JavaScript Fundamentals Part 2

1. Activating Strict Mode

Using `"use strict";` activates JavaScript's **strict mode**, which enforces stricter parsing and error handling rules. This helps developers write safer and more optimized code by catching common coding errors early.

- **Key Benefits**:
  - Prevents the use of undeclared variables.
  - Makes assignments to non-writable properties or constants throw errors.
  - Disallows the use of certain reserved keywords for future JavaScript versions (like `interface` or `private`).
- **Example Without Strict Mode**:

```
x = 10; // No error, but this creates a global variable unintentionally.
console.log(x); // 10
```

- **Example With Strict Mode**:

```
"use strict";
x = 10; // Error: x is not defined
```

- **Where to Use Strict Mode**:
  - At the top of a script: Applies to the entire file.
  - Inside a function: Applies only within the function.

```
function calculate() {
  "use strict";
  y = 5; // Error: y is not defined
}
calculate();
```

Using strict mode is considered a best practice in modern JavaScript development.

---

2. Functions

---

1. **Introduction to Functions**

Functions are reusable blocks of code designed to perform a specific task. They help reduce repetition and make code more modular and maintainable.

- **Syntax**:

```
function functionName(parameters) {
  // Function body
  return value; // optional
}
```

- **Key Components**:
  - `functionName`: Identifier for the function.
  - `parameters`: Variables that the function uses as input.
  - `return`: Outputs a value; if omitted, the function returns `undefined`.

Example 1

```
function logger() {
  console.log("My name is Hakeem");
}
```

Calling `logger()` executes the code inside the function.

2. **Passing Arguments**

Functions can accept inputs (arguments) to make them dynamic and flexible.

Example 2

```
function fruitProcessor(apples, oranges) {
  const juice = `Juice with ${apples} apples and ${oranges} oranges.`;
  return juice;
}
const appleJuice = fruitProcessor(5, 0);
console.log(appleJuice); // Output: "Juice with 5 apples and 0 oranges."
```

- **Arguments vs. Parameters**:
  - *Parameters* are placeholders defined when creating a function.
  - *Arguments* are actual values passed to the function during execution.

3. **Function Declarations vs. Function Expressions**

Function Declarations

Functions defined using the `function` keyword. They are **hoisted**, meaning they can be used before they are defined in the code.

```
function calcAge1(birthYear) {
  return 2037 - birthYear;
}
console.log(calcAge1(1991)); // Output: 46
```

Function Expressions

Functions assigned to a variable. These are **not hoisted** and must be defined before they are used.

```
const calcAge2 = function (birthYear) {
  return 2037 - birthYear;
};
console.log(calcAge2(1991)); // Output: 46
```

Key Differences:

- Declarations can be invoked before their definition due to hoisting.
- Expressions behave like any other variable and are scoped accordingly.

4. **Arrow Functions**

A shorter syntax for writing functions. Arrow functions do not have their own `this` context, making them ideal for simple operations but limited in certain object-oriented contexts.

```
const calcAge3 = (birthYear) => 2037 - birthYear;
console.log(calcAge3(1991)); // Output: 46
```

- **Multi-line Arrow Function**:
  If the function body has multiple lines, use curly braces `{}` and an explicit `return` statement:

```
const yearsUntilRetirement = (birthYear, firstName) => {
  const age = 2037 - birthYear;
  const retirement = 65 - age;
  return `${firstName} retires in ${retirement} years.`;
};
console.log(yearsUntilRetirement(1991, "Hakeem"));
```

5. **Functions Calling Other Functions**

Functions can call other functions for modularity and reuse.

Example 3

```
function cutFruitPieces(fruit) {
  return fruit * 4; // Each fruit is cut into 4 pieces
}

function fruitProcessor(apples, oranges) {
  const applePieces = cutFruitPieces(apples);
  const orangePieces = cutFruitPieces(oranges);

  const juice = `Juice with ${applePieces} pieces of apple and ${orangePieces}
pieces of orange.`;
  return juice;
}
console.log(fruitProcessor(2, 3)); // Output: Juice with 8 pieces of apple and
12 pieces of orange.
```

This technique is called **function composition**, where smaller functions combine to perform a more complex operation.

6. **Conditionals in Functions**

Adding logic to functions improves their usability.

Example 4

```
const yearsUntilRetirement = function (birthYear, firstName) {
  const age = 2037 - birthYear;
  const retirement = 65 - age;

  if (retirement > 0) {
    return `${firstName} retires in ${retirement} years.`;
  } else {
    return `${firstName} has already retired! 🎉`;
  }
};

console.log(yearsUntilRetirement(1991, "Hakeem"));
console.log(yearsUntilRetirement(1950, "Mike"));
```

Summary for functions

- **Function Declaration**: Hoisted, reusable with `function` keyword.
- **Function Expression**: Stored in variables, not hoisted.
- **Arrow Function**: Concise syntax, no `this` binding.
- **Modular Design**: Use smaller functions to create larger, complex operations.
- **Conditionals**: Add decision-making to functions for dynamic behavior.

