1. Closures:

A closure is a combination of a function and the lexical environment in which it was declared. It allows a function to access variables from its outer function scope even after the outer function has finished executing. This is achieved by creating a reference to the outer function's variables and keeping it alive in memory.

Example:

```
function outer() {
  var outerVar = 'I am from outer';

  function inner() {
    console.log(outerVar);
  }

  return inner;
}

var closureFn = outer(); // Assign the returned inner function to a variable
closureFn(); // Output: 'I am from outer'
```

In the example above, the inner function `inner()` has access to the `outerVar` variable, even though `outer()` has finished executing. The closure preserves the environment of `outer()` and allows `inner()` to access `outerVar`.

2. OOPs Paradigm vs. Functional Programming:

OOPs (Object-Oriented Programming) and Functional Programming are two different programming paradigms.

OOPs focuses on organizing code into objects that encapsulate data and behavior. It uses concepts like classes, objects, inheritance, and polymorphism. In JavaScript, OOPs can be implemented using constructor functions, classes (ES6), and prototypes.

Example of OOPs in JavaScript:

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

```
  speak() {
    console.log(this.name + ' makes a sound.');
  }
}

const cat = new Animal('Cat');
cat.speak(); // Output: 'Cat makes a sound.'
```

Functional Programming, on the other hand, emphasizes the use of pure functions and immutable data. It avoids changing state and mutable data. Functions are treated as first-class citizens and can be passed around as arguments or returned as values.

Example of Functional Programming in JavaScript:

```
const numbers = [1, 2, 3, 4, 5];

const doubleNumbers = numbers.map(num => num * 2);
console.log(doubleNumbers); // Output: [2, 4, 6, 8, 10]
```

Functional programming promotes writing pure functions that don't have side effects and can be easily composed to solve complex problems.

3. Imperative and Declarative Code Writing:

Imperative programming focuses on describing the specific steps or procedures to perform a task. It explicitly defines how to achieve a result by specifying the control flow and manipulating mutable state.

Example of Imperative Code:

```
const numbers = [1, 2, 3, 4, 5];
let sum = 0;

for (let i = 0; i < numbers.length; i++) {
  sum += numbers[i];
}

console.log(sum); // Output: 15
```

Declarative programming, on the other hand, focuses on describing the desired result without specifying the control flow or mutable state manipulation. It emphasizes what should be achieved rather than how to achieve it.

Example of Declarative Code:

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

console.log(sum); // Output: 15
```

In declarative programming, the `reduce()` method is used to declare the desired result (summing the numbers) without specifying the step-by-step process.

4. Mutability and Immutability:

Mutability refers to the ability to change or modify an object or data after it is created. In JavaScript, objects (including arrays) and functions are mutable by default.

Example of Mutable Object:

```
const person = {
  name

: 'John',
  age: 30,
};

person.age = 31;
console.log(person); // Output: { name: 'John', age: 31 }
```

Immutability, on the other hand, refers to the inability to change an object or data after it is created. Immutable data cannot be modified once it is created. In JavaScript, primitive data types (like strings and numbers) are immutable.

Example of Immutable String:

```
let name = 'John';
name = name.concat(' Doe');
console.log(name); // Output: 'John Doe'
```

Immutability provides benefits such as easier debugging, predictable code, and better performance in certain scenarios. To achieve immutability with objects and arrays, techniques like creating copies, using immutability libraries (e.g., Immutable.js), or leveraging newer language features like the spread operator or `Object.assign()` can be used.

5. Polyfills:

Polyfills are code snippets or scripts that provide modern functionality in older or less capable browsers that do not natively support those features. They "fill" the gap by replicating the behavior of newer features using JavaScript code.

Polyfills are commonly used to ensure cross-browser compatibility and allow developers to write code using the latest JavaScript features without worrying about unsupported environments.

Example of a Polyfill for the `Array.prototype.includes()` method:

```
if (!Array.prototype.includes) {
  Array.prototype.includes = function(element) {
    for (var i = 0; i < this.length; i++) {
      if (this[i] === element) {
        return true;
      }
    }
    return false;
  };
}
```

- The first line checks if the `includes` method does not already exist on the `Array.prototype`. If it doesn't exist, it means the current environment doesn't support it.
- Inside the if statement, we define the `includes` method on the `Array.prototype`. This allows all arrays to have access to this method.
- The `includes` method takes an `element` parameter and checks if the array contains that element.
- We loop through each element of the array using a for loop. If we find a match, we return `true`.
- If the loop completes without finding a match, we return `false` to indicate that the element is not present in the array.

6. Prototypes of Map, Filter, and Reduce:

In JavaScript, the `map()`, `filter()`, and `reduce()` methods are built-in array methods used for manipulating and transforming arrays.

- `map()` creates a new array by applying a callback function to each element of the original array.
- `filter()` creates a new array by filtering out elements based on a condition specified in the callback function.
- `reduce()` applies a callback function to each element of the array, accumulating a single result.

Example usage of `map()`, `filter()`, and `reduce()`:

