Session 1

1. Development Environment & Programming Language Basics

1.1 What Is a Programming Language?

At its core, a programming language is how we give instructions to a computer. Not in human language, computers don't understand that. Instead, we speak to them using structured, logical commands like:

- "Add 5 to this number."
- "Store this result for later."
- "If this condition is true, do something."

Every programming language is built around three main ideas:

- **Syntax** – the rules for how code is written
- **Semantics** – the meaning behind what you write
- **Execution** – how the code runs (interpreter, compiler, etc.)

You can think of a programming language as a bridge between human logic and machine behaviour. It's how we turn ideas into actions the computer can perform.

1.2 Components of a Programming Language

Regardless of the syntax style or ecosystem, most programming languages share a common set of building blocks:

Concept	Purpose
Literals	Raw values like `"hello"` or `42`
Data Types	Define what kind of data it is (string, number, boolean)
Variables	Store data to reuse later
Operators	Perform operations like `+`, `-`, `==`
Control Flow	Make decisions with `if`, `else`, `switch`, etc.

Concept	Purpose
Loops	Repeat actions using `for`, `while`, etc.
Functions	Group logic into reusable blocks
Input/Output	Communicate with the outside world (user, files, APIs)

Every language, from Python to Go to JavaScript relies on these core concepts. The syntax may change, but the logic stays the same.

1.2 What is JavaScript (JS)?

JavaScript is a programming language used to add interactivity to websites. Without it, websites would just be static text and images.

You use JavaScript to:

- Validate forms
- Handle user actions (like clicks or typing)
- Show/hide elements dynamically
- Fetch and display data without refreshing the page

JavaScript is the language that brings life to websites. Think interactive buttons, popups, dropdowns, and animations. It was originally designed to run in the browse, but with the introduction of Node.js, we can now use JavaScript to build server-side applications too.

- **Frontend**: Runs in browsers
- **Backend**: Runs on servers using Node.js

1.3 What is Node.js?

Node.js is a **JavaScript runtime,** that means it lets you run JS code **outside** a web browser.

Normally, JS runs inside a browser (like Chrome). But what if you want to run JS on your laptop, server, or cloud machine? That's where Node.js comes in.

It lets you:

- Build web servers
- Read/write files

- Connect to databases
- Handle backend logic

And all this using JavaScript.

Node.js is a **runtime environment**.

That means: it gives your JavaScript code a place to run **outside the browser**.

Think of it like this:

- The browser is one place where JavaScript can run.
- Node.js is another place where it can run—like your computer's terminal.

So, when you write `console.log("Hello")` and run it using Node, it's Node.js that reads the JS and runs it line by line.

Without a runtime, JavaScript would just be text—it needs something to run it.

1.4 Your First JavaScript File

Create a file called `hello.js`:

```
// hello.js
console.log("Hello World");
```

To run it:

```
node hello.js
```

You should see:

```
Hello World
```

1.5 Output

To show results or debug values in JavaScript, we use `console.log()`.

It simply prints data to the terminal or browser console.

```
console.log("Hello World");
```

Output:

```
Hello World
```

You can also log variables or results of expressions:

```
let a = 5;
let b = 10;
console.log(a + b);       // 15
console.log("Sum:", a + b); // Sum: 15
```

Output:

```
15
Sum:15
```

1.6 Input Using process.argv (Command Line Arguments)

When you run a Node.js file, you can pass inputs directly through the terminal.

These values are available in the `process.argv` array.

```
//input.js
let name = process.argv[2];
console.log("Hello", name);
```

Run it like this in terminal:

```
node input.js Ani
```

This is what the process.argv array looks like now

```
process.argv = [
  '/path/to/node',      // process.argv[0]
  '/path/to/input.js',  // process.argv[1]
  'Ani'                 // process.argv[2]
];
```

Output:

```
Hello Ani
```

Explanation:

- `process.argv[0]` → node path
- `process.argv[1]` → file name
- `process.argv[2]` → first user input

Useful for quick test inputs.

1.7 Input Using prompt-sync (Interactive User Input)

Node.js doesn't have a built-in `prompt`, but we can use a package called `prompt-sync`.

1. First, install it:

```
npm install prompt-sync
```

1. Then use it like this:

```
const prompt = require("prompt-sync")();
let name = prompt("Enter your name: ");
console.log("Hello", name);
```

Sample Output:

```
Enter your name: Ani
Hello Ani
```

🧠 Best when you want real-time user input from the terminal.

