JavaScript Fundamentals Part 1

1. Linking a JavaScript File

Linking a JavaScript file to an HTML document is a fundamental step in making the web page interactive. Here's a complete guide to linking and optimizing JavaScript files in an HTML page.

How to Link a JavaScript File

To link a JavaScript file to an HTML document, use the `<script>` tag. This tag tells the browser to load and execute the JavaScript file.

- **Basic Syntax**:

```
<script src="script.js"></script>
```

- **Where to Place the `<script>` Tag**:
  You can place the `<script>` tag either in the `<head>` or before the closing `</body>` tag. However, where you place it affects page loading behavior.

  1. **In the `<head>` Section**:
     If you place the `<script>` tag inside the `<head>`, the browser will pause rendering the page until the script is downloaded and executed. This can delay page load time.

```
<head>
  <script src="script.js"></script>
</head>
```

  2. **Before `</body>` Tag (Recommended)**:
     It's generally best to place the `<script>` tag just before the closing `</body>` tag. This ensures the HTML content is loaded first, and the JavaScript runs afterward, improving page load performance.

```
<body>
  <h1>Welcome to My Website</h1>
  <script src="script.js"></script>
</body>
```

Using `defer` to Improve Performance

If you want to improve the loading performance of your page, consider using the `defer` attribute. It allows the browser to download the JavaScript file in the background while the HTML content is still being

parsed. The script will be executed only after the HTML document is fully loaded, ensuring the DOM is ready for manipulation.

- **Syntax with `defer`**:

```
<script defer src="script.js"></script>
```

- **Why Use `defer`?**
  - **Non-blocking**: It allows the HTML to continue loading while the JavaScript file is being fetched, without blocking the rendering of the page.
  - **Execution Order**: If multiple scripts are deferred, they will be executed in the order they appear in the HTML document.
  - **Ensures DOM Ready**: Since scripts are executed after the document is parsed, you can safely interact with the DOM elements inside the script.

Difference Between `defer`, `async`, and Regular `<script>` Tags

- **Without `defer`**:
  - The script is executed immediately when encountered, which can block the page rendering if the script is large or takes time to load.
  - Example:

```
<script src="script.js"></script>
```

- **With `async`**:
  - The `async` attribute tells the browser to load the script asynchronously and execute it as soon as it's available. However, it doesn't guarantee the execution order if multiple scripts are present. It can also execute before the HTML is fully parsed, which can cause issues if the script interacts with the DOM.
  - Example:

```
<script async src="script.js"></script>
```

- **With `defer`**:
  - The script is loaded asynchronously, but it’s executed only after the entire HTML document is parsed. It maintains the execution order of multiple scripts.
  - Example:

```
<script defer src="script.js"></script>
```

When to Use `defer`

- Use `defer` when your script depends on the DOM being fully loaded or when interacting with DOM elements.
- It's ideal for most scripts, especially those that require DOM manipulation (e.g., JavaScript functions, event listeners, etc.).

Example: Linking a JavaScript File with `defer`

Here's a simple example where JavaScript is linked with `defer` for optimal performance:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>JavaScript Linking Example</title>

    <!-- Defer ensures the script loads after the HTML is parsed -->
    <script defer src="script.js"></script>
  </head>
  <body>
    <h1>Welcome to My Website</h1>
    <p>This content will be rendered before the JavaScript runs.</p>
  </body>
</html>
```

Inline JavaScript (Not Recommended for Large Projects)

Alternatively, you can write JavaScript directly within the HTML file using the `<script>` tag. However, this method is discouraged for larger projects as it makes the code harder to manage.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Inline JavaScript Example</title>
  </head>
  <body>
    <h1>Hello, Inline JavaScript!</h1>

    <!-- Inline JavaScript directly within the HTML file -->
    <script>
```

```
        console.log("This is inline JavaScript.");
      </script>
    </body>
  </html>
```

Best Practice

- **Separate JavaScript**: For maintainability, always link JavaScript in an external file instead of writing inline JavaScript in your HTML document.

