JavaScript Notes

1. JavaScript as a Multi-Paradigm Language

JavaScript is a versatile language that supports multiple programming paradigms, allowing developers to choose the best approach for their needs.

Supported Paradigms:

- **Functional Programming:** Emphasizes the use of functions as the primary building blocks of code. Focuses on pure functions, immutability, and avoiding side effects.
- **Procedural Programming:** Involves writing code as a series of step-by-step instructions or procedures to achieve a desired outcome.
- **Object-Oriented Programming (OOP):** Organizes code around objects and classes, which encapsulate data and behavior. Promotes concepts like inheritance, encapsulation, and polymorphism.

2. Program vs. Process

It's important to distinguish between a program and a process:

- **Program:** The static set of instructions or code written by a developer.
- **Process:** An instance of a program under execution; the dynamic state of the code while it is running.

3. Variables

Variables are fundamental in programming for storing and managing data.

- **Definition:** Containers used to store data values.
- **Mutability:** Variables can be updated and reused throughout the execution of a program.

Naming Rules for Variables:

Allowed Characters: * Letters (a-z, A-Z) * Underscore (`_`) * Dollar sign (`$`) * Camel case (e.g., `myVariableName`)

Not Allowed Characters/Rules: * Digits (cannot start with a digit) * Special characters (e.g., `-`, `!`, `@`, `#`, etc.) * Spaces * Keywords (reserved words in JavaScript)

Example:

```
let name = "hazik";
name = "iqram"; // Variable updated
```

4. Keywords

Keywords are reserved words in JavaScript that have special meanings and cannot be used as variable names.

Examples: `let`, `const`, `var`, `return`, `if`, `else`, `for`, `while`

5. Data Types in JavaScript

JavaScript supports several types of values that can be stored in variables.

Primitive Data Types:

These are atomic in nature and do not depend on other types. They represent single, simple values.

1. **Number:** Represents both integer and floating-point numbers.
    - **Examples:** `10`, `-3`, `3.6`, `3.1415`, `100`, `1000`
2. **String:** Represents textual data. Enclosed in single quotes (`' '`), double quotes (`" "`), or backticks (`` ` ``).
    - **Examples:** `'HAZIK'`, `"iqram"`, `` `wasim` ``
    - Strings are used to refer to text.
3. **Boolean:** Represents logical entities with two possible values.
    - **Values:** `true` or `false` (these are keywords).
4. **Undefined:** A primitive value automatically assigned to variables that have been declared but not yet assigned a value. It signifies that a variable has not been defined yet but might be later.
    - **Example:** `let age; // age is undefined`

5. **Null:** Represents the intentional absence of any object value. It signifies an empty or unknown value.
    - **Difference from `undefined`:** `null` is an assigned value indicating emptiness, whereas `undefined` means a value has not been assigned at all.
    - **Example:** `javascript let a; // a is undefined let b = 10; b = null; // b's value is now intentionally empty`

Non-Primitive Data Types (Objects):

These are compositions of other types and are used to store collections of data or more complex entities.

1. **Objects:** Used to store key-value pairs, where keys are unique identifiers for values.
    - **Example (Facebook User):** `javascript let user = { name: "ahmad", age: 23, // Can be a complex entity post: { createdAt: "Jun 12, 2033", text: "my first post" }, gender: "Male" };`

6. Data Type Examples and `console.log`

```
let marks = 100;
let name = "iqram";
let age; // undefined
let company = null;
let salary = undefined;
let isStudent = false;

console.log("Marks = ", marks);
console.log("Name = ", name);
console.log("Company = ", company);
console.log("Age = ", age);
console.log("Salary = ", salary);
console.log("Is a student = ", isStudent);

// Using objects
let user = {
    name: "hazik",
    company: undefined,
    salary: null,
    age: undefined
};
console.log("Details of user", user);
```

7. Special Characters (Escape Sequences)

Special characters, also known as escape sequences, are used within strings to represent characters that are difficult or impossible to type directly.

- `\n` : Newline character (creates a line break)
- `\t` : Tab character (creates a horizontal tab space)

Examples:

```
let text1 = "the new apple iphone \n has been launched";
console.log(text1);
// Output:
// the new apple iphone
// has been launched

let text2 = "the new apple iphone \t has been launched";
console.log(text2);
// Output:
// the new apple iphone \t has been launched

let age = 10;
let name = "anil";
console.log(name, "\n", age);
// Output:
// anil
// 10
```

⚙️ JavaScript Operators – Deep Dive

JavaScript gives us tools—called operators—to perform actions on values (operands). They can do math, assign values, compare data, or even manipulate bits.

➕ Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations.

Operator	Description	Example	Output
`+`	Addition	`10 + 5`	`15`

Operator	Description	Example	Output
`-`	Subtraction	`10 - 5`	`5`
`*`	Multiplication	`10 * 5`	`50`
`/`	Division	`10 / 5`	`2`
`%`	Modulus (Remainder)	`10 % 3`	`1`
`**`	Exponentiation	`2 ** 3`	`8`

🎯 Operands

Operands are the values that operators act upon.

```
 10 + 3;
// 10 and 3 are operands
// + is the operator
```

📝 Assignment Operators

Used to assign or update the value of a variable.