1.8 Literals

Literals are the fixed values you assign directly to variables.

```
let name = "Ani";      // string literal
let age = 22;          // number literal
let isCool = true;     // boolean literal
let score = null;      // null literal
let notDefined;        // undefined (not assigned yet)
let result = age + 10;
console.log(result);
```

Output:

```
32
```

1.9 Data Types

- **String** – text (`"Hello"`)
- **Number** – integers or floats
- **Boolean** – `true` or `false`
- **Null** – intentionally empty
- **Undefined** – declared but not assigned

1.10 Truthy and Falsy Values

JavaScript treats some values as true/false when used in conditions.

Truthy Values

These behave like `true`:

```
if ("hello") {
  console.log("This is truthy!");
}
```

- `"text"` (non-empty strings)
- `42` , `1` (non-zero numbers)
- `[]` (empty array)

- `{}` (empty object)
- functions

Falsy Values

These behave like `false`:

```
if(0) {
  console.log("This won't run");
}
```

- `false`
- `0`
- `""` (empty string)
- `null`
- `undefined`
- `NaN`

1.11 let vs const

`let` and `const` are used to declare variables in JavaScript.

- Use `let` when the value **might change** later.
- Use `const` when the value **should stay the same** after it's set.

```
let a = 5;
a = 10; // ✅ Reassignable

const b = 3;
// b = 4; ❌ Error: Assignment to constant variable
```

- **let** → block-scoped, reassignable
- **const** → block-scoped, **not** reassignable
- Neither can be re-declared in the same scope

1.12 Operators

Operators

are symbols that perform actions on values or variables.

```
let x = 5;
let y = 2;
console.log(x + y); // 7
console.log(x * y); // 10
console.log(x ** y)// 25
console.log(x > y); // true
```

Output:

```
7
10
25
true
```

1.13 Output – Template Literals vs Concatenation

There are **two ways** to combine text and variables when printing:

Using `+` (Concatenation)

```
let name = "Ani";
console.log("Hello " + name);
```

Output:

```
Hello Ani
```

Using Template Literals (Backticks ``)

```
let name = "Ani";
let age = 22;
console.log(`Hello ${name}, you are ${age} years old.`);
```

Output:

```
Hello Ani, you are 22 years old.
```

Difference

Method	Style	Pros
`+`	Concatenation	Good for quick and short strings
`` `${} ``	Template literal	Cleaner for dynamic or multi-line strings

You can even do calculations or expressions inside template literals:

```
let a = 5, b = 10;
console.log(`Sum = ${a + b}`);
```

Output:

```
Sum = 15
```

1.14 Other Data Types

```
let x;                  // undefined
let y = null;           // null
let big = 123456789012345n;   // bigint
let sym = Symbol("id");       // symbol
let arr = [1, 2, 3];          // array
let obj = { name: "Ani" };    // object
```

- **undefined** → declared but no value
- **null** → intentional “nothing”
- **bigint** → very large integers (`n` at the end)
- **symbol** → unique identifier
- **array** → ordered list
- **object** → key-value pair

1.15 Conditional Statements

Conditional statements are used to make decisions in your code based on certain conditions.

Use `if`, `else if`, and `else` to control what gets executed.

Example with `if / else if / else`

Only one block will run—the first condition that matches.

```
let score = 75;

if (score >= 90) {
  console.log("Excellent");
} else if (score >= 60) {
  console.log("Good");
} else {
  console.log("Needs Improvement");
}
```

Output:

```
Good
```

Example with only `if` statements

All matching `if` conditions will run, even if one already did.

```
let score = 90;

if (score >= 90) {
  console.log("Excellent");
}
if (score >= 75) {
  console.log("Good");
}
if (score < 75) {
```

```
  console.log("Needs Improvement");
}
```

Output:

```
Excellent
Good
```

Explanation:

In the second example, all `if` conditions are checked independently. Since we didn't use `else if`, both `score >= 90` and `score >= 75` are true—so both blocks run.

Use `else if` when you want **only one block to run** out of many.

Use separate `if`s when **multiple blocks can run** if their conditions are true.

1.16 Equality Checks

JavaScript gives you two ways to compare values:

- `==` → compares values (does type conversion if needed)
- `===` → compares values **and** types (strict)

```
console.log(5 == "5");      // true – value is same after type conversion
console.log(5 === "5");     // false – number and string are different

console.log(null == undefined);   // true – both mean "nothing"
console.log(null === undefined);  // false – different types
```

Output:

```
true
false
true
false
```

- `==` → loose equality (type coercion happens)
- `===` → strict equality (checks type + value)

SSSSO Vidyakshina 2025