By linking JavaScript externally, you improve page performance, separate content (HTML) from behavior (JavaScript), and make your codebase more modular and scalable.

Summary for Linking a JavaScript File

- **Place `<script>` Tags**: Preferably before the `</body>` tag to avoid blocking page rendering.
- **Use `defer`**: To download the JavaScript asynchronously and ensure it runs only after the HTML is fully parsed, enhancing performance.
- **Avoid Using `async` in Most Cases**: Unless your script doesn't depend on the DOM being fully loaded and can be executed independently.

By following these practices, you'll ensure better performance, smoother page load, and more reliable JavaScript execution.

2. Values and Variables

In JavaScript, values represent the data you want to work with, and variables are used to store these values for later use.

Example:

```
console.log("Hakeem"); // Outputs the string "Hakeem"
console.log(23); // Outputs the number 23

let firstName = "Olaogun";
console.log(firstName); // Outputs the value stored in the variable firstName,
which is "Olaogun"
```

Variables

A variable is a container for storing data values. In the above example, the variable `firstName` stores the value `"Olaogun"`. When we refer to `firstName`, we are actually referencing the string `"Olaogun"` that's stored inside it.

JavaScript allows us to declare variables using three different keywords:

- `let`: Used to declare variables that **can be reassigned** later.

- `const`: Used to declare variables that **cannot be reassigned** after being initialized.
- `var`: The older way of declaring variables, which is now less commonly used due to certain limitations, such as function scope and hoisting issues (explained below).

Naming Conventions

When naming variables in JavaScript, there are a few important rules and best practices to follow:

1. Variable names must start with a letter, an underscore (_), or a dollar sign ($), but **cannot** start with a number.
2. Avoid using JavaScript **reserved keywords** (like `function`, `return`, `var`, etc.) as variable names, unless you're prefixing them with special symbols like `$`.

Example of Valid and Invalid Variable Names

```
let hakeem_olaogun = "HO"; // Valid: underscores are allowed
let $function = 27; // Valid: using a dollar sign to avoid the reserved keyword
'function'
// let 1stName = "Hakeem"; // Invalid: Variable names cannot start with a
number
```

Best Practices

1. **Use descriptive names**: Variables should clearly describe the data they hold. For example, `customerAge` is more meaningful than just `age`.
2. **Camel Case**: Follow the camelCase naming convention, where the first word is lowercase, and subsequent words start with an uppercase letter. For example, `myFirstJob`.
3. **Constants in UPPERCASE**: Variables that hold constant values (like mathematical constants or configuration values) are often written in uppercase. For example, `const PI = 3.14159`.

3. Data Types

JavaScript has different data types to represent different kinds of values, such as numbers, strings, and booleans.

Primitive Data Types

These are the most basic data types in JavaScript:

1. **Number**: Represents numeric values. For example, `23` or `9.81`.
2. **String**: Represents text values. Strings are written within quotes, like `"Hello, World!"`.
3. **Boolean**: Represents `true` or `false` values. These are used for logical decisions.
4. **Undefined**: A variable that has been declared but not assigned a value is `undefined`.
5. **Null**: Represents an intentional absence of any value.
6. **Symbol**: A unique and immutable primitive introduced in ES6.
7. **BigInt**: Used to represent integers with arbitrary precision (values larger than `Number.MAX_SAFE_INTEGER`).

Dynamic Typing

JavaScript is dynamically typed, meaning the type of a variable can change at runtime.

Example 1

```
let javascriptIsFun = true;
console.log(typeof javascriptIsFun); // Outputs "boolean"
```

In this example, `javascriptIsFun` is a boolean type because it holds the value `true`. You can check the type of any variable using the `typeof` operator.

Example 2

```
javascriptIsFun = "YES!";
console.log(typeof javascriptIsFun); // Outputs "string"
```

Here, `javascriptIsFun` was initially a boolean, but it has now been reassigned to a string. JavaScript allows you to change the data type of variables on the fly.

