Kulsoom

WD-II

2-4

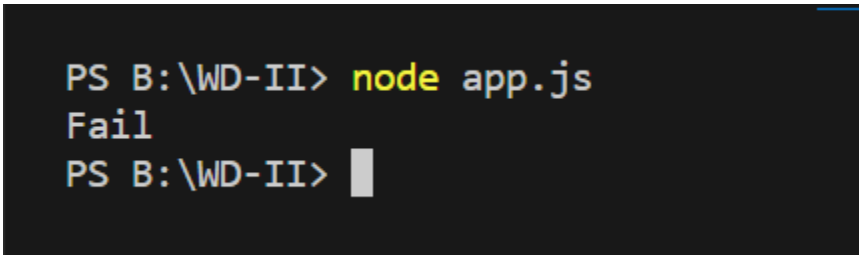
Assignment 3

1. Rewrite the following code using a ternary operator:

```
let result;
if (score >= 80) {
   result = "Pass";
} else {
   result = "Fail";
}
```

```
let score = 40;
let result = score >= 80 ? "Pass" : "Fail";
console.log(result);
```

Output:



2. How does the optional chaining operator (?.) work, and how can it be used to access nested properties of an object?

Optional Chaining Operator (?.):

The optional chaining operator (?.) simplifies the process of accessing nested properties in JavaScript, especially when dealing with the possibility of encountering null or undefined values. Instead of writing multiple conditional checks, you can use the optional chaining operator to safely navigate through nested properties, shortcircuiting the evaluation if any property is null or undefined.

In simpler terms, it allows you to access nested properties without worrying about whether each level of the hierarchy exists, making your code cleaner and more concise.

```
const person = {
  name: "Kulsoom",
  age:21,
  address: {
    city: "Karachi",
```

```
    postalCode: 75800
  }
};

// Without optional chaining
const cityName = person.address ? person.address.country : "Unknown";

// With optional chaining
const cityName1 = person.address?.country || "Unknown";

console.log("Without Optional Chaining:", cityName);
console.log("With Optional Chaining:", cityName1);
```

Output:

```
PS B:\WD-II> node app.js
Without Optional Chaining: undefined
With Optional Chaining: Unknown
PS B:\WD-II>
```

3. Compare the for...in loop and the for...of loop in terms of their use cases and the types of values they iterate over.

The for...in loop is used to iterate over the enumerable properties of an object, while the for...of loop is used to iterate over iterable objects like arrays, strings, maps, etc. Use for...in for objects and for...of for values within iterable objects.

```
//for in
const person = {
  name: "Kulsoom",
  age: 21,
  city: "Karachi"
};

console.log("Properties of the person object:");

for (let key in person) {
  console.log(${key}: ${person[key]});
}
```

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

console.log("Values in the numbers array:");

for (let num of numbers) {
  console.log(num);
}
```

Output:

```
PS B:\WD-II> node app.js
Properties of the person object:
name: Kulsoom
age: 21
city: Karachi
Values in the numbers array:
1
2
3
4
5
PS B:\WD-II>
```

4. Define a function calculateAverage that takes an array of numbers as an argument and returns the average value.

```
function calculateAverage(numbers) {

  // Check if the array is not empty
  if (numbers.length === 0) {
    return 0; // Return 0 for an empty array to avoid division by zero
  }

  // Calculate the sum of all numbers in the array
  const sum = numbers.reduce((accumulator, currentNumber) => accumulator +
currentNumber, 0);

  // Calculate the average by dividing the sum by the number of elements
  const average = sum / numbers.length;
```

```
  return average;
}

// Example usage:
const numbersArray = [10, 20, 30, 40, 50];
const averageValue = calculateAverage(numbersArray);

console.log("Average:", averageValue);
```

Output:

```
PS B:\WD-II> node app.js
Average: 30
PS B:\WD-II>
```

5. Explain the concept of "closures" in JavaScript and provide an example of their practical use.

In JavaScript, a closure is created when a function is defined within another function, allowing the inner function to access the outer function's variables and parameters. Closures have access to the outer function's scope even after the outer function has finished executing. This behavior provides a way to create private variables and encapsulate functionality.