```
const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map(num => num * 2);
console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]

const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // Output: [2, 4]

const sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // Output: 15
```

These methods provide powerful ways to transform, filter, and aggregate data within arrays.

7. Polyfills of Map, Filter, and Reduce:

Since the `map()`, `filter()`, and `reduce()` methods are built-in array methods in JavaScript, they are available in modern browsers and environments that support ECMAScript 5 (ES5) or later. Therefore, they usually do not require polyfills.

Polyfills are typically needed for older browsers that lack support for these methods. However, as of 2021, most widely used browsers support these methods, so polyfills are less commonly used for them. It is still recommended to check the browser compatibility requirements of your target audience before deciding whether to use polyfills.

8. Strict Mode and Non-Strict Mode:

Strict mode is a feature introduced in ECMAScript 5 (ES5) that enforces stricter parsing and error handling in JavaScript. It helps to prevent common mistakes, enhance security, and optimize JavaScript code.

Strict mode can be enabled for an entire script or a specific function scope by adding the following line at the beginning:

```
'use strict';
```

Enabling strict mode affects several aspects of JavaScript behavior, including the following:

- Variables must be declared before use (no implicit global variables).
- Assigning a value to an undeclared variable throws an error.
- Deleting variables, functions, or function arguments is not allowed.
- Duplicate parameter names in function declarations throw an error.
- `this` is undefined in functions that are not methods or constructors.
- Octal literals and escape characters are not allowed, among other changes.

Non-strict mode, or "sloppy mode," is the default behavior in JavaScript when strict mode is not explicitly enabled. In non-strict mode, JavaScript code may exhibit more lenient and potentially error-prone behavior.

9. The `this` Keyword in JavaScript:

In JavaScript, the `this` keyword refers to the object that is currently executing the code or the object that a method is called on. The value of `this` depends on the context in which it is used.

The binding of `this` can be determined by the following rules:

- Global Scope: In the global scope (outside of any function), `this` refers to the global object, which is `window` in browsers or `global` in Node.js.
- Function Invocation: When a function is invoked as a standalone function (not as a method or constructor), `this` refers to the global object (`window` or `global`). In strict mode, `this` will be `undefined`.
- Method Invocation: When a function is called as a method of an object, `this` refers to the object on which the method is called.
- Constructor Invocation: When a function is used as a constructor with the `new` keyword, `this` refers to the newly created instance of the object.
- Explicit Binding: The `call()` and `apply()` methods allow explicit binding of `this` to a specific object.
- Arrow Functions: Arrow functions do not bind their own `this` but inherit it from the surrounding scope lexically.

Understanding the context and scope in which a function is called is crucial for correctly determining the value of `this`.

10. Function Currying:

Function currying is a technique used to transform a function with multiple arguments into a sequence of functions, each taking a single argument. It allows partial application of arguments and the creation of more specialized functions.

Example of Function Currying:

```
function add(a) {
  return function (b) {
    return a + b;
  };
}

const add5 = add(5);
console.log(add5(3)); // Output: 8
```

In the example above, the `add()` function takes the first argument `a` and returns an inner function that takes the second argument `b`. This allows us to create a specialized `add5` function that adds `5` to any given number.

Function currying can be useful for creating reusable and composable functions, as well as for creating functions with preset or default arguments.

11. Method Borrowing:

Method borrowing is a technique used in JavaScript to reuse methods from one object in another object. It allows an object to use a method defined in another object's prototype or property.

Example of Method Borrowing:

```
const person = {
  name: 'John',
  introduce: function () {
    console.log('Hi, my name is ' + this.name);
  },
};

const student = {
  name

: 'Jane',
};
```

```
person.introduce.call(student); // Output: 'Hi, my name is Jane'
```

In the example above, the `person` object has an `introduce()` method. By using the `call()` method, we can borrow and invoke the `introduce()` method with the `student` object as the context (`this`). This allows the `student` object to use the `introduce()` method defined in the `person` object.

12. Constructor Function and "this" Keyword:

In JavaScript, constructor functions are used to create and initialize objects. They are typically defined with an initial capital letter and are called using the `new` keyword.

Inside a constructor function, the `this` keyword refers to the newly created object. It allows assigning values to properties and methods of the object being created.

Example of Constructor Function and "this":

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

const john = new Person('John', 30);
console.log(john.name); // Output: 'John'
console.log(john.age); // Output: 30
```

In the example above, the `Person` constructor function takes `name` and `age` as parameters and assigns them to the `name` and `age` properties of the newly created object (`this`).

