Bano Qabil 2.0

Course: Web – 3 (8 – 10) M

Assignment # 06

Question # 01: Rewrite the following code using a ternary operator:

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

Answer: Here's the code rewritten using a ternary operator:

```
let score = 85;
let result = (score >= 80) ? "Pass" : "Fail";
console.log(result);
```

Question # 02: How does the optional chaining operator (?.) work, and how can it be used to access nested properties of an object?

Answer: The optional chaining operator (?.) is a feature introduced in ECMAScript 2020 (ES11) that simplifies the process of accessing nested properties of an object, especially when dealing with potentially undefined values. It helps avoid the need for explicit null or undefined checks before accessing nested properties, making the code more concise and less prone to errors.

1. Accessing nested properties:

```
// Without optional chaining
const Name = user && user.address && user.address.city;

// With optional chaining
const name = user?.address?.city;
```

2. Calling functions:

```
// Without optional chaining
const Length = str && str.length;

// With optional chaining
const length = str?.length;
```

3. Array access:

```
// Without optional chaining
const FirstElement = arr && arr[0];

// With optional chaining
const firstElement = arr?.[0];
```

4. Function calls:

```
// Without optional chaining
const Result = obj && obj.method && obj.method();

// With optional chaining
const result = obj?.method?.();
```

Question # 03: Compare the for...in loop and the for...of loop in terms of their use cases and the types of values they iterate over.

Answer:

1. For…in Loop:

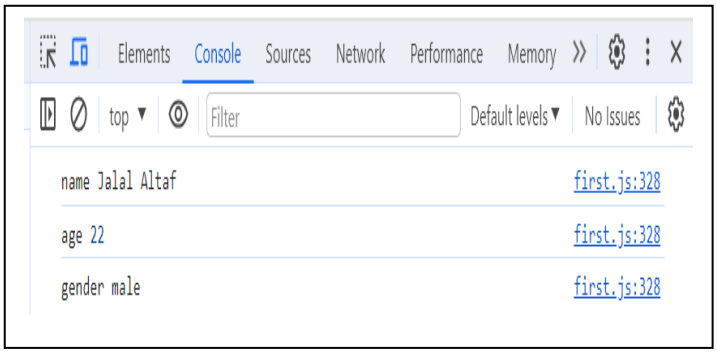
- **Use cases:**
  - ➢ Used to iterate over the enumerable properties of an object.
  - ➢ Typically used with objects to loop through keys or properties.

- **Syntax:**

```
for (variable in object) {
    // code to be executed
}
```

- **Example:**

```
const person = {
    name: 'Jalal Altaf',
    age: 22,
    gender: 'male'
};

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



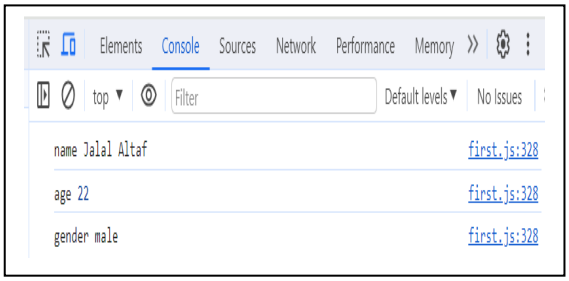
2. For...of Loop:

- **Use cases:**
  - Used to iterate over iterable objects, such as arrays, strings, maps, sets, etc.
  - Suitable for cases where you want to access the values directly, rather than the keys or indices
- **Syntax:**

```
for (variable of iterable) {
    // code to be executed
}
```

- **Example:**

```
const person = {
    name: 'Jalal Altaf',
    age: 22,
    gender: 'male'
};

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

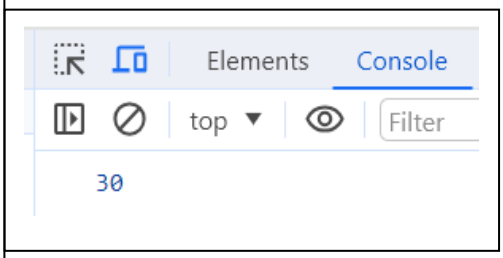


Comparison:

- Use '**for...in**' when you need to iterate over the properties of an object, including inherited properties.
- Use '**for...of'** when you need to iterate over the values of an iterable object like an array or string.
- Avoid using '**for...in'** with arrays, as it might include unexpected properties from the prototype chain.
- **'for...of'** is generally more concise and is the preferred choice for iterating over iterable values in modern JavaScript.