```
function outerFunction(x) {
  // innerFunction is a closure because it remembers 'x'
  function innerFunction(y) {
    return x + y;
  }

  return innerFunction;
}

// Create a closure by calling outerFunction
const closure = outerFunction(10);

// Use the closure
const result = closure(5);
console.log(result); // Output: 15 (10 + 5)
```

1. outerFunction takes a parameter x.

2. Inside outerFunction, innerFunction is defined and can access x.

3. outerFunction returns innerFunction.

4. When outerFunction(10) is called, it creates a closure where x is remembered as 10.

5. The returned function (innerFunction) is now closure.

6. When closure(5) is called, it uses the remembered x (which is 10) and adds 5 to it.

So, a closure is like a little "memory" of the environment in which it was created.

Output:

```
PS B:\WD-II> node app.js
15
PS B:\WD-II>
```

6. Create an object named student with properties name, age, and grades. Add a method calculateAverage that calculates the average of the grades.

```
const student = {
  name: "Kulsoom",
  age: 21,
  grades: [85, 90, 88, 92, 87],

  calculateAverage: function () {
    const totalGrades = this.grades.reduce((sum, grade) => sum + grade, 0);
    const average = totalGrades / this.grades.length;
    return average;
  }
};

// Example usage:
console.log("Student Name:", student.name);
console.log("Student Age:", student.age);
console.log("Grades:", student.grades);
console.log("Average Grade:", student.calculateAverage());
```

Output:

```
PS B:\WD-II> node app.js
Student Name: Kulsoom
Student Age: 21
Grades: [ 85, 90, 88, 92, 87 ]
Average Grade: 88.4
PS B:\WD-II>
```

7. How can you clone an object in JavaScript and also give one example each deep copy, shallow copy, and reference copy

Shallow Copy:

A shallow copy creates a new object and copies the toplevel properties of the original object. However, if the original object contains nested objects, those nested objects are still references, not copies.

```
// Original object
const originalObject = { name: "John", age: 25, hobbies: ["reading", "coding"] };

// Shallow copy using Object.assign()
const shallowCopy = Object.assign({}, originalObject);

// Modify the shallow copy
shallowCopy.age = 26;
shallowCopy.hobbies.push("gaming");

console.log("Original Object:", originalObject);
console.log("Shallow Copy:", shallowCopy);
```

Output:

```
PS B:\WD-II> node app.js
Original Object: { name: 'John', age: 25, hobbies: [ 'reading', 'coding', 'gaming' ] }
Shallow Copy: { name: 'John', age: 26, hobbies: [ 'reading', 'coding', 'gaming' ] }
Student Name: Kulsoom
Student Age: 21
Grades: [ 85, 90, 88, 92, 87 ]
Average Grade: 88.4
PS B:\WD-II> cd -app.js
```

In this example, modifying the shallowCopy object's properties does not affect the originalObject, but modifying the nested array (hobbies) does affect both objects.

2. Deep Copy:

A deep copy creates a new object and recursively copies all properties, including nested objects. Changes to the copied object or its nested objects do not affect the original object.

```
// Original object with nested objects
const originalObject = { name: "John", age: 25, address: { city: "New York", zip:
10001 } };

// Deep copy using JSON.parse(JSON.stringify())
const deepCopy = JSON.parse(JSON.stringify(originalObject));

// Modify the deep copy
deepCopy.age = 26;
deepCopy.address.city = "Los Angeles";

console.log("Original Object:", originalObject);
console.log("Deep Copy:", deepCopy);
```

Output:

```
PS B:\WD-II> node app.js
Original Object: { name: 'John', age: 25, address: { city: 'New York', zip: 10001 } }
Deep Copy: { name: 'John', age: 26, address: { city: 'Los Angeles', zip: 10001 } }
PS B:\WD-II>
```

In this example, modifying the deepCopy object does not affect the originalObject, even for nested objects.

3. Reference Copy:

A reference copy doesn't actually create a new object; it creates a new reference to the same object. Changes made to the properties of one reference will affect the other.