Constructor functions are often used with the `new` keyword to create multiple instances of objects with similar properties and behaviors.

13. Prototype:

In JavaScript, every object has a prototype, which is a reference to another object. The prototype is used for inheritance and allows objects to inherit properties and methods from their prototype.

Prototype-based inheritance means that an object can inherit properties and behaviors from another object, known as its prototype. Objects in JavaScript are linked to a prototype object through the `prototype` property.

Example of Prototype:

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

Animal.prototype.speak = function () {
  console.log(this.name + ' makes a sound.');
};

const cat = new Animal('Cat');
cat.speak(); // Output: 'Cat makes a sound.'
```

In the example above, the `speak()` method is added to the `Animal.prototype`, which is the prototype of the `Animal` constructor function. The `cat` object created using `new Animal('Cat')` inherits the `speak()` method from its prototype and can invoke it.

Prototypes allow for efficient memory usage by sharing common properties and methods among objects.

14. Prototypal Inheritance:

Prototypal inheritance is a way of creating objects based on existing objects. In JavaScript, objects can inherit properties and behaviors from their prototypes.

When an object is accessed for a property or method that it doesn't have, JavaScript looks up the prototype chain until it finds the property or reaches the end of the chain (usually the `Object.prototype`).

Example of Prototypal Inheritance:

```
function Animal() {}

Animal.prototype.eat = function () {
  console.log('The animal is eating.');
};

function Dog() {}

Dog.prototype = Object.create(Animal.prototype); // Set Dog's prototype to Animal's prototype
```

```
Dog.prototype.constructor = Dog; // Reset the constructor property

Dog.prototype.bark = function () {
  console.log('The dog is barking.');
};

const dog = new Dog();
dog.eat(); // Output: 'The animal is eating.'
dog.bark(); // Output: 'The dog is barking.'
```

In the example above, the `Animal` function acts as the parent constructor, and the `Dog` function is the child constructor. By setting `Dog.prototype` to `Object.create(Animal.prototype)`, the `Dog` object inherits the `eat

()` method from the `Animal` object.

Prototypal inheritance allows objects to share and extend functionality, providing a powerful mechanism for code reuse and abstraction.

15. Creating Objects Using Classes:

In ECMAScript 6 (ES6) and later versions, JavaScript introduced the `class` syntax, which provides a more convenient way to create objects and implement object-oriented programming concepts.

The `class` keyword allows defining a template for creating objects, called a "class." Objects created from a class are known as "instances" or "objects."

Example of Creating Objects Using Classes:

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

  speak() {
    console.log(this.name + ' makes a sound.');
  }
}

const cat = new Animal('Cat');
cat.speak(); // Output: 'Cat makes a sound.'
```

In the example above, the `Animal` class is defined using the `class` keyword. The `constructor` method is used to initialize the object's properties. Other methods, such as `speak()`, can be added directly inside the class.

The `new` keyword is used to create instances of the class. Objects created from a class have their own unique state but share the same methods defined in the class.

16. Differences between null, undefined, and not defined in JavaScript:

- `null` is a value that represents the intentional absence of any object value. It is a primitive value assigned to a variable to indicate the absence of an object or empty value.

Example:

```
let myVariable = null;
console.log(myVariable); // Output: null
```

- `undefined` is a value that represents the absence of a defined value. It is the default value assigned to a variable that has been declared but has not been assigned a value.

Example:

```
let myVariable;
console.log(myVariable); // Output: undefined
```

- "Not defined" refers to a variable or identifier that has not been declared or defined in the current scope. It is an error to use a variable that has not been declared.

Example:

```
console.log(myVariable); // Output: ReferenceError: myVariable is not defined
```

In summary, `null` is a value that represents the intentional absence of an object, `undefined` is the default value for uninitialized variables, and "not defined" refers to a variable that has not been declared or defined.

17. Truthy and Falsy Values:

In JavaScript, every value has an inherent boolean truthiness or falsiness. A value that is considered "truthy" evaluates to `true` in a boolean context, while a value that is considered "falsy" evaluates to `false`.

The following values are considered falsy:
- `false`
- `0` (zero)
- `''` (empty string)
- `null`
- `undefined`
- `NaN` (Not a Number)

All other values, including non-empty strings, numbers other than zero, and objects, are considered truthy.

Example:

```
if ('Hello') {
  console.log('Truthy');
} else {
  console.log('Falsy');
}
```

In the example above, the string `'Hello'` is considered a truthy value, so the output will be 'Truthy'.

Truthy and falsy values are often used in conditional statements and can be leveraged to write concise and expressive code.