Special Cases

```
let year;
console.log(typeof year); // Outputs "undefined"
```

- **Undefined**: When a variable is declared but not assigned any value, its type is `undefined`.

```
console.log(typeof null); // Outputs "object"
```

- **Null**: Despite being an intentional absence of value, `typeof null` returns "object". This is a quirk in JavaScript dating back to its early days and is considered a bug, but it's important to be aware of.

4. let, const, and var

In JavaScript, we use `let`, `const`, and `var` to declare variables. Each has its own specific behavior.

Using `let`

Variables declared with `let` can be reassigned later.

```
let age = 30;
age = 31; // Reassigning the value of age
```

```
console.log(age); // Outputs: 31
```

- **Block-scoped**: `let` is block-scoped, meaning its value is limited to the block it is defined in (such as within `{ }` braces).

Using `const`

Variables declared with `const` cannot be reassigned after they are initialized.

```
const birthYear = 1991;
birthYear = 1992; // This will throw an error: Assignment to constant variable.
```

- **Immutable**: `const` is used when you know the value of the variable will not change throughout the program.
- **Block-scoped**: Like `let`, `const` is also block-scoped.

Using `var`

Before `let` and `const`, JavaScript only had `var` for declaring variables. `var` behaves differently compared to `let` and `const` because:

- It is **function-scoped**, not block-scoped.
- It gets **hoisted** to the top of its scope, meaning you can use the variable even before it's declared, which can lead to unintended behavior.

```
var job = "programmer";
job = "teacher"; // Reassigning the value of job
console.log(job); // Outputs: 'teacher'
```

Hoisting Example

```
console.log(salary); // Outputs: undefined, due to hoisting
var salary = 50000;
```

Here, JavaScript "hoists" the declaration of `salary` to the top of its scope but **not the assignment**. That's why the output is `undefined`. This can cause confusion, which is why it's generally recommended to use `let` and `const`.

Best Practice for let, const, and var

- Use `const` for variables that should not be reassigned.
- Use `let` for variables that need to be reassigned.

- Avoid `var` due to its function-scoped behavior and hoisting issues, which can lead to bugs in larger codebases.

These concepts of variables and data types form the foundation of programming in JavaScript, enabling you to store and manipulate data effectively in your programs.

5. Basic Operators

Operators in JavaScript are symbols that tell the interpreter to perform specific mathematical or logical manipulations. Let's explore some of the basic operators used in JavaScript:

Basic Operators Example:

```
const now = 2037;
const ageHakeem = now - 1991;
const ageYinka = now - 2018;
console.log(ageHakeem, ageYinka); // Outputs: 46, 19
```

Math Operators

- **Addition (`+`)**: Adds two values together.
  - Example: `5 + 2` results in `7`.
- **Subtraction (`-`)**: Subtracts one value from another.
  - Example: `10 - 3` results in `7`.
- **Multiplication (`*`)**: Multiplies two values.
  - Example: `4 * 3` results in `12`.
- **Division (`/`)**: Divides one value by another.
  - Example: `12 / 4` results in `3`.
- **Exponentiation (`\*\*\`)**: Raises the first value to the power of the second value.
  - Example: `2 ** 3` results in `8` (which is `2 * 2 * 2`).

These operators allow you to perform basic arithmetic operations, as seen in the example where the current year is subtracted from the birth year to calculate ages.

Assignment Operators

Assignment operators are used to assign values to variables while performing arithmetic operations in shorthand.

```
let x = 10 + 5; // 15 (adds 10 and 5, assigns the result to x)
x += 10; // x = x + 10; which results in x = 25
x *= 4; // x = x * 4; which results in x = 100
x++; // Increments x by 1 (x = x + 1), resulting in x = 101
x--; // Decrements x by 1, so x = 100 again
console.log(x); // Outputs: 100
```

- `+=` adds a value to the variable and updates it.
- `*=` multiplies the variable by a value and updates it.
- `++` increments the value by 1.
- `--` decrements the value by 1.

