27/2/24

ASSIGNMENT 1

Unit - 1, 2 & 3 Basics of JavaScript; OOP concepts, Proxies and Reflect & Functions, promises and async Function.

Q.1 What are the different ways of executing JavaScript in browser?

→ In the browser, Javascript can be executed in several ways:

i) Inline : JavaScript code can be directly embedded within HTML elements using event handlers attributes.

ii) Internal : JavaScript code can be placed within '<script>' tags directly within the HTML file, either in the '<head>' or '<body>' section.

iii) External : JavaScript code can be stored in separate '.js' files and linked to the HTML file using the '<script src="...">' tag.

iv) Using Browser Console : JavaScript can be ~~respon~~ executed directly in the browser's developer ~~consale~~ for debugging purposes or quick testing.

Q.2 Exemplify unary and bitwise ~~opter~~ operators using JavaScript.

→ i) Unary Operator

> Unary operator is used to operate on a single operand, which include the unary plus (+), unary minus (-), increment (++)

and decrement (--).

eg:
```
let x = 11;
console.log(+x);   // 11
console.log(-x);   // -11
console.log(++x);  // 12
console.log(--x);  // 11
```

ii) Bitwise Operators

* Bitwise operators treat their operands as a sequence of 32 bits (zeros and ones) rather than as decimal, hexadecimal or octal numbers.

eg:
```
let a = 5;
let b = 7;
console.log(a & b);   // 1
console.log(a | b);   // 15
console.log(~a);      // -6
console.log(a ^ b);   // 14
console.log(a << 1);  // 10
console.log(a >> 1);  // 2
```

Q.3 Explain iterative and reduction methods of an array.

→ i) Iterative Methods

Iterative methods involve looping through each element of the array and performing some operation on each element individually. It includes for loops,

while loops and methods like forEach(), map(), filter(), some() and every().

- forEach(): Executes a provided function once for each array element.
- map(): Creates a new array with the results of calling a provided function on every element in the calling array.
- filter(): Creates a new array with all elements that pass the test implemented by the provided function.

eg:
```
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(num => console.log(num * 2));
const doubledNumbers = numbers.map(num =>
    num * 2);
console.log(doubledNumbers);
```

ii) Reduction Methods

Reduction methods involve combining all elements of an array into a single value by repeatedly applying a combining operation. These methods typically take an initial value and a combining function as arguments.

- reduce(): Executes a reducer function on each element of the array, resulting in a single output value.

- reduceRight() : Similar to 'reduce()', but processes the array from right to left.

eg:
```
const numbers = [1,2,3,4,5]
const sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log(sum);
```

Q.4 Differentiate var and let. Write JavaScript code to explain the difference.

→

Var	let
The variables defined with var have function scope.	The variables defined with let have block scope.
It allows re-declaration within the same scope	Throws an error if re-declared within the same scope.
Can be declared without initialization	Needs to be initialized before use.
Introduced in ECMAScript 5 (ES5)	Introduced in ECMAScript 6 (ES6)
Syntax: var name = value;	Syntax: let name = value;

eg:
```
function varExample() {
    console.log(x);   // undefined
    var x = 10;
    console.log(x);   // 10
}
varExample();

function letExample() {
    console.log(y);   // Reference Error
    let y = 20;
    console.log(y);   // 20
}
letExample();
```

Q.5 Exemplify conditional and assignment operators using JavaScript.

—> i) Conditional Operator

The conditional operator, also known as the ternary operator, is a shorthand for an 'if...else' statement. It evaluates a condition and returns one of two expressions depending on whether the condition is true or false.

eg:
```
let age = 20;
let message = (age >= 18) ? "Adult" : "Minor";
console.log(message);
```

ii) Assignment Operator

Assignment operator are used to

assign values to variables, which includes '=', '+=', '-=', '*=', '/=', '%=', etc.

eg:
```
let x = 5;
console.log(x+=5);   // 10
console.log(x-=5);   // 5
console.log(x*=5);   // 25
console.log(x/=5);   // 5
console.log(x%=1);   // 0.
```

Q.6 State the usage of typed array and explain the creation of the same using JavaScript.