Question # 04: Define a function calculateAverage that takes an array of numbers as an argument and returns the average value.

```
function calculateAverage(numbers) {
    // Ensure that the array is not empty to avoid division by zero
    if (numbers.length === 0) {
        return 0;
    }

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

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

    return average;
}
const numbersArray = [10, 20, 30, 40, 50];
const result = calculateAverage(numbersArray);

console.log(result);
```



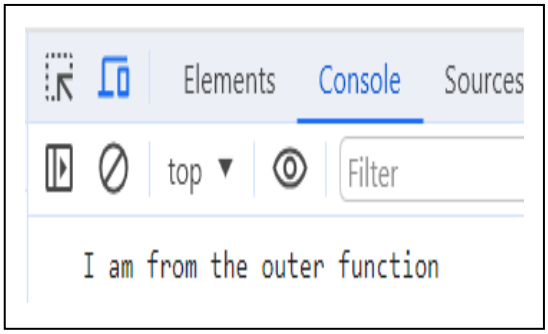
Question # 05:

Answer:

In JavaScript, a closure is a combination of a function and the lexical environment within which that function was declared. Closures allow a function to access variables from its outer (enclosing) scope even after that scope has finished executing. In other words, a closure "closes over" the variables from its outer scope, preserving them.

Example:

```
function outerFunction() {
    let outerVariable = 'I am from the outer function';

    function innerFunction() {
        console.log(outerVariable);
    }

    return innerFunction;
}

// Create a closure by calling outerFunction and assigning the returned function
const closureFunction = outerFunction();

// Call the closure, and it still has access to the outerVariable
closureFunction();
```



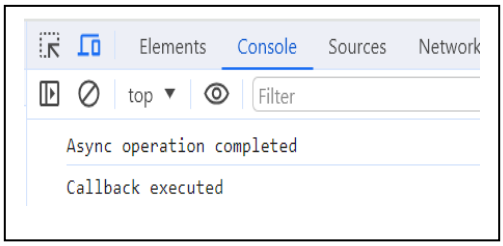
Practical Use of Closures:

- **Data Encapsulation**: Closures are often used to create private variables and methods, encapsulating data within a function's scope and preventing direct access from outside.

```
function createCounter() {
    let count = 0;

    return {
        increment: function() {
            count++;
        },
        getValue: function() {
            return count;
        }
    };
}

const counter = createCounter();
counter.increment();
console.log(counter.getValue());
```

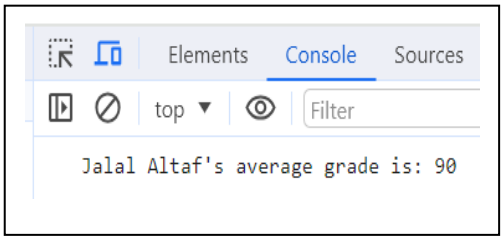
- **Calling Functions**: Closures are commonly employed in scenarios involving callback functions, allowing the callback to access variables from the outer function's scope.

```
function doSomethingAsync(callback) {
    setTimeout(function() {
        console.log('Async operation completed');
        callback();
    }, 1000);
}

function onComplete() {
    console.log('Callback executed');
}

doSomethingAsync(onComplete);
```



Question # 06: 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: 'Jalal Altaf',
    age: 22,
    grades: [90, 85, 92, 88, 95],
    calculateAverage: function() {
        if (this.grades.length === 0) {
            return 0;
        }
        const sum = this.grades.reduce((acc, grade) => acc + grade, 0);
        const average = sum / this.grades.length;
        return average;
    }
};
const averageGrade = student.calculateAverage();
console.log(`${student.name}'s average grade is: ${averageGrade}`);
```



Question # 07: How can you clone an object in JavaScript and also give one example each deep copy, shallow copy, and reference copy.

Answer: In JavaScript, you can clone an object using different methods, and the choice of method depends on whether you need a deep copy, a shallow copy, or a reference copy.

Shallow Copy: A shallow copy creates a new object and copies the properties of the original object to the new object. However, if the original object contains nested objects, only references to those nested objects are copied, not the nested objects themselves.

