Functions:

1. What is a function in JavaScript?

   A function in JavaScript is a block of reusable code that performs a specific task. It is defined once but can be called multiple times from different parts of the code.

2. How do you define a function in JavaScript?

   You can define a function in JavaScript using the function keyword followed by the function name, parameters (if any), and the function body enclosed in curly braces {}.

3. What is the difference between function declarations and function expressions in JavaScript?

   Function declarations are defined using the function keyword followed by a name, while function expressions are defined by assigning a function to a variable. Function declarations are hoisted to the top of the scope, while function expressions are not hoisted.

4. How do you call a function in JavaScript?

   To call a function in JavaScript, you simply write the function name followed by parentheses (), optionally passing any required arguments inside the parentheses.

5. What is the purpose of parameters in JavaScript functions?

   Parameters in JavaScript functions are placeholders for values that the function expects to receive when it is called. They allow functions to accept input values and perform actions based on those values.

6. How do you pass arguments to a function in JavaScript?

   Arguments are passed to a function by placing them within the parentheses () when calling the function. These arguments will be assigned to the parameters defined in the function declaration or expression.

7. How do you return a value from a function in JavaScript?

   You can return a value from a function using the return keyword followed by the value you want to return. Once a return statement is encountered in a function, the function will stop executing and return the specified value.

8. What is the difference between the `return` statement and `console.log()` in JavaScript?

   The return statement is used within a function to return a value to the caller of the function.
   console.log() is used to output data to the console for debugging purposes and does not affect the flow of the function. It does not return any value to the caller.

9. Explain the concept of function scope in JavaScript.

   Function scope refers to the visibility and accessibility of variables within a function. Variables defined within a function are only accessible within that function (unless they are explicitly returned or accessed from a higher scope).

10. What is a callback function in JavaScript?

    A callback function in JavaScript is a function that is passed as an argument to another function, and it is executed after a particular task is completed or when a certain condition is met. Callback functions are commonly used in asynchronous operations, event handling, and functional programming paradigms.

11. How do you use arrow functions in JavaScript?

    Arrow functions are a concise way to write function expressions in JavaScript. They have a shorter syntax compared to traditional function expressions and do not bind their own this, arguments, super, or new.target. Arrow functions are defined using the => syntax. Here's a basic example:

    ```
    // Traditional function expression
    let add = function(a, b) {
        return a + b;
    };

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

12. What is the `this` keyword in JavaScript functions?

    In JavaScript, the this keyword refers to the object that is currently executing the code. Its value is determined by how a function is called, not where the function is declared. The value of this can vary depending on the context in which the function is invoked.

13. Explain the concept of closures in JavaScript.

    Closures occur when a function is able to remember and access its lexical scope even when it is executed outside that lexical scope. This means that a function "closes over" variables from its surrounding scope, preserving those variables even after the outer function has finished executing. Closures are commonly used to create private variables, encapsulate state, and create functions with persistent state across multiple calls.

14. How do you use the `bind()`, `call()`, and `apply()` methods in JavaScript?

    bind(): The bind() method creates a new function that, when called, has its this keyword set to a specified value. It allows you to permanently bind a function to a specific context.
    call(): The call() method calls a function with a given this value and arguments provided individually.
    apply(): The apply() method calls a function with a given this value and an array or array-like object of arguments.

15. What are higher-order functions in JavaScript? Provide examples.

    Higher-order functions are functions that can take other functions as arguments or return functions as their results. They enable functional programming paradigms and allow for more expressive and flexible code. Examples include map, filter, and reduce functions on arrays:

```
// Higher-order function: map

let numbers = [1, 2, 3, 4, 5];

let doubled = numbers.map(num => num * 2); // Returns [2, 4, 6, 8, 10]


// Higher-order function: filter

let evenNumbers = numbers.filter(num => num % 2 === 0); // Returns [2, 4]


// Higher-order function: reduce

let sum = numbers.reduce((acc, num) => acc + num, 0); // Returns 15
```

Classes:

16. What is a class in JavaScript?

    A class in JavaScript is a blueprint for creating objects with predefined properties and methods. It provides a convenient way to create multiple objects with similar characteristics.

17. How do you define a class in JavaScript?

    You can define a class in JavaScript using the class keyword followed by the class name and the class body enclosed in curly braces {}. Here's an example:

```
class Person {
    // Class methods and properties go here
}
```

18. What is the purpose of the `constructor()` method in JavaScript classes?

    The constructor() method is a special method that is automatically called when a new instance of a class is created. It is used to initialize the object's properties and perform any setup necessary for the object.