→ Typed arrays are array-like objects that provide a way to work with binary data in a structured manner. They offer a more efficient way to handle raw binary data compared to traditional arrays. They are especially useful for tasks such as manipulating images, processing network packets, working with WebGL.

There are several types of typed arrays like Int8Array, Int16Array, Int32Array, Float32Array, Float32Array.

→ To create a typed array, you can use the constructor of the respective typed array type and pass in either an array or a length.

eg:
```
let myArray = [1,2,3,4,5];
let typedArray = new Int32Array(myArray);
console.log(typedArray);
```

Q.7 Enlist various data types. Exemplify any one.

→ There are several data types that can be classified into two main categories:

1. Primitive Data Types

i) Number: Represents numeric data, including integers and floating-point numbers.

ii) String: Represents sequences of characters, enclosed in single or double quotes.

iii) Boolean: Represents true or false values.

iv) Symbol: Represents a unique identifier.

v) Null: Represents the intentional absence of any object value.

vi) Undefined: Represents a variable that has been declared but not assigned a value.

2. Non-Primitive Data Types

i) Object: Represents a collection of key-value pairs, where keys are strings and values can be any data type.

ii) Array: Represents a special type of object used to store ordered collections of same data type.

iii) Date: Represents dates and times.

eg:
```
let name = "Hello, Tony!";
console.log(typeof name); // string
```

Q.8 Explain scope of variables using ~~Java~~ JavaScript

→ The Scope of a variable refers to the region of code where the variable is accessible. Understanding scope is crucial for managing variable lifetimes, avoiding naming conflicts and writing maintainable code.

i) Global Scope

Variables declared outside of any function or declared with 'var' keyword at the top-level of a script, have global scope. They are accessible from anywhere.

eg:
```
var globalVariable = "Global";
function myFunction() {
    console.log(globalVariable);
}
myFunction();
```

ii) Local scope

Variables declared inside a function have local scope. They are only accessible within the function where they are defined.

eg:
```
function myFunction() {
    var localVar = "local";
    console.log(localVar);
}
myFunction();
// console.log(localVar); // Error
```

iii) Block scope

Variables declared with 'let' and 'const' have block scope, which means they are only accessible within the block ('{}') where they are defined. Block scope includes if statements, for, while loops, etc.

eg:
```
if (true) {
    let blockVar = "Block";
    console.log(blockVar);
}
// console.log(blockVar); // Error
```

Q.9 State the difference between Objects & Maps.

—↓

Objects	Maps
They are collection of key-value pairs where keys can be strings or symbols.	They are collection of key-value pairs where keys can be any data type.
Here, keys can be mutable	Here, keys can be mutable or immutable
They are defined using curly braces '{}'.	They are defined using the 'Map' constructor.
Size is determined by the number of properties	Size is determined by the number of key-value pairs.
commonly used for representing structured data	Preferred for key-value pairs when keys are unknown.

Q.10 Explain various methods and properties of an object.

—↓ Objects are collections of key-value pairs, where each key is a string and each value can be of any data type. Properties are simply variables associated within an

object, while methods are functions associated with an object that can perform some action.

- Object Methods :-

i) Object.keys() : Returns an array of a given object's own enumerable property names.

ii) Object.values() : Returns an array of a given object's own enumerable property values.

iii) Object.entries() : Returns an array of a given object's own enumerable property [key, value] pair.

iv) Object.assign() : Copies the values of all enumerable own properties from one or more source objects to a target objects.

v) Object.freeze() : Freezes an object, preventing new properties from being added to it and existing properties from being removed or changed.