Comparison Operators

These operators compare two values and return a boolean (`true` or `false`).

```
console.log(ageHakeem > ageYinka); // true, because 46 is greater than 19
```

- `>` checks if the value on the left is greater than the one on the right.
- `<` checks if the value on the left is smaller than the one on the right.
- `>=` checks if the value on the left is greater than or equal to the one on the right.
- `<=` checks if the value on the left is smaller than or equal to the one on the right.

6. Operator Precedence

Operator precedence determines the order in which different operators are evaluated in a complex expression. JavaScript follows a hierarchy of precedence where some operators are evaluated before others.

Operator Precedence Example:

```
let x, y;
x = y = 25 - 10 - 5; // x = y = 10
console.log(x, y); // Outputs: 10, 10
```

Here's what happens:

1. **Subtraction (`-`)**: JavaScript evaluates `25 - 10 - 5` first because subtraction has a higher precedence than assignment.
2. **Assignment (`=`)**: Once the subtraction is evaluated to `10`, JavaScript assigns the result to both `y` and `x` from right to left.

JavaScript evaluates expressions from **left to right** and follows precedence rules where **multiplication/division** takes precedence over **addition/subtraction**, and so on.

7. Strings and Template Literals

In JavaScript, strings are used to represent text. You can combine strings with variables and expressions using concatenation or template literals.

String Concatenation (Old Way)

In older versions of JavaScript, you would use the + operator to concatenate strings and variables together.

```
const firstName = "Hakeem";
const job = "teacher";
const birthYear = 1991;
const year = 2037;

const hakeem =
  "I'm " + firstName + ", a " + (year - birthYear) + " year old " + job + "!";
console.log(hakeem);
// Outputs: I'm Hakeem, a 46 year old teacher!
```

- **String Concatenation**: This method of combining strings and variables uses the + operator, which can become cumbersome when handling multiple variables or expressions, as seen above.

Template Literals (Modern Way)

A more modern and cleaner way of combining strings and variables is by using **template literals**, introduced in ES6.

```
const hakeemNew = `I'm ${firstName}, a ${year - birthYear} year old ${job}!`;
console.log(hakeemNew);
// Outputs: I'm Hakeem, a 46 year old teacher!
```

- **Template Literals**: These allow you to embed variables and expressions inside strings using ${}. They are enclosed by backticks (`) instead of quotes, making them easier to read and more flexible than the old concatenation method.

Multiline Strings

In older JavaScript, creating multiline strings required using escape characters (\n\ for a new line), which was not very readable.

```
console.log(
  "String with \n\
multiple \n\
lines"
);
// Outputs:
// String with
// multiple
// lines
```

With template literals, you can write multiline strings more easily without escape characters.

```
console.log(`String
with
multiple
lines`);
// Outputs:
// String
// with
// multiple
// lines
```

- **Template Literals for Multiline Strings**: This is a much cleaner way to write strings that span multiple lines, making the code more readable.

Summary

- **Math Operators**: Used for arithmetic operations (e.g., `+`, `-`, `*`, `/`, `**`).
- **Assignment Operators**: Combine math with assignment (e.g., `+=`, `*=`, `++`, `--`).
- **Comparison Operators**: Compare two values and return `true` or `false` (e.g., `>`, `<`, `>=`, `<=`).
- **Operator Precedence**: Determines the order in which operators are evaluated in complex expressions.
- **Strings**: Text data in JavaScript. Use `+` for concatenation or backticks and `${}` for template literals.
- **Template Literals**: Modern and cleaner way to create strings, allowing for embedded expressions and easy multiline strings.

8. If/else Statements

The `if/else` statement is a fundamental decision-making structure in programming. It allows a program to execute different blocks of code depending on the evaluation of a condition.