Operator	Meaning	Example	Equivalent
`=`	Assign	`let a = 10`	-
`+=`	Add and assign	`a += 2`	`a = a + 2`
`-=`	Subtract and assign	`a -= 2`	`a = a - 2`
`*=`	Multiply and assign	`a *= 2`	`a = a * 2`
`/=`	Divide and assign	`a /= 3`	`a = a / 3`
`%=`	Modulus and assign	`a %= 2`	`a = a % 2`

📊 Relational / Comparison Operators

These compare two operands and return a boolean result (`true` or `false`).

Operator	Meaning	Example	Result
`<`	Less than	`3 < 5`	`true`
`>`	Greater than	`3 > 5`	`false`
`<=`	Less than or equal to	`3 <= 3`	`true`
`>=`	Greater than or equal	`3 >= 4`	`false`

🧠 Logical Operators

Used to evaluate boolean logic.

Operator	Name	Description
`&&`	AND	Returns true if **both** operands are true
`||`	OR	Returns true if **at least one** is true
`!`	NOT	Inverts the value

🔎 Analogy

- `&&` = You need **both keys** to open the safe.
- `||` = You can enter through **any open door**.
- `!` = Like flipping a switch from ON to OFF.

⚡ Short-Circuiting

Logical operators **short-circuit** to optimize evaluation:

- `&&`: Returns the **first falsy** value or the last one if all are truthy.
- `||`: Returns the **first truthy** value or the last one if all are falsy.

```
false && console.log("Won't run"); // short-circuits
true || console.log("Won't run");  // short-circuits
```

🚨 Falsy Values in JavaScript

JavaScript considers the following values as **falsy**:

1. `false`
2. `0`, `-0`
3. `""` (empty string)
4. `null`
5. `undefined`
6. `NaN`

Everything else is truthy.

🌐 Type Coercion in Logical Expressions

JavaScript **automatically converts** values into boolean when needed (truthy/falsy checks, conditionals, etc.).

This is **type coercion** in action:

```
if ("hello") console.log("This runs"); // "hello" is truthy
if (0) console.log("Won't run");       // 0 is falsy
```

🧮 Special Numbers in JavaScript

JavaScript has some unique number values to represent edge cases:

Value	Description
`-0`	Captures **directionality** in some calculations
`NaN`	Stands for **Not a Number** (e.g., `"abc" * 2`)
`Infinity`	Result of numbers beyond the upper limit

Value	Description
`-Infinity`	Result of numbers beyond the lower limit

These help JavaScript handle **unexpected numerical behavior gracefully**.

🧮 Bitwise Operators

Bitwise operators work at the **bit level**. JavaScript converts numbers to **32-bit binary**, performs operations, and converts them back.

🧩 Types of Bitwise Operators

Operator	Name	Description
`&`	AND	1 if both bits are 1
`|`	OR	1 if at least one bit is 1
`^`	XOR	1 if bits are different
`~`	NOT	Inverts all bits (unary operator)

🧪 Example – `5 & 7`

```
 5  = 101
7  = 111
-----------
&  = 101 → 5
```

🔍 `typeof` Operator

Returns the **type of a value** as a string.

```
typeof 42;         // 'number'
typeof "hello";    // 'string'
typeof true;       // 'boolean'
```

🚨 Special Case: `typeof null`

- Returns `'object'`
- This is a **legacy bug** in JavaScript
- `null` is actually a **primitive**, not an object

🔁 `typeof` vs `instanceof`

Feature	`typeof`	`instanceof`
Use Case	Type of value	Is object an instance of class?
Works On	Primitives + objects	Objects only
Output	String	Boolean
Example	`typeof 42 // "number"`	`arr instanceof Array // true`
Limitation	Arrays are also 'object'	Doesn’ t work with primitives

---

🧪 Equality Operators

Operator	Name	Description
`==`	Loose Equality	Compares values **with** type coercion
`===`	Strict Equality	Compares values **without** coercion (type + value)

```
1 == "1";  // true
1 === "1"; // false
```

⚖️ Equality Operators in JavaScript

Key Notes:

- `==` **(Abstract Equality)**:
- Called "loose equality" informally, but officially **Abstract Equality** per the ECMAScript spec.

- Avoid in modern code—coercion rules can be unpredictable (e.g., `[] == ![]` → `true`).

- `===` **(Strict Equality)**:

- No coercion; checks type and value. Safer and more intuitive.

🔹 **Best Practice**: Default to `===` unless you explicitly need coercion (e.g., checking `null` vs. `undefined`).

Why the Terminology Matters:

- **Abstract Equality**: Emphasizes the hidden steps (ToNumber, ToPrimitive) in the comparison.
- **Strict Equality**: Explicitly rejects coercion.

This version:
1. Uses the official ECMAScript term ("Abstract Equality").
2. Adds a "When to Use" column for practicality.
3. Clarifies the spec vs. colloquial names.
4. Includes a real-world example of coercion pitfalls.
5. Keeps it concise but actionable.