```
// Original object
const originalObject = { name: "John", age: 25 };
```

```
// Reference copy
const referenceCopy = originalObject;

// Modify the reference copy
referenceCopy.age = 26;

console.log("Original Object:", originalObject);
console.log("Reference Copy:", referenceCopy);
```

Output:

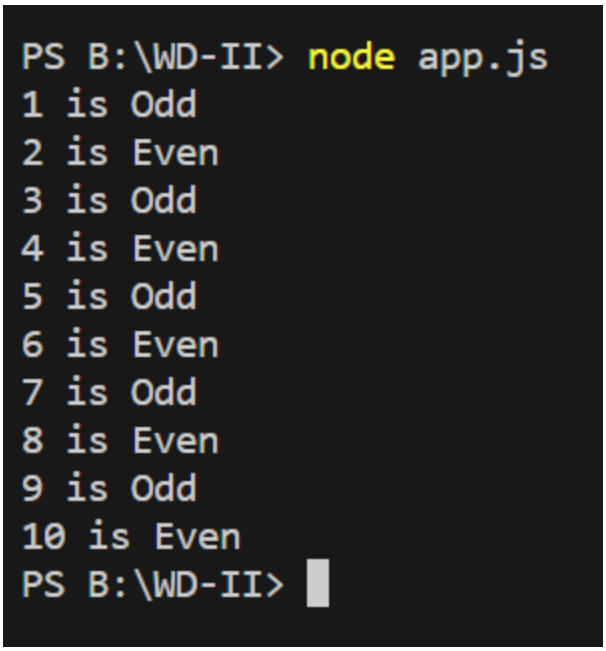
```
PS B:\WD-II> node app.js
Original Object: { name: 'John', age: 26 }
Reference Copy: { name: 'John', age: 26 }
PS B:\WD-II>
```

8. Write a loop that iterates over an array of numbers and logs whether each number is even or odd, using a ternary operator.

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

for (let number of numbers) {
  const message = number % 2 === 0 ? "Even" : "Odd";
  console.log(${number} is ${message});
}
```

Output:



9. Describe the differences between the for loop, while loop, and do...while loop in JavaScript. When might you use each?

1. for Loop:

```
for (initialization; condition; iteration) {
  // code to be executed
}
```

Use Cases:

When you know the exact number of iterations in advance.

Convenient for iterating over elements in an array.

Well suited for situations where the loop control variables need initialization, condition checking, and iteration steps.

Example:

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

2. while Loop:

```
while (condition) {
  // code to be executed
}
```

Use Cases:

When the number of iterations is not known in advance and depends on a condition.

Useful for situations where you want to loop as long as a certain condition is true.

Example:

```
let count = 0;
while (count < 5) {
  console.log(count);
  count++;
}
```

3. do...while Loop:

Structure:

```
do {
  // code to be executed
} while (condition);
```

Use Cases:

When you want to guarantee that the loop body is executed at least once.

Similar to while but with the condition checked after the execution of the loop body.