```
const originalObject = {
    name: 'John',
    age: 25,
    hobbies: ['reading', 'traveling']
};
const shallowCopy = Object.assign({}, originalObject);
shallowCopy.age = 30;
shallowCopy.hobbies.push('swimming');
console.log(originalObject.age);
console.log(originalObject.hobbies);
```

Deep Copy: A deep copy creates a new object and recursively copies all properties, including nested objects. This ensures that changes in the new object do not affect the original object or any nested objects.

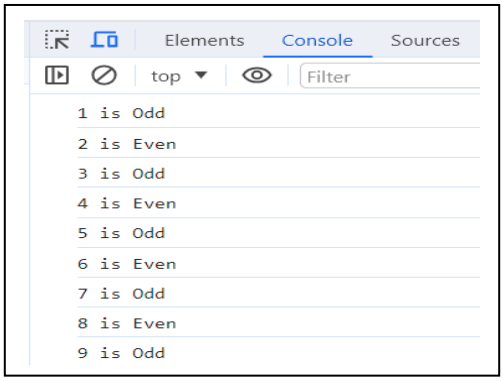
```
const _ = require('lodash');
const originalObject = {
    name: 'John',
    age: 25,
    hobbies: ['reading', 'traveling']
};
const deepCopy = _.cloneDeep(originalObject);
deepCopy.age = 30;
deepCopy.hobbies.push('singing');
console.log(originalObject.age);
console.log(originalObject.hobbies);
```

Reference Copy: A reference copy creates a new reference to the same object in memory. Any changes made to the object through one reference will affect the other reference.

```
const originalObject = {
    name: 'Jalal Altaf',
    age: 22,
    hobbies: ['reading', 'traveling']
};
const referenceCopy = originalObject;
referenceCopy.age = 30;
referenceCopy.hobbies.push('singing');
console.log(originalObject.age);
console.log(originalObject.hobbies);
```

Question # 08: 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];

for (let i = 0; i < numbers.length; i++) {
    const isEven = numbers[i] % 2 === 0;
    const result = isEven ? 'Even' : 'Odd';

    console.log(`${numbers[i]} is ${result}`);
}
```



Question # 09: Describe the differences between the for loop, while loop, and do...while loop in JavaScript. When might you use each?

Answer: In JavaScript, the for loop, while loop, and do...while loop are different constructs for implementing iteration, and each has its own use cases and characteristics.

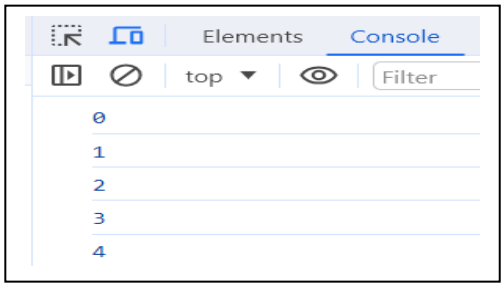
1. **For Loop**:

Syntax:

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

Example:

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



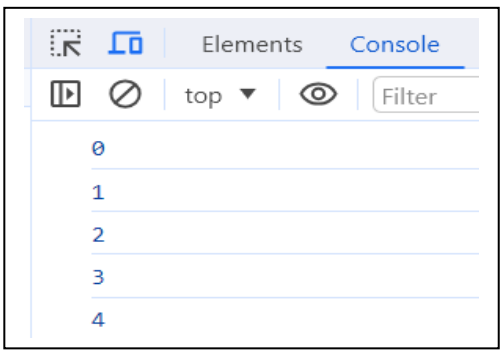
2. **While Loop**:

Syntax:

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

Example:

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



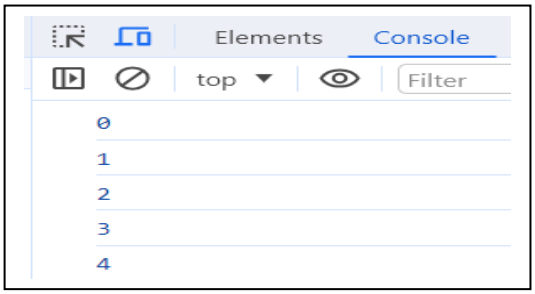
3. Do While Loop:

Syntax:

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

Example:

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

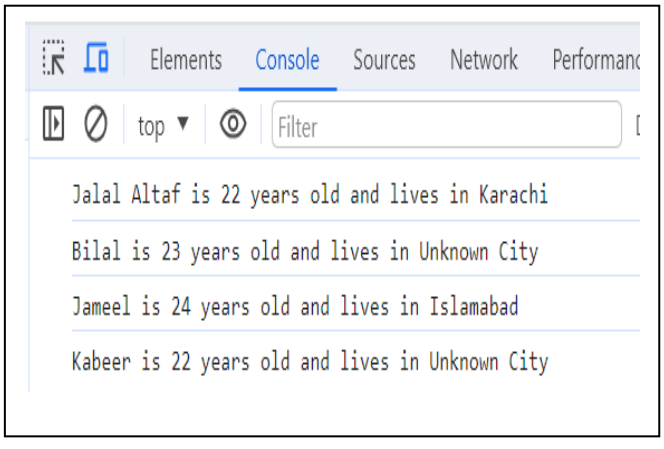


When to Use Each:

- Use for loop when you know the number of iterations and want to iterate over a specific range or array.
- Use while loop when the number of iterations is not known beforehand and depends on a condition that is checked before each iteration.
- Use do...while loop when you want to ensure that the loop body is executed at least once, regardless of the condition.

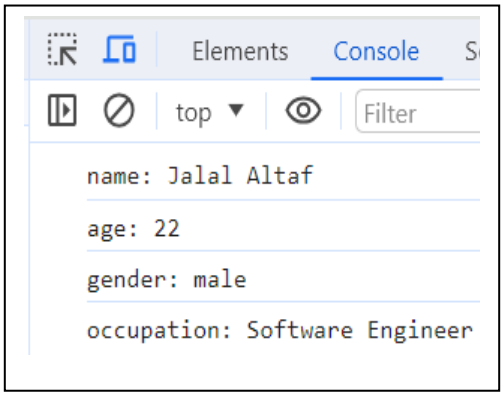
Question # 10: Provide an example of using optional chaining within a loop to access a potentially missing property of an object.

```
const people = [
    { name: 'Jalal Altaf', age: 22, address: { city: 'Karachi' } },
    { name: 'Bilal', age: 23 },
    { name: 'Jameel', age: 24, address: { city: 'Islamabad' } },
    { name: 'Kabeer', age: 22 },
];

for (const person of people) {
    const city = person?.address?.city || 'Unknown City';
    console.log(`${person.name} is ${person.age} years old and lives in ${city}`);
}
```



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

```
const person = {
    name: 'Jalal Altaf',
    age: 22,
    gender: 'male',
    occupation: 'Software Engineer'
};

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



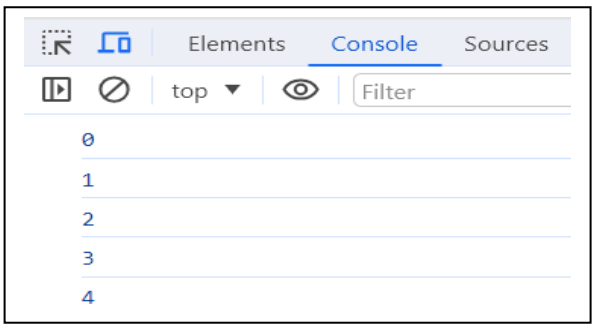
Question # 12: Explain the use of the break and continue statements within loops. Provide scenarios where each might be used.

ANSWER:

Break Statement: The break statement is used to exit a loop prematurely. When encountered, it immediately terminates the loop, and control is transferred to the statement following the loop.

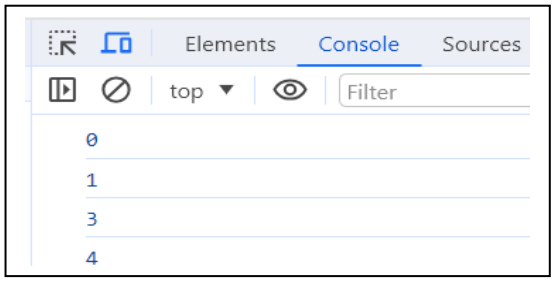
Example:

```
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        break;
    }
    console.log(i);
}
```



Continue Statement: The continue statement is used to skip the rest of the code inside a loop for the current iteration and proceed to the next iteration.

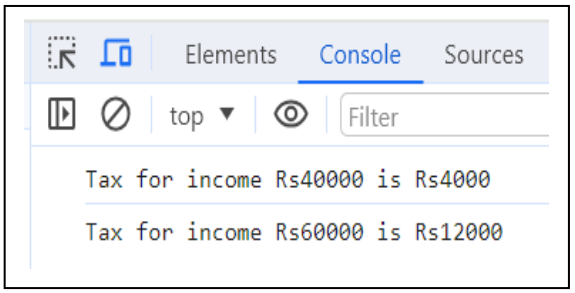
```
for (let i = 0; i < 5; i++) {
    if (i === 2) {
        continue;
    }
    console.log(i);
}
```



Question # 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.

ANSWER:

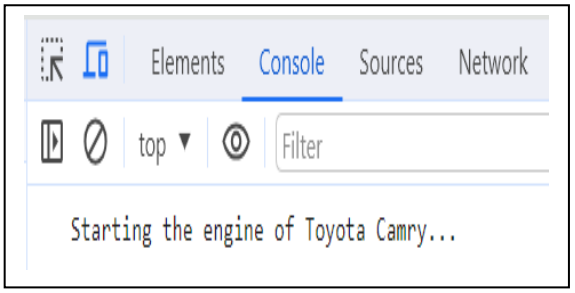
```
function calculateTax(income) {
    const lowRate = 0.1;
    const highRate = 0.2;
    const taxRate = income <= 50000 ? lowRate : highRate;
    const taxAmount = income * taxRate;
    return taxAmount;
}
const income1 = 40000;
const tax1 = calculateTax(income1);
console.log(`Tax for income Rs${income1} is Rs${tax1}`);
const income2 = 60000;
const tax2 = calculateTax(income2);
console.log(`Tax for income Rs${income2} is Rs${tax2}`);
```



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

ANSWER:

```
const car = {
    make: 'Toyota',
    model: 'Camry',
    startEngine: function() {
        console.log(`Starting the engine of ${this.make} ${this.model}...`);
    }
};
car.startEngine();
```



Question # 14: Explain the differences between regular functions and arrow functions in terms of scope, this binding, and their use as methods.

ANSWER:

Scope:

Regular Functions:

- Have their own arguments object.
- Have their own '**this'** binding, which is dynamically scoped based on how the function is called.
- Bind their own '**this'** value when invoked.

Arrow Functions:

- Do not have their own '**arguments'** object.
- Inherit the '**this'** value from their enclosing scope (lexical scoping).
- Do not bind their own '**this'** value; they inherit it from the surrounding context.

'this' Binding:

Regular Functions:

- Have their own this binding, which is dynamically determined at runtime based on how the function is called.
- Suited for methods where '**this'** needs to be determined dynamically, such as in an object method.

Arrow Functions:

- Inherit the 'this' value from their enclosing scope (lexical scoping).
- Do not bind their own '**this'** value, making them less suitable for methods where dynamic '**this'** is required.

Use as Methods:

Regular Functions:

- Suitable for object methods where dynamic this binding is needed.
- Have access to the **arguments** object.

```
const object = {
    regularMethod: function() {
        console.log(this);
    }
};
```

Arrow Functions:

- Not well-suited for object methods when dynamic this binding is required.
- Concise syntax is beneficial for short and simple functions.

```
const object = {
    arrowMethod: () => {
        console.log(this);
    }
};
```

Example:

```
function regularFunction() {
    console.log(this);
}
const arrowFunction = () => {
    console.log(this);
}
regularFunction();
arrowFunction();
```