19. How do you create an instance of a class in JavaScript?

    You can create an instance of a class using the new keyword followed by the class name and any required arguments for the constructor method. Here's an example:

```
let person = new Person();
```

20. What are class properties and methods in JavaScript?

Class properties are variables defined within a class, while class methods are functions defined within a class. These properties and methods are accessible on instances of the class.

21. How do you define instance methods in JavaScript classes?

Instance methods are defined within the class body without the static keyword. They operate on individual instances of the class and typically access or modify instance-specific data.

```
class Person {
   sayHello() {
      console.log('Hello!');
   }
}
```

22. What is the difference between instance methods and static methods in JavaScript classes?

- Instance methods are called on instances of the class and operate on instance-specific data.
- Static methods are called on the class itself and do not have access to instance-specific data. They are often used for utility functions or operations that do not require access to instance data.

23. How do you use the `static` keyword in JavaScript classes?

You can define static methods or properties in a class using the static keyword. Static methods are called on the class itself rather than on instances of the class.

```
class MathUtils {
   static add(a, b) {
      return a + b;
   }
}

MathUtils.add(2, 3); // Returns 5
```

24. Explain the concept of inheritance in JavaScript classes.

Inheritance is a mechanism by which a class can inherit properties and methods from another class. The class that inherits from another class is called a subclass or derived class, and the class it inherits from is called a superclass or base class.

25. How do you implement inheritance in JavaScript?

In JavaScript, you can implement inheritance using the extends keyword to create a subclass that inherits from a superclass. The subclass can then access the properties and methods of the superclass.

```
class Animal {
    speak() {
        console.log('Animal speaks');
    }
}

class Dog extends Animal {
    bark() {
        console.log('Dog barks');
    }
}

let dog = new Dog();
dog.speak(); // Inherits from Animal class
dog.bark();  // Defined in Dog class
```

26. What is method overriding in JavaScript classes?

    Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. This allows the subclass to customize the behavior of inherited methods.

27. How do you access parent class methods in a subclass in JavaScript?

    You can access parent class methods in a subclass by using the super keyword followed by the method name and parentheses. This allows you to call the superclass's method from within the subclass.

28. What is the purpose of the `super` keyword in JavaScript classes?

    The super keyword is used in subclasses to call methods or access properties of the superclass. It can be used in the constructor to call the superclass's constructor, or in other methods to call specific methods of the superclass.

29. How do you use the `extends` keyword in JavaScript classes for inheritance?

    The extends keyword is used to create a subclass that inherits from a superclass. It is followed by the name of the superclass, allowing the subclass to inherit all properties and methods of the superclass.

```
class Subclass extends Superclass {
    // Subclass definition
}
```

30. Explain the concept of encapsulation in JavaScript classes.

    Encapsulation is the practice of bundling the data (properties) and methods (functions) that operate on the data within a single unit, such as a class. It allows for better organization, modularization, and abstraction of code by hiding the internal implementation details of an object and exposing only the necessary interfaces.

Advanced Concepts:

31. What is prototypal inheritance in JavaScript?

Prototypal inheritance is a mechanism in JavaScript where objects inherit properties and methods from other objects through prototype chains. Each object in JavaScript has a prototype object, and when a property or method is accessed on an object, if it does not exist on the object itself, JavaScript looks for it in the prototype chain.

32. How do you create and use prototypes in JavaScript?

Prototypes in JavaScript are created automatically when you define a constructor function or a class. You can add properties and methods to the prototype object using the prototype property of the constructor function or class.

```
function Person(name) {
  this.name = name;
}

Person.prototype.sayHello = function() {
  console.log('Hello, ' + this.name + '!');
};
```

33. What is the prototype chain in JavaScript?

The prototype chain is the mechanism through which JavaScript resolves property and method lookups on objects. When a property or method is accessed on an object, if it does not exist on the object itself, JavaScript follows the prototype chain by looking for it in the prototype of the object, and recursively in the prototypes of its parent objects.

34. How do you create private variables and methods in JavaScript classes?

Private variables and methods can be created in JavaScript classes by using closure and scope. By defining variables and methods within the constructor function or using ES6 closures, you can create private members that are inaccessible from outside the class.

