1. Introduction to Java

- **Java:** A high-level, class-based, object-oriented programming language designed to have as few implementation dependencies as possible, enabling "Write Once, Run Anywhere" (WORA).
- **Platform Independence (WORA):** The ability of Java code to be compiled into platform-neutral bytecode, which can then be executed on any device with a Java Virtual Machine (JVM).
- **JVM (Java Virtual Machine):** The runtime environment that executes Java bytecode, acting as an interpreter for the `.class` files.
- **JRE (Java Runtime Environment):** A software package that includes the JVM and core libraries necessary to run Java applications.
- **JDK (Java Development Kit):** A comprehensive software development kit that includes the JRE along with development tools like the Java compiler (`javac`) and debugger.

---

2. Java Basics: Syntax Rules

- **Case-Sensitivity:** Java differentiates between uppercase and lowercase letters; `myVariable` is distinct from `MyVariable`.
- **Statement Terminator:** The semicolon (`;`) used to mark the end of a single executable instruction.
- **Code Blocks:** Groups of statements enclosed within curly braces (`{}`), used to define classes, methods, and control flow structures.
- **Comments:** Non-executable lines in the code used for explanation and documentation (`//` for single-line, `/* ... */` for multi-line, `/** ... */` for Javadoc).

---

3. Variables and Data Types

- **Variable:** A named memory location used to store a data value, which must be declared with a specific data type before use.
- **Data Type:** A classification that specifies the type of values a variable can hold, determining its memory allocation and valid operations.
    - **Primitive Data Types:** Basic, built-in data types that directly store values (e.g., `int`, `double`, `boolean`, `char`).
    - **Non-Primitive (Reference) Data Types:** Data types that store references (memory addresses) to objects rather than the actual values (e.g., `String`, Arrays, Classes).

- **`String`:** A non-primitive data type representing an immutable sequence of characters.

---

4. Operators

- **Operator:** A special symbol that performs operations on one or more values (operands).
- **Arithmetic Operators:** Perform mathematical calculations (`+`, `-`, `*`, `/`, `%`).
- **Assignment Operators:** Assign values to variables (`=`, `+=`, `-=`, etc.).
- **Relational (Comparison) Operators:** Compare two values and return a `boolean` result (`==`, `!=`, `<`, `>`, `<=`, `>=`).
- **Logical Operators:** Combine or modify boolean expressions (`&&` (AND), `||` (OR), `!` (NOT)).
- **Increment/Decrement Operators:** Increase or decrease a variable's value by one (`++`, `--`).
- **Ternary Operator:** A shorthand conditional operator that evaluates a boolean expression and returns one of two values based on the result (`condition ? valueIfTrue : valueIfFalse`).

---

5. Control Flow Statements

- **Control Flow Statement:** An instruction that determines the order in which individual statements or blocks of code are executed.
- **Conditional Statements:**
  - **`if-else if-else`:** Executes a block of code based on whether a condition is `true` or `false`, providing multiple decision paths.
  - **`switch`:** Selects one of many code blocks to execute based on the value of a single expression, often used as an alternative to a long `if-else if` chain.
- **Looping Statements:**
  - **`for` loop:** Repeats a block of code a specific, predetermined number of times.
  - **`while` loop:** Repeats a block of code as long as a specified condition remains `true`, checking the condition *before* each iteration.
  - **`do-while` loop:** Repeats a block of code as long as a specified condition remains `true`, guaranteeing at least one execution of the block as the condition is checked *after* each iteration.

    - **`for-each` loop (Enhanced For Loop):** A simplified `for` loop designed for iterating over elements in arrays or collections without explicitly managing an index.
- **Branching Statements:**
    - **`break`:** Terminates the innermost loop or `switch` statement, transferring control to the statement immediately following it.
    - **`continue`:** Skips the current iteration of a loop and proceeds to the next iteration.
    - **`return`:** Exits the current method and, if the method has a non-`void` return type, provides a value back to the caller.

---

6. Object-Oriented Programming (OOP) Fundamentals

- **OOP (Object-Oriented Programming):** A programming paradigm based on the concept of "objects," which can contain data (attributes) and code (methods) that operate on that data.
- **Class:** A blueprint or template that defines the structure (attributes) and behavior (methods) common to all objects created from it.
- **Object:** An instance of a class; a concrete entity created from a class blueprint, occupying memory.
- **Method:** A block of code within a class that performs a specific task or defines a behavior for an object.
- **Constructor:** A special method within a class, having the same name as the class and no return type, used to initialize new objects when they are created using the `new` keyword.

---

7. Arrays

- **Array:** A fixed-size, sequential collection of elements of the same data type, accessed via a numerical index starting from 0.

---

8. Encapsulation (Pillar of OOP)

- **Encapsulation:** The bundling of data and the methods that operate on that data into a single unit (a class), and the restriction of direct access to some of an object's

components (data hiding), typically achieved using `private` access modifiers and `public` getter/setter methods.

---

9. Inheritance (Pillar of OOP)

- **Inheritance:** A mechanism where one class (subclass) acquires the properties and behaviors (fields and methods) of another class (superclass), modeling an "is-a" relationship and promoting code reusability.
- **`extends` keyword:** Used by a subclass to declare its inheritance from a superclass.
- **`super` keyword:** Used to refer to members (fields, methods, constructors) of the immediate superclass.

---

10. Polymorphism (Pillar of OOP)

- **Polymorphism:** Meaning "many forms"; it allows objects of different classes to be treated as objects of a common type, where the specific method called depends on the actual object's type at runtime.
    - **Compile-time (Static) Polymorphism / Method Overloading:** Occurs when multiple methods within the same class share the same name but have different parameter lists, with the correct method determined at compile time.
    - **Run-time (Dynamic) Polymorphism / Method Overriding:** Occurs when a subclass provides its own specific implementation for a method already defined in its superclass, with the actual method called determined at runtime based on the object's type.

---

11. Abstraction (Pillar of OOP)

- **Abstraction:** The process of hiding complex implementation details and showing only the essential features or functionality to the user, focusing on "what" an object does rather than "how" it does it.
    - **Abstract Class:** A class that cannot be instantiated directly and may contain abstract (unimplemented) methods; subclasses must implement these abstract methods.
    - **Interface:** A blueprint of a class that defines a set of methods that a class must implement, providing a contract for behavior; it can also contain default and static methods (Java 8+).