- Object properties :-

* Property Access : we can access properties of

an object using dot notation or bracket notation.

eg:
```
let person = {
    name: 'Tony',
    age: 21,
    greet: function() {
        console.log(this.name);
    }
};
console.log(person.name);
person.greet();
console.log(Object.keys(person));
```

Q.11 Take an object containing multiple key-value pair and iterate it using for-in and for-of statement.

—↳
```
let person = {
    name: "Tony",
    age: 21,
    city: "New York"
};
// using for-in statement
for (let key in person) {
    console.log(key + ": " + person[key]);
}
// using for-of statement
for (let [key, value] of Object.entries(person)) {
    console.log(key + ": " + value);
}
```

Output : Same for Both statement
name : Tony
age : 21
city : New York.

Q.12 Describe RegExp instance properties.

→ Regular Expressions are represented by RegExp objects which have several instance properties that provide information about the regular expression pattern.

i) Source : This property returns a string representing the regular expression pattern.

ii) flags : This property returns a string containing the flags used with the regular expressions, Flags modify the behavior of the regExp pattern.

iii) global : This property indicates whether the 'g' flags is used in the regular expression.

iv) ignoreCase : This property indicates whether the 'i' flag is used in regular expression.

v) multiline : This property indicates whether the 'm' flag is used in regular expression.

vi) sticky : This property indicates whether the 'y' flag is used in regular expression.

(vii) unicode : This property indicates whether the 'u' flag is used in regular expressions.

eg :
```
let pattern = /\d+/gimu;
console.log(pattern.source); // \d+
console.log(pattern.flags); // gimu
console.log(pattern.global); // true
console.log(pattern.ignoreCase); // true
console.log(pattern.multiline); // true
console.log(pattern.sticky); // false
console.log(pattern.unicode); // true
```

Q.13 Differentiate iterator and generator.

Iterator	Generator
An iterator is an object that enables traversal through a sequence.	A generator is a special type of iterator that simplifies the creation of iterators.
class is used to implement an iterator	Function is used to implement a generator
Iterator user iter() and next() functions	Generator uses yield keyword
Every iterator is not a generator.	Every generator is an iterator.
local variables aren't used here.	All the local variables before the yield function are stored.

Q.14 Describe data and accessor properties of an object.

-> Object properties can be classified into two main categories:

i) Data Properties: They are the properties that directly contain a data value. Each data property has four attributes:

1. Value: The value associated with the property
2. Writable: A boolean indicating whether the value of the property can be changed.
3. Enumerable: A boolean indicating whether the property can be enumerated using a 'for... in'.
4. Configurable: A boolean indicating whether the property can be deleted and whether its attributes can be changed.

ii) Accessor Properties: They are the properties that define a function to be called when the property is read (getter) or written (setter). Each accessor property has two attributes:

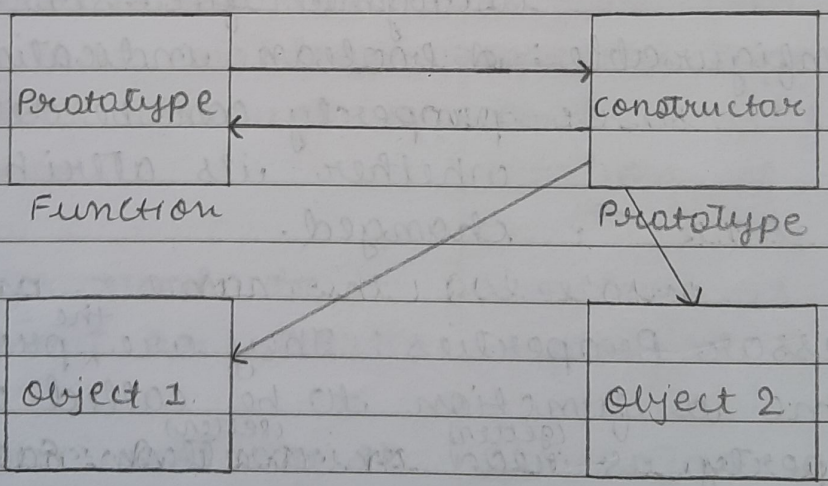
1. GEt: A function that is called when the property is read.
2. Set: A function that is called when the property is written to.