18. Operators in JavaScript:

JavaScript provides various operators that perform operations on values or variables. Operators can be classified into several categories:

- Arithmetic Operators: Perform basic arithmetic operations such as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and more.
- Assignment Operators: Assign values to variables

. For example, the `=` operator assigns a value to a variable.
- Comparison Operators: Compare two values and return a boolean result. Examples include `==` (equality), `!=` (inequality), `>` (greater than), `<` (less than), etc.
- Logical Operators: Combine multiple conditions and return a boolean result. Examples include `&&` (logical AND), `||` (logical OR), and `!` (logical NOT).

- Bitwise Operators: Perform operations at the bit level. Examples include `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise XOR), and more.
- Unary Operators: Operate on a single value. Examples include `++` (increment), `--` (decrement), `typeof` (returns the type of a value), and more.
- Ternary Operator: Also known as the conditional operator (`? :`). It allows for a compact way of writing conditional expressions.

These are just a few examples of the many operators available in JavaScript. Operators play a crucial role in performing computations, making decisions, and manipulating data.

19. Comparing Operators with Truthy and Falsy Values:

When using comparison operators (`==` and `===`) in JavaScript, the truthy and falsy nature of values comes into play.

The `==` operator performs type coercion, meaning it attempts to convert values of different types before comparing them. On the other hand, the `===` operator performs strict equality comparison without type coercion.

Example:

```
console.log(1 == true); // Output: true
console.log(1 === true); // Output: false
```

In the first comparison, `1` and `true` are considered equal because the `==` operator performs type coercion, converting `true` to `1` before comparison.

In the second comparison, `1` and `true` are not considered equal because the `===` operator performs strict equality comparison and does not perform type coercion.

It is generally recommended to use strict equality (`===`) to avoid unexpected behavior and ensure accurate comparisons.

20. Shallow Copy:

Shallow copy is a method of creating a new object or array that duplicates the top-level structure of the original object or array. However, the nested objects or arrays inside the original object are still shared between the original and the copied object.

Example of Shallow Copy:

```
const originalArray = [1, 2, [3, 4]];
const shallowCopy = [...originalArray];
```

```
originalArray[0] = 'modified';
originalArray[2][0] = 'modified';

console.log(originalArray); // Output: ['modified', 2, ['modified', 4]]
console.log(shallowCopy); // Output: [1, 2, ['modified', 4]]
```

In the example above, the `shallowCopy` is created using the spread operator (`...`). Modifying the elements in the original array (`originalArray`) affects the corresponding elements in the shallow copy. However, modifying nested arrays still affects both the original and the shallow copy.

Shallow copy is useful when you need to create a copy of an object or array without modifying the original, but be aware that changes to nested objects or arrays will be reflected in both the original and the shallow copy.

21. Deep Copy:

Deep copy is a method of creating a new object or array that duplicates both the top-level structure and the nested objects or arrays inside the original object or array. It creates a completely independent copy of the original data.

Example of Deep Copy:

```
const originalArray = [1, 2, [3, 4]];
const deepCopy = JSON.parse(JSON.stringify(originalArray));

originalArray[

0] = 'modified';
originalArray[2][0] = 'modified';

console.log(originalArray); // Output: ['modified', 2, ['modified', 4]]
console.log(deepCopy); // Output: [1, 2, [3, 4]]
```

In the example above, the `deepCopy` is created by serializing the original array to JSON (`JSON.stringify()`) and then parsing it back to an object (`JSON.parse()`). This process creates a new independent copy that is not affected by modifications to the original array or its nested arrays.

Deep copy is useful when you need to create a completely independent copy of an object or array, including all nested objects or arrays.

22. Flattening an Object:

Flattening an object refers to converting a nested object into a single-level object, where all the properties are at the top level.

Example of Flattening an Object:

```
function flattenObject(obj, prefix = '') {
  const flattened = {};

  for (const key in obj) {
    if (obj.hasOwnProperty(key)) {
      const propName = prefix ? `${prefix}.${key}` : key;

      if (typeof obj[key] === 'object' && obj[key] !== null) {
        const nestedObj = flattenObject(obj[key], propName);
        Object.assign(flattened, nestedObj);
      } else {
        flattened[propName] = obj[key];
      }
    }
  }

  return flattened;
}

const nestedObject = {
  user: {
    name: 'John',
    address: {
      street: '123 Main St',
      city: 'New York',
    },
  },
  age: 30,
};

const flattenedObject = flattenObject(nestedObject);

console.log(flattenedObject);
```

In the example above, the `flattenObject()` function takes a nested object as input and recursively flattens it. The resulting flattened object contains all the properties of the original object at the top level, using dot notation to represent nested properties.

Flattening an object can be useful in scenarios where you need to work with a simplified representation of the data or when you need to serialize the object for storage or transmission purposes.

Prepared By :
Divyansh Singh (LinkedIn : rgndunes)