Structure of if/else

```
if (condition) {
  // Code block executed if the condition is true
} else {
  // Code block executed if the condition is false
}
```

Detailed Breakdown

- **Condition**: This is an expression that evaluates to either `true` or `false`.
- **If block**: The block of code that runs **only** when the condition is `true`.
- **Else block**: If the `if` condition is `false`, the `else` block (if provided) runs instead. It provides an alternate path of execution.

If/else Statements Example

```
let age = 20;

if (age >= 18) {
  console.log("You are an adult");
} else {
  console.log("You are a minor");
}
```

Explanation:

- The condition `age >= 18` checks if the value of `age` is greater than or equal to 18.
- If true (i.e., the person is an adult), the message "You are an adult" is logged.
- If false, the alternative "You are a minor" is logged.

if/else if/else

If you want to check multiple conditions, you can chain `else if` blocks.

```
let score = 85;

if (score >= 90) {
  console.log("Grade A");
} else if (score >= 80) {
  console.log("Grade B");
} else if (score >= 70) {
  console.log("Grade C");
} else {
  console.log("Grade D");
}
```

- Here, the program evaluates multiple conditions in sequence, and the first one that is `true` gets executed.

9. Type Conversion and Coercion

In JavaScript, values can be converted from one data type to another either **manually** (Type Conversion) or **automatically** (Type Coercion).

Type Conversion (Explicit Conversion)

This is when you explicitly convert a value from one type to another using built-in methods like `Number()`, `String()`, `Boolean()`, etc.

Type Conversion Example:

```
```

```
const inputYear = "1991";
console.log(Number(inputYear), inputYear); // Outputs: 1991, "1991"
console.log(Number(inputYear) + 18); // Outputs: 2009
```

- `Number(inputYear)`: Converts the string `'1991'` to the number `1991`.
- Without conversion, `'1991'` is treated as a string, so performing mathematical operations would be impossible. But by using `Number()`, you can now perform numeric calculations.

Other useful conversions:

- `String(value)`: Converts a number or boolean into a string.

  ```
  const age = 30;
  console.log(String(age)); // Outputs: "30"
  ```

- `Boolean(value)`: Converts a value into `true` or `false`.

  ```
  console.log(Boolean(0)); // Outputs: false
  console.log(Boolean("Hello")); // Outputs: true
  ```

Type Coercion (Implicit Conversion)

JavaScript automatically converts types when necessary, particularly when operators are used between different data types. This can lead to some unexpected results.

Type Coercion Example:

```
console.log("23" - "10"); // Outputs: 13
console.log("23" + "10"); // Outputs: "2310"
```

- **Subtraction (`-`)**: JavaScript automatically converts both strings `'23'` and `'10'` to numbers and performs the subtraction, resulting in `13`.
- **Addition (`+`)**: JavaScript concatenates the two strings `'23'` and `'10'`, resulting in the string `"2310"`, because the `+` operator is also used for string concatenation.

10. Equality Operators: `==` vs. `===`

Equality operators are used to compare values in JavaScript. There are two main types: **strict equality** (`===`) and **loose equality** (`==`).

Strict Equality (`===`)

The strict equality operator checks both **value** and **data type**. If both are identical, the result is `true`; otherwise, it's `false`.

Strict Equality Example:

```
const age = "18";
if (age === 18) console.log("You just became an adult :D (strict)");
```

- In this example, `age` is a string (`'18'`), and `18` is a number. Since the data types are different, `age === 18` evaluates to `false`, and the message won't be logged.

Loose Equality (`==`)

The loose equality operator performs type coercion before making a comparison, meaning it converts one of the values to the type of the other before checking for equality.

Loose Equality Example:

```
const age = "18";
if (age == 18) console.log("You just became an adult :D (loose)");
```

- In this case, JavaScript converts the string `'18'` to the number `18` and then compares the values. Since they are now both `18`, this evaluates to `true`, and the message is logged.