Q.15 Define Prototype. Explain working of prototype using a diagram.

—↓ The prototype is a mechanism through which objects inherit properties and methods from other objects. Each object has an internal property called [[Prototype]] that points to another object known as its prototype.

When you try to access a property or method object on an object that doesn't exist on the object itself, JavaScript will look for it in the object's prototype and if it's not found there, it will continue to look up the prototype chain until it reaches the end.

—↓



eg:
```
function Person(name) {
    this.name = name;
}
Person.prototype.sayHello = function() {
    console.log(this.name);
```

```
};
let person1 = new Person ('Tony');
console.log(person1.name);
person1.sayHello();
```

Q.16 Using a demonstrative JavaScript implement inheritance using JavaScript class.

—4

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    speak() {
        console.log(this.name + 'make a noise');
    }
}
class Dog extends Animal {
    constructor(name) {
        super(name);
    }
    speak() {
        console.log(this.name + ' braks.');
    }
}
let animal = new Animal('Animal');
let dog = new Dog('Dog');
animal.speak(); // Animal
dog.speak(); // Dog Barks.
```

Q.17 Write Javascript code to read properties of an object. Also explain functions used in the Javascript.

—↓

```
let person = {
    name : 'Tony',
    age : 21
    city : 'New York'
};
function readProperties (obj) {
    for (let key in obj) {
        if (obj.hasOwnProperty(key)) {
            console.log (key + ' : ' + obj[key]);
        }
    }
}
readProperties (person);
```

—↓ Function used in the above code are:-

i) Object.keys (obj) : It returns an array containing the names of the enumerable own properties of the object. ~~obj~~.

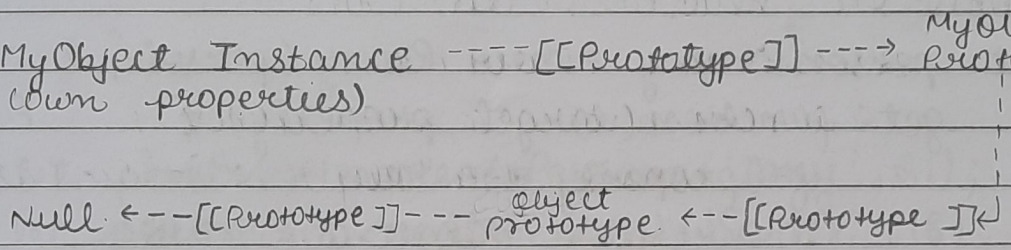
ii) obj.hasOwnProperty (key) : It checks whether the object has a property with the specified name as its own property. Used to filter out inherited properties

iii) console.log() : It is used to log messages to the console.

- The readProperties() function takes an object as a parameter and reads its properties using a for... in loop.

Q.18 Explain property hierarchy of a prototype using a diagram.

→ The property hierarchy of a prototype is as follows :-



- In this diagram:
  - The arrow labeled [[Prototype]] represents the link from an object to its prototype object.
  - "Own Properties" are properties defined directly on the instance.
  - Properties shared by MyObject instances are properties that exist on the MyObject prototype and are accessible by all ~~objects~~ instance
  - ~~The~~ of MyObject
  - Properties shared by all objects are properties that exist on the object prototype and are accessible by all objects.
  - The null at the end of the chain represents the end. JavaScript will stop looking up the chain when it reaches null.

Q.19 Using Javascript, create a proxy object for an object named person that will check if the person is allowed to vote based on the age property.

—↓