35. What is the purpose of the `get` and `set` keywords in JavaScript classes?

The get and set keywords are used to define getter and setter methods for accessing and modifying class properties, respectively. They allow you to define custom behavior when getting and setting the value of a property, providing more control and encapsulation over class data

36. How do you implement getter and setter methods in JavaScript classes?

Getter and setter methods can be implemented in JavaScript classes using the get and set keywords, respectively, followed by the method name.

```
class Person {
  constructor(name) {
    this._name = name;
  }

  get name() {
```

```
        return this._name;
    }

    set name(newName) {
        this._name = newName;
    }
}

let person = new Person('Alice');
console.log(person.name); // Output: Alice
person.name = 'Bob';
console.log(person.name); // Output: Bob
```

37. What are mixins in JavaScript? Provide examples.

Mixins are a way to add functionality to objects in JavaScript by combining multiple objects or classes. They allow for code reuse and composition without inheritance.

```
// Example of a mixin
const canEat = {
    eat(food) {
        console.log(`${this.name} eats ${food}`);
    }
};

const canSleep = {
    sleep(hours) {
        console.log(`${this.name} sleeps for ${hours} hours`);
    }
};

class Person {
    constructor(name) {
        this.name = name;
    }
}

// Apply mixins to the Person class
Object.assign(Person.prototype, canEat, canSleep);

let person = new Person('Alice');
person.eat('pizza');   // Output: Alice eats pizza
person.sleep(8);       // Output: Alice sleeps for 8 hours
```

38. How do you implement mixins in JavaScript classes?

Mixins can be implemented in JavaScript classes by using Object.assign() to merge the properties and methods of mixin objects into the class prototype.

```
class MyClass {}

Object.assign(MyClass.prototype, mixin1, mixin2, ...);
```

What is the purpose of the `instanceof` operator in JavaScript?

The instanceof operator is used to check if an object is an instance of a particular class or constructor function. It returns true if the object is an instance of the specified class, otherwise false.

39. How do you check the type of an object in JavaScript?

In JavaScript, you can check the type of an object using the typeof operator, which returns a string indicating the type of the operand, or using the instanceof operator to check if an object is an instance of a particular class.0

40. What are closures in JavaScript, and how do they work?

Closures are functions that have access to variables from their containing scope, even after the scope has closed. They "remember" the environment in which they were created and have access to variables from that environment.

41. Explain the concept of lexical scoping in JavaScript.

Lexical scoping refers to the way variables are resolved in nested functions based on the lexical structure of the code. In JavaScript, functions are executed using the scope chain at the time of their definition, not at the time of their execution. This means that a function can access variables from its parent scope, even if the parent function has finished executing.

42. How do you use closures to create private variables in JavaScript?

Private variables can be created in JavaScript using closures. By defining variables within a function scope and returning inner functions that have access to those variables, you can create private variables that are inaccessible from outside the function scope.

43. What are IIFE (Immediately Invoked Function Expressions) in JavaScript?

IIFE stands for Immediately Invoked Function Expression. It is a JavaScript function that is executed immediately after it is defined, without being stored in a variable. IIFE is often used to create a private scope and avoid polluting the global namespace.

44. How do you create and use an IIFE in JavaScript?

An IIFE can be created by defining an anonymous function and immediately invoking it using parentheses ().

```
(function() {
    // IIFE code here
})();
```

45. What is the purpose of the `new` keyword in JavaScript?

In JavaScript, the new keyword is used to create an instance of a constructor function or a class. It initializes a new object, binds this to the new object, sets up the prototype chain, and returns the newly created object.

46. Explain the concept of constructor functions in JavaScript.

Constructor functions in JavaScript are special functions that are used to create and initialize objects. They are typically used as blueprints for creating multiple objects with similar properties and methods. Constructor functions are called with the new keyword, which creates a new object, binds this to the new object, and implicitly returns the newly created object.

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

let person1 = new Person('Alice', 30);
let person2 = new Person('Bob', 25);
```

47. How do you create custom error classes in JavaScript?

You can create custom error classes in JavaScript by extending the built-in Error object or any of its subclasses. This allows you to define your own error types with custom properties and methods.

```
class CustomError extends Error {
    constructor(message) {
        super(message);
        this.name = 'CustomError';
    }
}

throw new CustomError('Something went wrong');
```

48. What are generators in JavaScript? Provide examples.

Generators are functions in JavaScript that can pause and resume their execution, allowing for the generation of a sequence of values over time. They are defined using the function* syntax and use the yield keyword to yield values one at a time.

```
function* generateNumbers() {
    yield 1;
    yield 2;
    yield 3;
}

let iterator = generateNumbers();
console.log(iterator.next()); // Output: { value: 1, done: false }
console.log(iterator.next()); // Output: { value: 2, done: false }
console.log(iterator.next()); // Output: { value: 3, done: false }
console.log(iterator.next()); // Output: { value: undefined, done: true }
```

How do you use generators to create iterators in JavaScript?

Generators are often used to create iterators in JavaScript. Since generators can pause and resume their execution, they are ideal for generating values lazily one at a time.

```
function* generateNumbers() {
    let i = 0;
    while (true) {
        yield i++;
    }
}

let iterator = generateNumbers();
console.log(iterator.next().value); // Output: 0
console.log(iterator.next().value); // Output: 1
console.log(iterator.next().value); // Output: 2
// ...
```