Key Differences

- `===` **(strict)**: No type conversion is done. Both the value and the data type must be the same for the comparison to return `true`.
- `==` **(loose)**: JavaScript performs type coercion before comparison, meaning it converts the values if necessary, making it less reliable in some cases.

Best Practice: It's recommended to always use `===` for comparisons to avoid unintended type coercion.

11. Logical Operators

Logical operators are used to combine multiple conditions and return `true` or `false` based on their evaluation.

AND (`&&`) Operator

The `&&` (AND) operator checks if **all** conditions are `true`. If they are, the entire expression evaluates to `true`. If **any** condition is `false`, the result is `false`.

AND (`&&`) Operator Example:

```
const hasDriversLicense = true;
const hasGoodVision = true;
const isTired = false;

if (hasDriversLicense && hasGoodVision && !isTired) {
  console.log("Yinka is able to drive!");
} else {
  console.log("Someone else should drive...");
}
```

- In this example, `hasDriversLicense` and `hasGoodVision` are both `true`, and `!isTired` evaluates to `true` (because `isTired` is `false`, and the `!` operator negates it). Since all conditions are `true`, the message "Yinka is able to drive!" is logged.
- If any of the conditions were `false`, the else block would execute.

OR (`||`) Operator

The `||` (OR) operator checks if **at least one** of the conditions is `true`. If any condition is `true`, the result is `true`. If **all** conditions are `false`, the result is `false`.

OR (`||`) Operator Example:

```
const isWeekend = true;
const isHoliday = false;

if (isWeekend || isHoliday) {
  console.log("You can take a day off!");
} else {
  console.log("You have to work today.");
}
```

- Here, the `isWeekend` is `true`, so the OR operator returns `true` and logs "You can take a day off!" even though `isHoliday` is `false`.

NOT (`!`) Operator

The `!` (NOT) operator inverts the truthiness of a value. If a condition is `true`, applying the `!` operator will make it `false`, and vice versa.

NOT (`!`) Operator Example:

```
const isTired = true;

if (!isTired) {
  console.log("You are ready to go!");
```

```
} else {
  console.log("You should rest.");
}
```

- Since `isTired` is `true`, the `!isTired` becomes `false`, so the second message "You should rest." is logged.

Combining Logical Operators

You can combine `&&`, `||`, and `!` to form more complex conditions.

Combining Logical Operators Example:

```
const hasDriversLicense = true;
const hasGoodVision = true;
const isTired = false;

if (hasDriversLicense && hasGoodVision && !isTired) {
  console.log("Yinka is able to drive!");
} else {
  console.log("Someone else should drive...");
}
```

- `hasDriversLicense && hasGoodVision && !isTired`: This checks if Yinka has a driver's license, good vision, and is not tired. If all conditions are `true`, she can drive. Otherwise, someone else should drive.

12. Switch Statement

The `switch` statement is used for comparing a variable against multiple values, where each value represents a case.

Structure

```
switch (expression) {
  case value1:
    // Code to execute if expression === value1
    break;
  case value2:
    // Code to execute if expression === value2
    break;
  default:
  // Code to execute if none of the cases match
}
```

Explanation

- The `switch` expression is compared against each `case`.
- If a match is found, the corresponding code block runs.
- The `break` statement ensures that the program exits the switch block after a match is found. Without `break`, execution would "fall through" to subsequent cases.
- The `default` case runs if no other cases match.

Switch Statement Example

```
let day = 3;

switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  default:
    console.log("Invalid day");
}
```

- In this example, since `day` is 3, the output is "Wednesday."

Why use a switch over if/else?

- When comparing a single value against many possible matches, `switch` can be more readable and organized than chaining multiple `else if` conditions.

13. Ternary Operator

The ternary operator provides a shorthand way to write simple `if/else` statements in a single line. It has three parts: a condition, a result for `true`, and a result for `false`.

Syntax

```
condition ? expressionIfTrue : expressionIfFalse;
```

Ternary Operator Syntax Explanation

- If the `condition` is `true`, `expressionIfTrue` is executed.