```
let person = {
    name: "Tony",
    age: 21
};

let personProxy = new Proxy(person, {
    get: function(target, property) {
        if (property === "canVote") {
            return target.age >= 18;
        }
        return target[property];
    }
});

console.log(personProxy.name)    // Tony
console.log(personProxy.age)     // 21
console.log(personProxy.canVote) // true
```

Q.20 Explain object creation using :

a) Factory Pattern.

—↓ The Factory Pattern involves creating objects using a factory ~~pattern~~ function, which is a function that returns objects. It provides a way to create multiple instances of objects without the need for ~~constructors~~ constructors.

eg:
```
function createPerson(name, age) {
    return {
        name: name,
        age: age,
        greet: function() {
            console.log('Hello, my name is' +
            this.name + ' and I am' + this.age +
            'years old.');
        }
    };
}
let person1 = createPerson('Tony', 21);
let person2 = createPerson('Spider', 18);
person1.greet();
person2.greet();
```

Output: Hello, my name is Tony and I am 21 years old.
Hello, my name is Spider and I am 18 years old

b) Function Constructor Pattern
→ Here, objects are created using constructor functions, which are regular functions used with 'new' keyword to create instances. They act as blueprint for creating objects. They typically use 'this' keyword to define properties and methods for instances.

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

```
    this.sayName = function() {
        console.log(this.name);
    };
}
let person = new Person('Tony', 21);
person.sayName();   // Tony
```

c) Prototype Pattern

—> This is similar to the function constructor pattern but focuses more on utilizing prototype chain for inheritance. Instead of defining methods directly in the constructor function, methods are added to the prototype object of the constructor function.

eg:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.greet = function() {
    console.log(this.name + '&' + this.age);
};
let person = new Person('Tony', 21)
person.greet();  // Tony & 21
```

Q.21 Using a demonstrative Javascript implement inheritance using prototype chaining method.

—>

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

```
Animal.prototype.speak = function() {
    console.log(this.name + ' makes noise.');
};
function Dog(name) {
    Animal.call(this, name);
}
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

Dog.prototype.bark = function() {
    console.log(this.name + 'Barks');
};
let animal = new Animal('Animal');
let dog = new Dog('Dog');
animal.speak();  // Animal makes noise.
dog.speak();     // Dog makes noise.
dog.bark();      // Dog Barks.
```

Q.22 Explain has() an reflect API.

—4 The Reflect API provides a collection of methods that provide default implementations of various operations on objects. One of the methods provided by Reflect API is 'Reflect.has()'.

This method is used to check if a property exists on an object. It returns a boolean value indicating whether the object has the specified property as its own property.

Syntax: Reflect.has(target, property)

eg:
```
let person = {
    name : 'Tony';
    age : 21
};
console.log(Reflect.has(person, 'name')); // true.
```

Q.23 Describe execution flow of function constructor way of creating an object.

—> When you use the function creation to create an object, here's the execution flow:

i) First define a constructor function using the 'function' keyword, inside this (you) typically initialize properties using the 'this' keyword.

ii) Then create an instance of an object, you use the constructor function with the 'new' keyword.

iii) Now the constructor function is executed during this execution, properties are assigned to the instance using 'this', based on the arguments passed to the constructor function.

iv) Then the constructor function implicitly returns the newly created object.

v) Once the object is created, you can access

its properties and methods using dot notation or bracket notation. Properties initialized in the constructor function are directly accessible as instance properties.

Q.24 Describe problems associated with prototype chaining method of inheritance. Explain the solution for the same.

→ Prototype chaining is a method of inheritance where objects inherit properties and methods from their prototype chain. There are some problems associated with it :-

i) Shared State : Here, properties and methods are shared among all instances created from the same constructor functions. If one instance modifies a property that is shared through the prototype chain, it affects all other instances.

ii) Lack of Private Members : It does not provide built-in support for private members.

iii) Difficult in accessing super methods : There is no built-in way to access methods of the parent constructor directly from the child constructor.

