making it easier to organize and structure code in an object-oriented manner.

Q.10 What's the `super` keyword and what does it do?

In JavaScript, the `super` keyword is used to call the constructor or methods of a parent class from within a subclass. It allows the subclass to inherit and leverage the functionality defined in the parent class.

When used in a constructor, `super()` is used to call the constructor of the superclass and initialize the inherited properties. It is mandatory to call `super()` within a subclass constructor if the subclass extends a parent class with a constructor. This ensures that the superclass's constructor is executed before initializing the subclass's own properties.

Here's an example that demonstrates the use of `super` in a subclass:

```javascript
class Vehicle {
  constructor(color) {
    this.color = color;
  }

  start() {
    console.log(`The vehicle starts.`);
  }
}

class Car extends Vehicle {
  constructor(color, brand) {
    super(color); // Calling the constructor of the Vehicle class
```

```
    this.brand = brand;
  }

  start() {
    super.start(); // Calling the start method of the Vehicle class
    console.log(`The ${this.brand} car starts.`);
  }
}

const myCar = new Car('blue', 'Honda');
myCar.start();
// Output:
// The vehicle starts.
// The Honda car starts.
```

In the above example, the `Car` class extends the `Vehicle` class using the `extends` keyword. The `Car` class has its own constructor that calls `super(color)` to invoke the `Vehicle` class's constructor and initialize the inherited `color` property. The `start` method of the `Car` class uses `super.start()` to call the `start` method of the `Vehicle` class before adding its own custom behavior.

The `super` keyword is essential for maintaining the inheritance chain and accessing the functionality provided by the parent class in the subclass.