2. Arrays

Arrays are used to store multiple values in a single variable. They are one of the most commonly used data structures in JavaScript.

Creating Arrays

You can create arrays using square brackets `[]` or the `Array` constructor.

- Example:

```
const fruits = ["Apple", "Banana", "Orange"]; // Using square brackets
const numbers = new Array(10, 20, 30); // Using Array constructor
```

Common Methods

1. **Adding Elements**:
   - `push()`: Adds an element to the **end** of the array.

     ```
     fruits.push("Grapes");
     console.log(fruits); // ['Apple', 'Banana', 'Orange', 'Grapes']
     ```

   - `unshift()`: Adds an element to the **start** of the array.

     ```
     fruits.unshift("Mango");
     console.log(fruits); // ['Mango', 'Apple', 'Banana', 'Orange']
     ```

2. **Removing Elements**:
   - `pop()`: Removes the last element of the array.

     ```
     const lastFruit = fruits.pop();
     console.log(lastFruit); // 'Orange'
     console.log(fruits); // ['Apple', 'Banana']
     ```

   - `shift()`: Removes the first element of the array.

     ```
     const firstFruit = fruits.shift();
     console.log(firstFruit); // 'Apple'
     console.log(fruits); // ['Banana', 'Orange']
     ```

3. **Searching**:

   - `indexOf()`: Finds the index of a specific element. Returns `-1` if not found.

     ```
     console.log(fruits.indexOf("Banana")); // 1
     console.log(fruits.indexOf("Grapes")); // -1
     ```

   - `includes()`: Checks if the array contains a specific value.

     ```
     console.log(fruits.includes("Banana")); // true
     console.log(fruits.includes("Grapes")); // false
     ```

4. **Other Methods**:

   - `slice(start, end)`: Creates a new array by extracting elements between the `start` and `end` indices.
   - `splice(start, deleteCount, ...items)`: Adds or removes elements at a specific position.
   - `join(separator)`: Combines elements into a string separated by `separator`.

---

3. Loops

Loops allow repetitive execution of code blocks based on a condition.

For Loop

Executes a block of code a fixed number of times. It is ideal when you know how many iterations are needed.

```
for (let i = 0; i < 5; i++) {
  console.log(`Iteration ${i}`);
}
// Output:
// Iteration 0
// Iteration 1
// Iteration 2
// Iteration 3
// Iteration 4
```

While Loop

Executes as long as a condition is true. It is useful when the number of iterations is not predetermined.

```
let i = 0;
while (i < 5) {
  console.log(`Iteration ${i}`);
  i++;
}
// Output:
// Iteration 0
// Iteration 1
// Iteration 2
// Iteration 3
// Iteration 4
```

Summary

- Arrays are versatile, with many built-in methods for manipulation.
- Loops enable repetitive tasks, with variations like `for` and `while` suited to different scenarios.

---

4. Objects

Objects are one of the fundamental data types in JavaScript. They are collections of key-value pairs, where the keys are properties (usually strings) and the values can be any data type, including functions or other objects.

Creating Objects

You can create an object using curly braces `{}` and define properties inside as key-value pairs.

- **Example**:

  ```
  const person = {
    firstName: "Moyo",
    lastName: "Doe",
    age: 30,
    friends: ["Mike", "Jane"],
    isEmployed: true,
  };
  ```

  Here:

  - `firstName`, `lastName`, `age`, and `friends` are **keys (properties)**.
  - `'Moyo'`, `'Doe'`, `30`, and `['Mike', 'Jane']` are the **values**.

Accessing Properties

You can retrieve the values of object properties using either **dot notation** or **bracket notation**.

1. **Dot Notation**:
    - Use the property name directly after the dot (`.`).
    - Example:

```
console.log(person.firstName); // Output: Moyo
console.log(person.age); // Output: 30
```

2. **Bracket Notation**:
    - Use square brackets `[]` and wrap the property name in quotes.
    - Useful when the property name is dynamic or contains special characters.
    - Example:

```
console.log(person["lastName"]); // Output: Doe
const key = "friends";
console.log(person[key]); // Output: ['Mike', 'Jane']
```

Modifying Properties

You can update or add properties to an object.

- Example:

```
person.age = 31; // Update existing property
person.job = "Engineer"; // Add a new property
console.log(person);
```

Object Methods

In JavaScript, methods are functions that are defined inside objects and are used to perform actions on or with that object's data.

Defining Methods

You can add a method by defining a function as the value of a property.

- **Example**:

```
const person = {
  firstName: "Moyo",
  lastName: "Doe",
  birthYear: 1990,
  calcAge: function () {
    return 2024 - this.birthYear;
  },
};
```

Here:

- `calcAge` is a method of the `person` object.
- `this` refers to the object that the method belongs to (`person` in this case).

Calling Methods

Invoke a method using dot notation or bracket notation, just like accessing properties.