Example:

```
let x = 0;
do {
  console.log(x);
```

```
  x++;
} while (x < 5);
```

10. Provide an example of using optional chaining within a loop to access a potentially missing property of an object.

```
const students = [
  { name: "Kulsoo", age: 21 },
  { name: "Safia", age: 22, grades: { math: 90, english: 85 } },
  { name: "Nida", age: 21 }
];

for (let student of students) {
  // Using optional chaining to access the 'grades' property, which might be
missing
  const mathGrade = student?.grades?.math;

  // Logging the result
  console.log(${student.name}'s math grade: ${mathGrade !== undefined ? mathGrade
: 'Not available'});
}
```

Output:

```
PS B:\WD-II> node app.js
Kulsoo's math grade: Not available
Safia's math grade: 90
Nida's math grade: Not available
PS B:\WD-II>
```

11. Write a for...in loop that iterates over the properties of an object and logs each property name and value.

```
const car = {
  make: "Toyota",
  model: "Camry",
```

```
  year: 2022,
  color: "Silver"
};

// Iterate over the properties of the 'car' object
for (let property in car) {
  if (car.hasOwnProperty(property)) {
    console.log(Property: ${property}, Value: ${car[property]});
  }
}
```

Output:

```
PS B:\WD-II> node app.js
Property: make, Value: Toyota
Property: model, Value: Camry
Property: year, Value: 2022
Property: color, Value: Silver
PS B:\WD-II>
```

12. Explain the use of the break and continue statements within loops. Provide scenarios where each might be used.

Scenarios:

break:

Exiting a loop when a certain condition is met.

Searching for an element in an array and breaking out of the loop once found.

```
//break
for (let i = 0; i < 7; i++) {
  console.log(i);
  if (i === 5) {
    console.log("Number found");
    break; // Exit the loop when i is equal to 5
  }

}
```

Output:

```
PS B:\WD-II> node app.js
0
1
2
3
4
5
Number found
PS B:\WD-II>
```

continue:

Skipping iterations based on a condition.

Processing only odd or even numbers in a loop.

```
for (let i = 0; i < 5; i++) {
  if (i === 2) {
    console.log("Number 2 has skipped")
    continue; // Skip the iteration when i is equal to 2
  }
  console.log(i);
}
```

Output:

```
PS B:\WD-II> node app.js
0
1
Number 2 has skipped
3
4
PS B:\WD-II>
```

13. Write a function calculateTax that calculates and returns the tax amount based on a given income. Use a ternary operator to determine the tax rate.

```
function calculateTax(income) {
  const taxRate = income > 50000 ? 0.2 : 0.1;
  const taxAmount = income  taxRate;
  return taxAmount;
}

// Example usage:
const income1 = 45000;
const income2 = 60000;

console.log("Tax for income1:", calculateTax(income1)); // Output: 4500 (10% of
45000)
console.log("Tax for income2:", calculateTax(income2)); // Output: 12000 (20% of
60000)
```

Output:

```
PS B:\WD-II> node app.js
Tax for income1: 4500
Tax for income2: 12000
PS B:\WD-II>
```

14. Create an object car with properties make, model, and a method startEngine that logs a message. Instantiate the object and call the method.

```
const car = {
  make: "Toyota",
  model: "Camry",
  startEngine: function () {
    console.log("Engine started!");
  }
};
car.startEngine();
```

Output:

```
PS B:\WD-II> node app.js
Engine started!
PS B:\WD-II>
```

15. Explain the differences between regular functions and arrow functions in terms of scope, this binding, and their use as methods.

1. Scope:

Regular Functions: Create their own scope.

Arrow Functions: Share the scope of where they are defined.

2. this Binding:

Regular Functions: Have their own this binding.

Arrow Functions: Inherit this from the surrounding code.

3. Arguments Object:

Regular Functions: Have the arguments object.

Arrow Functions: Use rest parameters (...args) instead.

4. Use as Methods:

Regular Functions: Good for object methods with their own this.

Arrow Functions: Not suitable for object methods; they inherit this.

5. Binding in Events:

Regular Functions: Useful in event handlers; this is the element that triggered the event.

Arrow Functions: Not ideal for events; this is lexically scoped and might not be what you expect.

In simpler terms, regular functions and arrow functions behave differently regarding this and scope. Regular functions are versatile and often better for objectoriented patterns, while arrow functions are concise but have limitations, especially when it comes to this binding.