→ Solution for these problems are :-

i) Constructor Functions : Use constructor functions

to define properties specific to each instance. Inside the constructor, use 'this' keyword to assign instance-specific properties.

ii) Closures for Private Members: use closures to create private members which involves defining private members inside the constructor function and accessing them through privileged methods.

iii) Super Methods: To access methods of the parent constructor, you can manually set up the prototype chain using 'Object.create()'.

Q.25 Explain getOwnPropertyDescriptor() method as reflect API.

-> The 'Reflect.getOwnPropertyDescriptor()' method is a part of the Reflect API; which provides a set of static methods for performing meta-programming tasks. It is used to retrieve the descriptor for a specified property of an object.

Syntax:- Reflect.getOwnPropertyDescriptor (target, propertyKey).

It returns an objects representing the property descriptor of the specified property or 'undefined' if the property

does not exist on the object.

eg:
```
let obj = {
    name : 'Tony',
    age : 21
};
let descriptor = Reflect.getOwnPropertyDescriptor
                                    (obj, 'name');
console.log(descriptor);
```

Output :-
```
{
    value : 'Tony',
    writable : true,
    enumerable : True,
    configurable : True
}
```

Q.26 Explain difference between function expression and function declaration.

Ans

Function Expression	Function Declaration
A function expression is similar to function declaration without the function name.	A function declaration must have a function name.
Assigned to a variable followed by an assignment operator.	Declared using the 'function' keyword followed by name.
Not hoisted. It must	HOISted. It can be

be defined before it is called.	called before it is declared in the code.
They are bound by the scope in which they are defined.	They generally have a broader scope due to hoisting.
used for callback functions, immediately invoked function expressions (IIFEs)	used for defining standalone functions, especially when structuring a program.
Syntax: var Demo = function (a,b) {...};	Syntax: function Demo(a,b) {...}

Q.27 Explain ways of passing function as an argument and return a function using Javascript.

→ Functions are first-class citizens, which means they can be passed as arguments to other functions and returned from functions. There are several ways to pass a function as an argument and return a function :-

i) Passing a Function as an Argument

You can pass a function as an argument to another function by simply providing the function name as argument.

eg:
```
function greet(name) {
    console.log('Hello, ' + name + '!');
}
function sayHello(callback) {
    callback('Tony');
}
sayHello(greet);  // Hello, Tony!
```

ii) Anonymous Function as an Argument.

Instead of passing a named function, you can pass an anonymous function directly as an argument.

eg:
```
function sayHello(callback) {
    callback('Tony');
}
sayHello(function(name) {
    console.log('Hello, ' + name + '!');
});  // Hello, Tony!
```

iii) Returning a Function.

You can return a function from another function by defining and returning the function within the enclosing function.

eg:
```
function createGreeter(greeting) {
    return function(name) {
        console.log(greeting + ', ' + name + '!');
    };
}
let greetHello = createGreeter('Hello');
greetHello('Tony');  // Hello, Tony!
```

Q.28 Explain promise method using a demonstrative Javascript.

—↓ Promises are a way to handle asynchronous operations. They represent a future value that may be available at some point or an error if the operation fails. It provides a cleaner and more flexible alternative to traditional callback-based approaches for handling asynchoro asynchronous code.

eg:

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeOut(() => {
            let data = { name: 'Tony', age: 21 };
            resolve(data);
        }, 2000);
    });
}
console.log('Fetching data...');
fetchData()
    .then(data => {
        console.log('Data fetched:', data);
    })
    .catch(error => {
        console.error('Error fetching', error.message);
    });
```

Output :- Fetching data...
Data fetched: { name: 'Tony', age: 21 }

Q.29 Explain arrow function using an example.

—4 Arrow functions (ES6), provide a more concise syntax for writing functions. They are particularly useful for writing shorter and more readable code, especially when working with callbacks, array methods and functional programming constructs.

eg: (1)
```
let addArrow = (a, b) => a + b;
console.log(addArrow(2,3)); // 5
```
(2)
```
let multiply = (a, b) => {
    let result = a * b;
    return result;
};
console.log(multiply(2,3)); // 6
```
(3)
```
let greet = () => {
    console.log('Hello!');
};
greet(); // Hello!
```

Q.30 Explain various function internals using a demonstrative Javascript.

—4 Function Internals refers to the inner workings of a function which encompasses several aspects like :-

i) Parameters and Arguments

Parameters are the variable listed as a part of the function definition.

Arguments are the actual values passed to the function when it's called.

eg:
```
function add(a, b) {
    return a + b;        → parameters
}                        → arguments
let result = add(3 + 5);
console.log(result);  // 8
```

ii) Closures

Closures allow functions to retain access to variables from their containing scope even after the parent function has finished executing.

eg:
```
function outer() {
    let outerVar = 'Outer';
    function inner() {
        console.log(outerVar);
    }
    return inner;
}
let innerFunc = outer();
innerFunc();  // Outer
```

iii) 'this' keyword.

'this' keyword refers to the object to which the function belongs or the object that is currently executing the function.

eg:
```
const person = {
    name: 'Tony',
    greet: function() {
```

```
        console.log('Hello, ${this.name}');
    }
};
person.greet(); // Hello, Tony
```

iv) Arrow Function

Arrow Functions are a concise way to write functions. They lexically bind 'this' value and are particularly useful for callback functions.

eg:
```
const squared = numbers.map(number =>
                    number * number);
console.log(squared);
```

Q.31 write a Javascript to implement promise chaining. Also explain all the functions used in it.

→ Promise chaining is a technique to execute multiple asynchronous operations in sequence, where each operations depends on the result of the previous one. This is achieved by chaining multiple '.then()' methods to a Promise.

eg:
```
function delay(ms) {
    return new Promise(resolve => setTimeOut(resolve, ms));
}
function sayHello() {
    return delay(1000).then(() => {
        return "Hello";
    });
}
```

```
function sayWorld() {
    return delay(1500).then(() => {
        return "World";
    });
}
sayHello()
    .then(message => {
        console.log(message); // Hello.
        return sayWorld();
    })
    .then(message => {
        console.log(message); // World.
    });
```

- 'delay(ms)': Returns a promise that resolves after given time.
- 'sayHello()': Returns a promise that resolves with the message "Hello" after 1000 ms.
- 'sayWorld()': Returns a promise that resolves with the message "World" after 1500 ms.

Q.32 Using Javascript explain call() and apply() methods.

—> 'call()' and 'apply()' are methods available on functions and they are used to invoke functions with a specified 'this' value and arguments.

i) 'call()' Method

This method is used call to invoke a

function with a specified 'this' value and individual arguments

eg:
```
function greet(name) {
    console.log('Hello, ' + name + '!');
}
greet.call(null, 'Tony'); // Hello, Tony!
```

iii) 'apply()' Method

This method is similar to 'call()', but it accepts arguments as an array or an array-like object.

eg:
```
function greet(name) {
    console.log('Hello, ' + name + '!');
}
greet.apply(null, ['Tony']); // Hello, Tony!
```

Q.33 Explain function closures using a Javascript example.

→ Function closure occur when an inner function has access to variables from its outer scope, even after the outer function has finished executing. This happens because the inner function maintains a reference to the variables in its lexical environment, allowing it to "close over" those variables.

eg:
```
function outer() {
    let outerVar = 'Outer';
    function inner() {
        console.log(outerVar);
    }
```