- Otherwise, `expressionIfFalse` is executed.

Ternary Operator Example

```
let age = 20;
let canVote = age >= 18 ? "Yes, you can vote" : "No, you can't vote";
console.log(canVote);
```

- Here, the condition checks if `age >= 18`.
- If true, the message "Yes, you can vote" is assigned to `canVote`. If false, the message "No, you can't vote" is assigned.

Why use the ternary operator?

- It is a concise alternative for simple `if/else` statements.
- It can make the code more readable for straightforward conditions.

Nested Ternary Operators

- Ternary operators can be nested, but overuse can make the code hard to read.

```
let grade = score >= 90 ? "A" : score >= 80 ? "B" : score >= 70 ? "C" : "D";
```

- Here, multiple conditions are checked using the ternary operator, though it's better to use `if/else` for complex logic.

Conclusion

- **If/else** is best for basic decision-making and more complex conditions.
- **Type Conversion**: Manual conversion of one data type to another using `Number()`, `String()`, etc.
- **Type Coercion**: JavaScript automatically converts types when performing certain operations (e.g., `'23' - '10'` results in `13`).
- **Equality Operators**: Use `===` for strict equality (checks value and type) and `==` for loose equality (performs type coercion).
- **Logical Operators**: `&&` (AND), `||` (OR), and `!` (NOT) are used to combine multiple conditions. `&&` requires all conditions to be `true`, `||` requires at least one `true`, and `!` negates the truthiness of a value.
- **Switch statements** are suitable when checking one expression against many possible values.
- The **ternary operator** is a shorthand for simple `if/else` decisions, great for keeping code concise.

Challenges ***Questions:***

Challenge #1: Comparing BMI

Question:
Yinka and Moyo are trying to compare their BMI (Body Mass Index) using the formula:
BMI = mass / height² (or mass / (height * height)), where:

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

1. Store Yinka's and Moyo's mass and height in variables.
2. Calculate their BMIs using the formula (you can even implement both versions).
3. Create a boolean variable `yinkaHigherBMI` to check if Yinka has a higher BMI than Moyo.

Test Data 1:
Yinka: 78 kg, 1.69 m
Moyo: 92 kg, 1.95 m

Test Data 2:
Yinka: 95 kg, 1.88 m
Moyo: 85 kg, 1.76 m

Challenge #2: Improved BMI Comparison

Question:
Using the BMI example from Challenge #1, improve the code by:

1. Printing a console message indicating who has a higher BMI.
2. Using template literals to include BMI values in the output, e.g.,
   "*Yinka's BMI (28.3) is higher than Moyo's (23.9)!*"

Challenge #3: Gymnastics Competition

Question:
Two gymnastics teams, **Hidee** and **Silas**, compete 3 times. The team with the highest average score wins.

1. Calculate the average score for both teams.
2. Compare averages to determine the winner or a draw.

Bonus 1:
Include a **minimum score of 100**. A team only wins if it has a higher average score **and** a score of at least 100.

Bonus 2:
Minimum score also applies to draws. A draw only occurs if both teams have the same average **and** both scores are >= 100.

Test Data:

- Hidee: 96, 108, 89
- Silas: 88, 91, 110

Bonus Test Data 1:

- Hidee: 97, 112, 101
- Silas: 109, 95, 123

Bonus Test Data 2:

- Hidee: 97, 112, 101
- Silas: 109, 95, 106

Challenge #4: Tip Calculator

Question:
Victor wants a tip calculator for restaurant bills. Tips are calculated as follows:

- **15%** for bills between **50 and 300**.
- **20%** for other values.

1. Calculate the tip based on the bill using a ternary operator (avoid `if/else`).
2. Print the bill, tip, and total value in a string, e.g.,
   "*The bill was 275, the tip was 41.25, and the total value was 316.25.*"

Test Data:
Bills: 275, 40, 430