- **Example**:

```
console.log(person.calcAge()); // Output: 34
```

Shorter Syntax (ES6)

In modern JavaScript, you can define methods using shorthand syntax.

- Example:

```
const person = {
  firstName: "Moyo",
  lastName: "Doe",
  birthYear: 1990,
  calcAge() {
    return 2024 - this.birthYear;
  },
};
```

Key Points About `this` in Methods

- `this` refers to the object that called the method.
- Example:

```
const person = {
  name: "Jane",
```

```
  greet() {
    console.log(`Hello, my name is ${this.name}`);
  },
};
person.greet(); // Output: Hello, my name is Jane
```

- If `this` is used in a standalone function or in a method assigned to another object, it might not refer to the original object.

```
const greet = person.greet;
greet(); // Output: Hello, my name is undefined (in strict mode, this
will be undefined)
```

Summary for Object

- Objects group related data and behavior using key-value pairs.
- Access object properties with dot or bracket notation.
- Methods add functionality to objects, and `this` helps reference the object context within a method.

Challenges **questions** and their corresponding **solutions**

Questions:

Challenge 1: Gymnastics Teams

Scenario

The two gymnastics teams, **Hidee** and **Silas**, compete in a new discipline. Each team competes three times, and the average score is calculated. A team wins **only if its average score is at least double the other team's average score**.

Tasks

1. **Arrow Function for Averages**:
    - Write an arrow function `calcAverage` to calculate the average of three scores.
2. **Calculate Averages for Both Teams**:
    - Use `calcAverage` to determine the average scores for Hidee and Silas.
3. **Determine the Winner**:

- Write a function `checkWinner` that:
  - Takes the average scores of both teams (`avgHidee` and `avgSilas`) as parameters.
  - Logs the winner and their score, e.g., `"Silas wins (30 vs. 13)"`.
  - No team wins if the criteria are not met.

4. **Test the Function**:

   - Use the following test data:
     - **Data 1**: Hidee scores **44, 23, 71**. Silas scores **65, 54, 49**.
     - **Data 2**: Hidee scores **85, 54, 41**. Silas scores **23, 34, 27**.

Hints

- To calculate an average: `(score1 + score2 + score3) / 3`.
- A team wins if `avgTeamA >= 2 * avgTeamB`.

Challenge 2: Tip Calculator

Scenario - Challenge 2

Victor is still building his tip calculator, using the same rules as before:

- Tip **15%** of the bill if the bill is between 50 and 300.
- Tip **20%** otherwise.

Tasks - Challenge 2

1. **Tip Calculation**:

   - Write a `calcTip` function that:
     - Takes the bill value as input.
     - Returns the corresponding tip based on the above rules.
     - Test the function using a bill value of **100**.

2. **Use Arrays**:

   - Create an array `bills` with the test data.
   - Use the `calcTip` function to populate:
     - An array `tips` containing the tip values.
     - An array `total` containing the total bill values (bill + tip).

Test Data

- **Bills**: 125, 555, 44.

Bonus

- No intermediate variables—directly call the `calcTip` function inside the arrays.

Challenge 3: BMI Comparison

Scenario - Challenge 3

Let's go back to Yinka and Moyo comparing their BMIs! This time, let's use objects to implement the calculations! Remember:
BMI = mass / height² (or mass / (height * height)), where:

- **Mass**: kg
- **Height**: m.

Tasks - Challenge 3

1. **Objects for Each Person**:
    - Create objects for **Yinka Biobaku** and **Moyo Oladapo**.
    - Include properties for `fullName`, `mass`, and `height`.
2. **Calculate BMI**:
    - Add a method `calcBMI` to both objects that:
        - Calculates and stores the BMI as a property.
        - Returns the BMI value.
3. **Compare BMIs**:
    - Log the person with the higher BMI, e.g.,
      `"Moyo Oladapo's BMI (28.3) is higher than Yinka Biobaku's (23.9)!"`.

Test Data - Challenge 3

- **Yinka**: 78 kg, 1.69 m.
- **Moyo**: 92 kg, 1.95 m.

Challenge 4: Tip Calculator with Loops

Scenario - Challenge 4

Extend Juwon's tip calculator to handle multiple bills and calculate average totals using loops.

Tasks - Challenge 4

1. **Test Data**:
    - Create an array `bills` containing:

        - **10 values**: 22, 295, 176, 440, 37, 105, 10, 1100, 86, and 52.

2. **Empty Arrays**:
    - Create two empty arrays: `tips` and `totals`.
3. **Calculate Tips and Totals**:
    - Use the `calcTip` function from Challenge 2 to:
        - Calculate tips for each bill.
        - Compute total values (bill + tip).
    - Use a `for` loop to iterate through all bill values.
4. **Bonus Task: Average Calculation**:
    - Write a function `calcAverage` that:
        - Takes an array `arr` as input.
        - Calculates and returns the average of the array elements.
        - Call this function with the `totals` array.

Hints - Challenge 4

- Sum all values in the array and divide by the array length.
- Use `push()` to add elements to the arrays inside the loop.