```
    return inner;
}
let innerFunc = outer();
innerFunc(); // Outer
```

Q.34 Describe Async function basics using JavaScript.

-> Async functions provide a cleaner and more concise syntax for writing asynchronous code using promises. They allow you to write asynchronous code that looks synchronous, making it easier to understand and maintain.

i) Declaration & Return Value.

Async functions are declared using the 'async' keyword and it always returns a promise, which resolves with the value returned by the functions.

eg:
```
async function myAsync() {
    return 'Hello';
}
myAsync().then(result => {
    console.log(result); // Hello.
});
```

ii) Await keyword.

It is used before a promise to pause the execution of the function until the promise settles and then resumes with

the resolved value.

eg:
```
async function myAsync() {
    let result = await someAsyncOperation();
    console.log(result);
}
```

iii) Error Handling

You can 'try... catch' blocks to handle errors in async functions just like synchronous code.

eg:
```
async function myAsync() {
    try {
        let result = await someAsyncOperation();
        console.log(result);
    } catch (error) {
        console.error("Error:", error);
    }
}
```

Q.35 Define Callback. Explain asynchronous programming using the same.

-4 A Callback function is a function that is passed an argument to another function and is invoked or executed after the completion of some asynchronous operation.

eg:
```
function fetchData(callback) {
    setTimeOut(() => {
        let data = {name: 'Tony', age: 21};
        callback(data);
    }, 2000);
}
```

```
function processData(data) {
    console.log('Received data:', data);
}
fetchData(processData);
```

—↳ Asynchronous programming using callbacks allows to execute non-blocking code, meaning that other operations can continue while waiting for asynchronous tasks to complete. This is essent essential for tasks that ~~to~~ involve waiting for I/O operations or tasks that require a delay.

Q.36 Discuss default parameter passing and spread arguments to a function using a Javascript.

—↳ i) Default Parameter Passing.

It allows you to specify default values for function parameters in case no value or 'undefined' is provided when the function is called.

eg:
```
function greet(name = 'Guest') {
    console.log('Hello, ' + name + '!');
}
greet();          // Hello, Guest!
greet('Tony');    // Hello, Tony!
```

ii) Spread Arguments Passing.

It allows an iterable to be

expanded ~~to~~ into individual elements. It can be used to pass an array as multiple arguments to a function.

eg:
```
function add(a,b,c) {
    return a+b+c;
}
let numbers = [1,2,3];
console.log(add(...numbers)); // 6
```

Q.37 Define asynchronous programming and explain different ways of performing the same.

→ Asynchronous Programming is a programming paradigm ~~to~~ used to handle tasks that might take some time to complete, such as fetching data from a server, reading files or waiting for user input. Here, operations are initiated and then the program continues to execute other tasks without waiting for the initiated operations to complete. Once it is completed, a callback function is invoked to handle the result of the operation.

i) Callbacks: They are passed as ~~an~~ arguments to other functions and are invoked once an asynchronous operation completes.

ii) Promises: They represent the eventual completion or failure of an asynchronous

operation. They also allow you handle it in a more structured and manageable way.

iii) Async / Await : It provides a more concise and readable syntax for working with promises. (ECMAScript 2017 - ES8).

iv) Timers : They are used to schedule tasks to be executed after a certain delay or at regular intervals.

Q.38 Describe function properties and methods using a Javascript.

→ Functions are first-class citizens, meaning they can be treated as values, assigned to variables, passed an arguments to other functions and returned from functions.

- Function Properties :-

i) name : returns the name of the function as a string.

eg:
```
function greet() {...}
console.log(greet.name) // greet.
```

ii) length : returns the number of parameters expected by the function

eg:
```
function add(a,b) {...}
console.log(add.length); // 2.
```

- Function Methods :-

i) call (thisArg, arg1, arg2, ...) : calls a function with a specified 'this' value and ~~more~~ individual arguments.

eg:
```
function greet(name) {
    console.log('Hello, ' + name + '!');
}
greet.call(null, 'Tony'); // Hello, Tony!
```

ii) apply (thisArg, [argsArray]): calls a function with a specified 'this' value and arguments provided as an array.

eg:
```
function add(a, b) {
    return a+b;
}
console.log(add.apply(null, [2,3])); // 5
```

iii) toString() : returns a string representing the source code of the function.

eg:
```
function greet() {
    console.log('Hello');
}
console.log(greet.toString());
```

* Other methods are bind(), valueOf().

Rohini